

SISTEMAS DISTRIBUÍDOS

Conceitos e Projeto

5^a Edição

George Coulouris
Jean Dollimore
Tim Kindberg
Gordon Blair



George Coulouris é cientista visitante sênior do Laboratório de Computação da Cambridge University.

Jean Dollimore foi, até sua aposentadoria, conferencista sênior sobre ciência da computação do Queen Mary College, University of London.

Tim Kindberg foi pesquisador sênior do Hewlett-Packard Laboratories, em Bristol. É o fundador e atual diretor da matter 2 media.

Gordon Blair é professor de sistemas distribuídos e chefe do Departamento de Computação da Lancaster University.



S623 Sistemas distribuídos [recurso eletrônico] : conceitos e projeto /
George Coulouris ... [et al.] ; tradução: João Eduardo
Nóbrega Tortello ; revisão técnica: Alexandre Carissimi. –
5. ed. – Dados eletrônicos. – Porto Alegre : Bookman, 2013.

Editado também como livro impresso em 2013.
ISBN 978-85-8260-054-2

1. Computação. 2. Sistemas distribuídos. 3. Redes.
I. Coulouris, George.

CDU 004.652.3

George Coulouris

Cambridge University

Jean Dollimore

*ex-professor do
Queen Mary College,
University of London*

Tim Kindberg

matter 2 media

Gordon Blair

Lancaster University

SISTEMAS DISTRIBUÍDOS

Conceitos e Projeto

5^a Edição

Tradução:

João Eduardo Nóbrega Tortello

Revisão técnica:

Alexandre Carissimi

Doutor em Informática pelo Institut National Polytechnique de Grenoble, França
Professor do Instituto de Informática da UFRGS

Versão impressa
desta obra: 2013



2013

Obra originalmente publicada sob o título
Distributed Systems: Concepts and Design, 5th Edition
ISBN 978-0-13-214301-1

Authorized translation from the English language edition, entitled DISTRIBUTED SYSTEMS: CONCEPTS AND DESIGN, 5th Edition by GEORGE COULOURIS; JEAN DOLLIMORE; TIM KINDBERG; GORDON BLAIR, published by Pearson Education, Inc., publishing as Addison-Wesley, Copyright © 2012. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Portuguese language edition published by Bookman Companhia Editora Ltda, a company of Grupo A Educação S.A., Copyright © 2013.

Tradução autorizada da edição de língua inglesa, intitulada DISTRIBUTED SYSTEMS: CONCEPTS AND DESIGN, 5th Edition, autoria de GEORGE COULOURIS; JEAN DOLLIMORE; TIM KINDBERG; GORDON BLAIR, publicado por Pearson Education, Inc., sob o selo Addison-Wesley, Copyright © 2012. Todos os direitos reservados. Este livro não poderá ser reproduzido nem em parte nem na íntegra, nem ter partes ou sua íntegra armazenado em qualquer meio, seja mecânico ou eletrônico, inclusive fotocópia, sem permissão da Pearson Education, Inc.

A edição em língua portuguesa desta obra é publicada por Bookman Companhia Editora Ltda, uma empresa do Grupo A Educação S.A., Copyright © 2013.

Gerente editorial: *Arysinha Jacques Affonso*

Colaboraram nesta edição:

Editora: *Mariana Belloli*

Capa: *VS Digital*, arte sobre capa original

Leitura final: *Amanda Jansson Breitsameter*

Editoração: *Techbooks*

Reservados todos os direitos de publicação, em língua portuguesa, à
BOOKMAN EDITORA LTDA., uma empresa do GRUPO A EDUCAÇÃO S.A.
Av. Jerônimo de Ornelas, 670 – Santana
90040-340 – Porto Alegre – RS
Fone: (51) 3027-7000 Fax: (51) 3027-7070

É proibida a duplicação ou reprodução deste volume, no todo ou em parte, sob quaisquer formas ou por quaisquer meios (eletrônico, mecânico, gravação, fotocópia, distribuição na Web e outros), sem permissão expressa da Editora.

Unidade São Paulo
Av. Embaixador Macedo Soares, 10.735 – Pavilhão 5 – Cond. Espace Center
Vila Anastácio – 05095-035 – São Paulo – SP
Fone: (11) 3665-1100 Fax: (11) 3667-1333

SAC 0800 703-3444 – www.grupoa.com.br

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

Prefácio

Esta 5^a edição de nosso livro-texto aparece em um momento em que a Internet e a Web continuam a crescer e têm impacto em cada aspecto de nossa sociedade. Por exemplo, o capítulo introdutório registra a influência da Internet em áreas de aplicação tão diversificadas como finanças e comércio, artes e entretenimento, e o surgimento da sociedade da informação de modo geral. Destaca também os requisitos bastante exigentes de domínios de aplicação, como pesquisa na Web e jogos *online* para vários participantes. Do ponto de vista dos sistemas distribuídos, esses desenvolvimentos estão impondo novas demandas significativas na infraestrutura de sistema subjacente, em termos de variedade de aplicações, cargas de trabalho e tamanho suportados por muitos sistemas modernos. Tendências importantes incluem a diversidade e a onipresença cada vez maiores de tecnologias de interconexão em rede (incluindo a importância cada vez maior das redes sem fio), a integração inerente de elementos da computação móvel e ubíqua na infraestrutura dos sistemas distribuídos, levando a arquiteturas físicas radicalmente diferentes, à necessidade de suportar serviços multimídia e ao surgimento do paradigma da computação em nuvem que desafia nossa perspectiva de serviços de sistemas distribuídos.

O objetivo deste livro é fornecer um entendimento sobre os princípios nos quais são baseados a Internet e outros sistemas distribuídos, sobre sua arquitetura, algoritmos e projeto e sobre como atendem às exigências dos aplicativos distribuídos atuais. Começamos com um conjunto de sete capítulos que, juntos, abordam os elementos básicos para o estudo dos sistemas distribuídos. Os dois primeiros capítulos fornecem um panorama conceitual do tema, destacando as características dos sistemas distribuídos e os desafios que devem ser enfrentados em seu projeto: escalabilidade, heterogeneidade, segurança e tratamento de falhas são os mais importantes. Esses capítulos também desenvolvem modelos abstratos para entender interação, falha e segurança de processos. Depois deles, existem outros capítulos básicos dedicados ao estudo de interconexão em rede, comunicação entre processos, invocação remota, comunicação indireta e suporte do sistema operacional.

Novidades da 5^a edição

Novos capítulos:

Comunicação indireta: comunicação de grupo, sistemas de publicar-assinar e estudos de caso sobre JavaSpaces, JMS, WebSphere e Message Queues.

Objetos e componentes distribuídos: *middleware* baseado em componentes e estudos de caso sobre Enterprise JavaBeans, Fractal e CORBA.

Projeto de sistemas distribuídos: um novo estudo de caso sobre a infraestrutura Google.

Novos tópicos: computação em nuvem, virtualização de rede, virtualização de sistema operacional, interface de passagem de mensagem, *peer-to-peer* não estruturado, espaços de tupla, acoplamento livre em relação a serviços Web.

Novos estudos de caso: Skype, Gnutella, TOTA, L²imbo, BitTorrent, End System Multicast.

Veja a tabela na página ix para mais detalhes das mudanças.

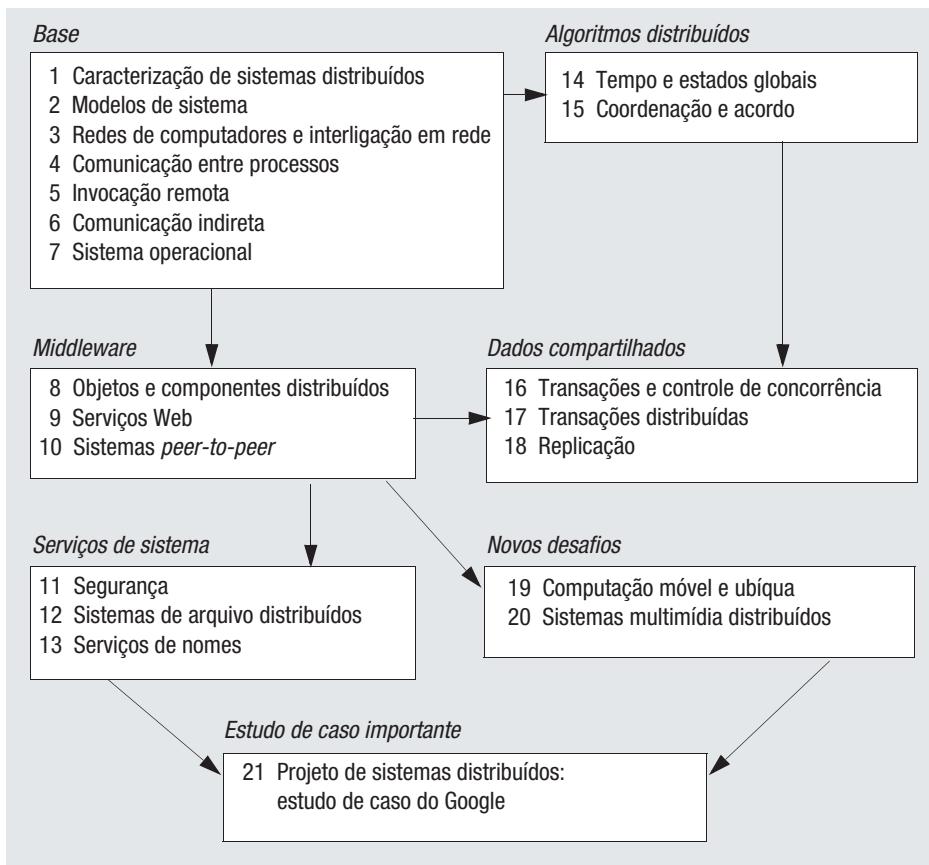
O grupo de capítulos seguinte aborda o importante tópico do *middleware*, examinando diferentes estratégias para suportar aplicativos distribuídos, incluindo objetos e componentes distribuídos, *Web services* e soluções *peer-to-peer* alternativas. Abordamos, em seguida, os tópicos bem-estabelecidos de segurança, sistemas de arquivos distribuídos e atribuição de nomes distribuída, antes de passarmos para os importantes aspectos relacionados a dados que incluem transações distribuídas e replicação. Os algoritmos associados a todos esses tópicos são abordados à medida que surgem, e também em capítulos separados, dedicados a temporização, coordenação e acordo.

O livro termina com os capítulos que tratam das áreas emergentes da computação móvel e ubíqua e dos sistemas multimídia distribuídos, antes de apresentar um estudo de caso enfocando o projeto e a implementação da infraestrutura de sistemas distribuídos que dão suporte ao Google em termos de funcionalidade de pesquisa básica e da crescente variedade de serviços adicionais oferecidos pelo Google (por exemplo, Gmail e Google Earth). O último capítulo tem um papel importante ao ilustrar como todos os conceitos de arquiteturas de sistemas, algoritmos e tecnologias apresentados neste livro podem ser reunidos em um projeto global coerente para determinado domínio de aplicação.

Objetivos e público-alvo

Este livro se destina a cursos de graduação e cursos introdutórios de pós-graduação. Ele pode ser igualmente usado de forma autodidata. Adotamos a estratégia *top-down* (de cima para baixo), tratando dos problemas a serem resolvidos no projeto de sistemas distribuídos e descrevendo as estratégias bem-sucedidas na forma de modelos abstratos, algoritmos e estudos de caso detalhados sobre os sistemas mais utilizados. Abordamos a área com profundidade e amplitude suficientes para permitir aos leitores prosseguirem com seus estudos usando a maioria dos artigos de pesquisa presentes na literatura sobre sistemas distribuídos.

Nosso objetivo é tornar o assunto acessível a estudantes que tenham um conhecimento básico de programação orientada a objetos, sistemas operacionais e arquitetura de computadores. O livro inclui uma abordagem dos aspectos de redes de computadores, relevantes aos sistemas distribuídos, inclusive as tecnologias subjacentes da Internet, das redes de longa distância, locais e sem fio. Ao longo de todo o livro são apresentados algoritmos e interfaces em Java ou, em alguns casos, em C ANSI. Por brevidade e clareza da apresentação, também é usada uma forma de pseudocódigo, derivado de Java/C.



Organização do livro

O diagrama mostra os capítulos sob sete áreas de assunto principais. Seu objetivo é fornecer um guia para a estrutura do livro e indicar as rotas de navegação recomendadas para os instrutores que queiram fornecer (ou leitores que queiram obter) um entendimento sobre as várias subáreas do projeto de sistemas distribuídos.

Referências

A existência da World Wide Web mudou a maneira pela qual um livro como este pode ser vinculado ao material de origem, incluindo artigos de pesquisa, especificações técnicas e padrões. Muitos dos documentos de origem agora estão disponíveis na Web; alguns estão disponíveis somente nela. Por motivos de brevidade e facilidade de leitura, empregamos uma forma especial de referência ao material da Web, que se assemelha um pouco a um URL: citações como www.omg.org e www.rsasecurity.com [I] se referem à documentação que está disponível somente na Web. Elas podem ser pesquisadas na lista de referências ao final do livro, mas os URLs completos são dados apenas em uma versão *online* dessa lista, no site do livro www.cdk5.net/refs (em inglês), em que assumem a forma de *links* e podem ser acessados diretamente com um simples clique de mouse. As duas versões da lista de referências incluem uma explicação mais detalhada desse esquema.

Alterações relativas à 4^a edição

Antes de começarmos a escrever esta nova edição, fizemos um levantamento junto aos professores que utilizavam a 4^a edição. A partir dos resultados, identificamos o novo material exigido e as várias alterações a serem feitas. Além disso, reconhecemos a diversidade cada vez maior de sistemas distribuídos, particularmente em termos da variedade de estratégias de arquiteturas de sistemas disponíveis atualmente para desenvolvedores de sistemas distribuídos. Isso exigiu alterações significativas no livro, especialmente nos capítulos iniciais (básicos).

Isso nos levou a escrever três capítulos totalmente novos, a fazer alterações significativas em vários outros capítulos e a realizar numerosas inserções por todo o livro para acrescentar material novo. Muitos dos capítulos foram alterados para refletir as novas informações que se tornaram disponíveis sobre os sistemas descritos. Essas alterações estão resumidas na tabela da página ix. Para ajudar os professores que usaram a 4^a edição, quando possível preservamos a estrutura adotada na edição anterior.

O material removido está disponível no site do livro (em inglês), junto ao material removido das edições anteriores. Isso inclui os estudos de caso sobre ATM, comunicação entre processos em UNIX, CORBA (uma versão reduzida desse estudo de caso permanece no Capítulo 8), a especificação de eventos distribuídos Jini, o capítulo sobre memória compartilhada distribuída e o estudo de caso do *middleware Grid* (apresentando OGSA e o *toolkit Globus*). Alguns capítulos do livro abrangem muito material; por exemplo, o novo capítulo sobre comunicação indireta (Capítulo 6). Os professores podem optar por abordar o amplo espectro, antes de escolherem duas ou três técnicas para examinar com mais detalhes (por exemplo, a comunicação em grupo, em razão de seu predomínio nos sistemas distribuídos comerciais).

A ordem dos capítulos foi alterada para acomodar o novo material e refletir a mudança na importância relativa de alguns tópicos. Para uma maior compreensão de alguns tópicos, talvez os leitores considerem necessário buscar outras referências. Por exemplo, há material no Capítulo 9, sobre técnicas de segurança em XML, que farão mais sentido depois que as seções do Capítulo 11, “Segurança”, às quais faz referência, forem lidas.

Agradecimentos

Somos muito gratos aos seguintes professores que participaram de nosso levantamento: Guohong Cao, Jose Fortes, Bahram Khalili, George Blank, Jinsong Ouyang, JoAnne Holliday, George K, Thiruvathukal, Joel Wein, Tao Xie e Xiaobo Zhou.

Gostaríamos de agradecer às seguintes pessoas que revisaram os capítulos novos ou ajudaram substancialmente: Rob Allen, Roberto Baldoni, John Bates, Tom Berson, Lynne Blair, Geoff Coulson, Paul Grace, Andrew Herbert, David Hutchison, Laurent Mathy, Rajiv Ramdhany, Richard Sharp, Jean-Bernard Stefani, Rip Sohan, Francois Taiani, Peter Triantafillou, Gareth Tyson e Sir Maurice Wilkes. Também gostaríamos de agradecer ao pessoal do Google que deu ideias sobre o fundamento lógico da Google Infrastructure, especialmente Mike Burrows, Tushar Chandra, Walfredo Cirne, Jeff Dean, Sanjay Ghemawat, Andrea Kirmse e John Reumann.

Capítulos novos:

6 Comunicação indireta	Inclui eventos e notificação da 4ª edição
8 Objetos e componentes distribuídos	Inclui versão resumida do estudo de caso CORBA da 4ª edição
21 Projeto de sistemas distribuídos	Inclui um novo estudo de caso sobre o Google

Capítulos que passaram por alterações significativas:

1 Caracterização de sistemas distribuídos	<i>Reestruturação de material</i> Nova Seção 1.2: Exemplos de sistemas distribuídos Seção 1.2.2: Introdução da computação em nuvem MMOGs
2 Modelos de sistema	<i>Reestruturação de material</i> Nova Seção 2.2: Modelos físicos Seção 2.3: reescrita significativa para refletir novo conteúdo do livro e perspectivas de arquiteturas de sistema associadas
4 Comunicação entre processos	<i>Várias atualizações</i> Comunicação cliente-servidor movida para o Capítulo 5 Nova Seção 4.5: Virtualização de rede (inclui estudo de caso sobre Skype) Nova Seção 4.6: Estudo de caso sobre MPI Estudo de caso sobre IPC no UNIX removido
5 Invocação remota	<i>Reestruturação de material</i> Comunicação cliente-servidor movida para este capítulo Progressão de comunicação cliente-servidor por meio de RPC para RMI introduzida Eventos e notificação movidos para o Capítulo 6

Capítulos nos quais novo material foi adicionado/removido, mas sem alterações estruturais:

3 Redes de computadores e interligação em rede	<i>Várias atualizações</i> Seção 3.5: material sobre ATM removido
7 Sistemas operacionais	Nova Seção 7.7: Virtualização em nível de sistema operacional
9 Serviços Web	Seção 9.2: adicionada discussão sobre acoplamento livre
10 Sistemas peer-to-peer	Nova Seção 10.5.3: <i>peer-to-peer</i> não estruturado
15 Coordenação e acordo	Material sobre comunicação em grupo movido para o Capítulo 6
18 Replicação	Material sobre comunicação em grupo movido para o Capítulo 6
19 Móvel seção	Seção 19.3.1: novo material sobre espaços de tupla (TOTÁ e L ² imbo)
20 Sistemas multimídia distribuídos	Seção 20.6: novos estudos de caso sobre BitTorrent e End System Multicast

Site

Acesse o material complementar (em inglês) do livro no site do Grupo A:

- Entre no site do Grupo A, em www.grupoa.com.br.
- Clique em “Acesse ou crie a sua conta”.
- Se você já tem cadastro em nosso site, insira seu endereço de e-mail ou CPF e sua senha na área “Acesse sua conta”; se ainda não é cadastrado, cadastre-se preenchendo o campo da área “Crie sua conta”.
- Depois de acessar a sua conta, digite o título do livro ou o nome do autor no campo de busca do site e clique no botão “Buscar”.
- Localize o livro entre as opções oferecidas e clique sobre a imagem de capa ou sobre o título para acessar a página do livro.

Na página do livro:

- Para fazer download dos códigos-fonte dos Capítulos 4, 5, 8, 9 e 11, clique no link “Conteúdo online”.
- Para fazer download de apresentações em PowerPoint para uso em sala de aula, apresentações em PowerPoint para os exercícios e soluções dos exercícios (todos em inglês), clique no link “Material para o Professor”. Atenção: este conteúdo é de acesso restrito a usuários com cadastro de “Professor”.

Os autores do livro também mantêm um site (em inglês) com uma ampla variedade de material destinado a ajudar professores e leitores. Esse site pode ser acessado em: www.cdk5.net

Guia do instrutor: dicas de ensino capítulo a capítulo, sugestões de projetos de laboratório e muito mais.

Lista de referências: a lista de referências que pode ser encontrada no final do livro está reproduzida no site. A versão Web da lista de referências inclui *links* para o material que está disponível *online*.

Lista de errata: uma lista dos erros conhecidos no livro, com suas correções. Tais erros serão corrigidos nas novas impressões e uma lista de errata separada será fornecida para cada impressão.*

Material suplementar: o código-fonte dos programas presentes no livro e o material de leitura relevante que foi apresentado nas edições anteriores do livro, mas removido por razões de espaço. As referências a esse material suplementar aparecem no livro com *links* como www.cdk5.net/ipc (o URL para material suplementar relacionado com comunicação entre processos).

*George Coulouris
Jean Dollimore
Tim Kindberg
Gordon Blair
<authors@cdk5.net>*

* A errata disponível no site da edição em língua inglesa já está corrigida nesta edição brasileira.

Sumário

1 Caracterização de Sistemas Distribuídos	1
1.1 Introdução	2
1.2 Exemplos de sistemas distribuídos	3
1.3 Tendências em sistemas distribuídos	8
1.4 Enfoque no compartilhamento de recursos	14
1.5 Desafios	16
1.6 Estudo de caso: a World Wide Web	26
1.7 Resumo	33
2 Modelos de Sistema	37
2.1 Introdução	38
2.2 Modelos físicos	39
2.3 Modelos de arquitetura para sistemas distribuídos	40
2.4 Modelos fundamentais	61
2.5 Resumo	76
3 Redes de Computadores e Interligação em Rede	81
3.1 Introdução	82
3.2 Tipos de redes	86
3.3 Conceitos básicos de redes	89
3.4 Protocolos Internet	106
3.5 Estudos de caso: Ethernet, WiFi e Bluetooth	128
3.6 Resumo	141
4 Comunicação Entre Processos	145
4.1 Introdução	146
4.2 A API para protocolos Internet	147
4.3 Representação externa de dados e empacotamento	158
4.4 Comunicação por multicast (difusão seletiva)	169
4.5 Virtualização de redes: redes de sobreposição	174
4.6 Estudo de caso: MPI	178
4.7 Resumo	181

5 Invocação Remota	185
5.1 Introdução	186
5.2 Protocolos de requisição-resposta	187
5.3 Chamada de procedimento remoto	195
5.4 Invocação a método remoto	204
5.5 Estudo de caso: RMI Java	217
5.6 Resumo	225
6 Comunicação Indireta	229
6.1 Introdução	230
6.2 Comunicação em grupo	232
6.3 Sistemas publicar-assinar	242
6.4 Filas de mensagem	254
6.5 Estratégias de memória compartilhada	262
6.6 Resumo	274
7 Sistema Operacional	279
7.1 Introdução	280
7.2 A camada do sistema operacional	281
7.3 Proteção	284
7.4 Processos e threads	286
7.5 Comunicação e invocação	303
7.6 Arquiteturas de sistemas operacionais	314
7.7 Virtualização em nível de sistema operacional	318
7.8 Resumo	331
8 Objetos e Componentes Distribuídos	335
8.1 Introdução	336
8.2 Objetos distribuídos	337
8.3 Estudo de caso: CORBA	340
8.4 De objetos a componentes	358
8.5 Estudos de caso: Enterprise JavaBeans e Fractal	364
8.6 Resumo	378

9 Serviços Web	381
9.1 Introdução	382
9.2 Serviços Web	384
9.3 Descrições de serviço e IDL para serviços Web	400
9.4 Um serviço de diretório para uso com serviços Web	404
9.5 Aspectos de segurança em XML	406
9.6 Coordenação de serviços Web	411
9.7 Aplicações de serviços Web	413
9.8 Resumo	420
10 Sistemas Peer-to-peer	423
10.1 Introdução	424
10.2 Napster e seu legado	428
10.3 Middleware para peer-to-peer	430
10.4 Sobreposição de roteamento	433
10.5 Estudos de caso: Pastry, Tapestry	436
10.6 Estudo de caso: Squirrel, OceanStore, Ivy	449
10.7 Resumo	459
11 Segurança	463
11.1 Introdução	464
11.2 Visão geral das técnicas de segurança	472
11.3 Algoritmos de criptografia	484
11.4 Assinaturas digitais	493
11.5 Criptografia na prática	500
11.6 Estudos de caso: Needham–Schroeder, Kerberos, TLS, 802.11 WiFi	503
11.7 Resumo	518
12 Sistemas de Arquivos Distribuídos	521
12.1 Introdução	522
12.2 Arquitetura do serviço de arquivos	530
12.3 Estudo de caso: Sun Network File System	536
12.4 Estudo de caso: Andrew File System	548
12.5 Aprimoramentos e mais desenvolvimentos	557
12.6 Resumo	563

13 Serviço de Nomes	565
13.1 Introdução	566
13.2 Serviços de nomes e o Domain Name System	569
13.3 Serviços de diretório	584
13.4 Estudo de caso: Global Name Service	585
13.5 Estudo de caso: X.500 Directory Service	588
13.6 Resumo	592
14 Tempo e Estados Globais	595
14.1 Introdução	596
14.2 Relógios, eventos e estados de processo	597
14.3 Sincronização de relógios físicos	599
14.4 Tempo lógico e relógios lógicos	606
14.5 Estados globais	610
14.6 Depuração distribuída	619
14.7 Resumo	625
15 Coordenação e Acordo	629
15.1 Introdução	630
15.2 Exclusão mútua distribuída	633
15.3 Eleições	641
15.4 Coordenação e acordo na comunicação em grupo	646
15.5 Consenso e problemas relacionados	659
15.6 Resumo	670
16 Transações e Controle de Concorrência	675
16.1 Introdução	676
16.2 Transações	679
16.3 Transações aninhadas	690
16.4 Travas	692
16.5 Controle de concorrência otimista	707
16.6 Ordenação por carimbo de tempo	711
16.7 Comparação dos métodos de controle de concorrência	718
16.8 Resumo	720

17 Transações Distribuídas	727
17.1 Introdução	728
17.2 Transações distribuídas planas e aninhadas	728
17.3 Protocolos de confirmação atômica	731
17.4 Controle de concorrência em transações distribuídas	740
17.5 Impasses distribuídos	743
17.6 Recuperação de transações	751
17.7 Resumo	761
18 Replicação	765
18.1 Introdução	766
18.2 Modelo de sistema e o papel da comunicação em grupo	768
18.3 Serviços tolerantes a falhas	775
18.4 Estudos de caso de serviços de alta disponibilidade: Gossip, Bayou e Coda	782
18.5 Transações em dados replicados	802
18.6 Resumo	814
19 Computação Móvel e Ubíqua	817
19.1 Introdução	818
19.2 Associação	827
19.3 Interoperabilidade	835
19.4 Percepção e reconhecimento de contexto	844
19.5 Segurança e privacidade	857
19.6 Adaptabilidade	866
19.7 Estudo de caso: Cooltown	871
19.8 Resumo	878
20 Sistemas Multimídia Distribuídos	881
20.1 Introdução	882
20.2 Características dos dados multimídia	886
20.3 Gerenciamento de qualidade de serviço	887
20.4 Gerenciamento de recursos	897
20.5 Adaptação de fluxo	899
20.6 Estudos de caso: Tiger, BitTorrent e End System Multicast	901
20.7 Resumo	913

21 Projeto de Sistemas Distribuídos – Estudo de Caso: Google	915
21.1 Introdução	916
21.2 Introdução ao estudo de caso: Google	917
21.3 Arquitetura global e filosofia de projeto	922
21.4 Paradigmas de comunicação	928
21.5 Serviços de armazenamento de dados e coordenação	935
21.6 Serviços de computação distribuída	956
21.7 Resumo	964
 Referências	 967
Índice	1025

1

Caracterização de Sistemas Distribuídos

- 1.1 Introdução
- 1.2 Exemplos de sistemas distribuídos
- 1.3 Tendências em sistemas distribuídos
- 1.4 Enfoque no compartilhamento de recursos
- 1.5 Desafios
- 1.6 Estudo de caso: a World Wide Web
- 1.7 Resumo

Um sistema distribuído é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens. Essa definição leva às seguintes características especialmente importantes dos sistemas distribuídos: concorrência de componentes, falta de um relógio global e falhas de componentes independentes.

Examinaremos vários exemplos de aplicações distribuídas modernas, incluindo pesquisa na Web, jogos *online* para vários jogadores e sistemas de negócios financeiros. Veremos também as tendências básicas que estimulam o uso dos sistemas distribuídos atuais: a natureza pervasiva da interligação em rede moderna, o florescimento da computação móvel e ubíqua, a crescente importância dos sistemas multimídia distribuídos e a tendência no sentido de considerar os sistemas distribuídos como um serviço público. Em seguida, o capítulo destacará o compartilhamento de recursos como uma forte motivação para a construção de sistemas distribuídos. Os recursos podem ser gerenciados por servidores e acessados por clientes, ou podem ser encapsulados como objetos e acessados por outros objetos clientes.

Os desafios advindos da construção de sistemas distribuídos são a heterogeneidade dos componentes, ser um sistema aberto, o que permite que componentes sejam adicionados ou substituídos, a segurança, a escalabilidade – isto é, a capacidade de funcionar bem quando a carga ou o número de usuários aumenta –, o tratamento de falhas, a concorrência de componentes, a transparência e o fornecimento de um serviço de qualidade. Por fim, a Web será discutida como exemplo de sistema distribuído de grande escala e serão apresentados seus principais recursos.

1.1 Introdução

As redes de computadores estão por toda parte. A Internet é uma delas, assim como as muitas redes das quais ela é composta. Redes de telefones móveis, redes corporativas, redes de fábrica, redes em campus, redes domésticas, redes dentro de veículos, todas elas, tanto separadamente como em conjunto, compartilham as características básicas que as tornam assuntos relevantes para estudo sob o título *sistemas distribuídos*. Neste livro, queremos explicar as características dos computadores interligados em rede que afetam os projetistas e desenvolvedores de sistema e apresentar os principais conceitos e técnicas que foram criados para ajudar nas tarefas de projeto e implementação de sistemas que os têm por base.

Definimos um sistema distribuído como aquele no qual os componentes de *hardware* ou *software*, localizados em computadores interligados em rede, comunicam-se e coordenam suas ações apenas enviando mensagens entre si. Essa definição simples abrange toda a gama de sistemas nos quais computadores interligados em rede podem ser distribuídos de maneira útil.

Os computadores conectados por meio de uma rede podem estar separados por qualquer distância. Eles podem estar em continentes separados, no mesmo prédio ou na mesma sala. Nossa definição de sistemas distribuídos tem as seguintes consequências importantes:

Concorrência: em uma rede de computadores, a execução concorrente de programas é a norma. Posso fazer meu trabalho em meu computador, enquanto você faz o seu em sua máquina, compartilhando recursos como páginas Web ou arquivos, quando necessário. A capacidade do sistema de manipular recursos compartilhados pode ser ampliada pela adição de mais recursos (por exemplo, computadores) na rede. Vamos descrever como essa capacidade extra pode ser distribuída em muitos pontos de maneira útil. A coordenação de programas em execução concorrente e que compartilham recursos também é um assunto importante e recorrente.

Inexistência de relógio global: quando os programas precisam cooperar, eles coordenam suas ações trocando mensagens. A coordenação frequentemente depende de uma noção compartilhada do tempo em que as ações dos programas ocorrem. Entretanto, verifica-se que existem limites para a precisão com a qual os computadores podem sincronizar seus relógios em uma rede – não existe uma noção global única do tempo correto. Essa é uma consequência direta do fato de que a *única* comunicação se dá por meio do envio de mensagens em uma rede. Exemplos desses problemas de sincronização e suas soluções serão descritos no Capítulo 14.

Falhas independentes: todos os sistemas de computador podem falhar, e é responsabilidade dos projetistas de sistema pensar nas consequências das possíveis falhas. Nos sistemas distribuídos, as falhas são diferentes. Falhas na rede resultam no isolamento dos computadores que estão conectados a ela, mas isso não significa que eles param de funcionar. Na verdade, os programas neles existentes talvez não consigam detectar se a rede falhou ou se ficou demasiadamente lenta. Analogamente, a falha de um computador ou o término inesperado de um programa em algum lugar no sistema (um *colapso no sistema*) não é imediatamente percebida pelos outros componentes com os quais ele se comunica. Cada componente do sistema pode falhar independentemente, deixando os outros ainda em funcionamento. As consequências dessa característica dos sistemas distribuídos serão um tema recorrente em todo o livro.

A principal motivação para construir e usar sistemas distribuídos é proveniente do desejo de compartilhar recursos. O termo “recurso” é bastante abstrato, mas caracteriza

bem o conjunto de coisas que podem ser compartilhadas de maneira útil em um sistema de computadores interligados em rede. Ele abrange desde componentes de *hardware*, como discos e impressoras, até entidades definidas pelo *software*, como arquivos, bancos de dados e objetos de dados de todos os tipos. Isso inclui o fluxo de quadros de vídeo proveniente de uma câmera de vídeo digital ou a conexão de áudio que uma chamada de telefone móvel representa.

O objetivo deste capítulo é transmitir uma visão clara da natureza dos sistemas distribuídos e dos desafios que devem ser enfrentados para garantir que eles sejam bem-sucedidos. A Seção 1.2 fornece alguns exemplos ilustrativos de sistemas distribuídos, e a Seção 1.3 aborda as principais tendências subjacentes que estimulam os recentes desenvolvimentos. A Seção 1.4 enfoca o projeto de sistemas com compartilhamento de recursos, enquanto a Seção 1.5 descreve os principais desafios enfrentados pelos projetistas de sistemas distribuídos: heterogeneidade, sistemas abertos, segurança, escalabilidade, tratamento de falhas, concorrência, transparência e qualidade do serviço. A Seção 1.6 apresenta o estudo de caso detalhado de um sistema distribuído bastante conhecido, a World Wide Web, ilustrando como seu projeto suporta o compartilhamento de recursos.

1.2 Exemplos de sistemas distribuídos

O objetivo desta seção é dar exemplos motivacionais de sistemas distribuídos atuais, ilustrando seu papel predominante e a enorme diversidade de aplicações associadas a eles.

Conforme mencionado na Introdução, as redes estão por toda parte e servem de base para muitos serviços cotidianos que agora consideramos naturais; por exemplo, a Internet e a World Wide Web associada a ela, a pesquisa na Web, os jogos *online*, os *e-mails*, as redes sociais, o *e-Commerce* etc. Para ilustrar ainda mais esse ponto, considere a Figura 1.1, que descreve uma variedade de setores de aplicação comercial ou social importantes, destacando alguns usos associados estabelecidos ou emergentes da tecnologia de sistemas distribuídos.

Como se vê, os sistemas distribuídos abrangem muitos dos desenvolvimentos tecnológicos mais significativos atualmente e, portanto, um entendimento da tecnologia subjacente é absolutamente fundamental para o conhecimento da computação moderna. A figura também dá um vislumbre inicial da ampla variedade de aplicações em uso hoje, desde sistemas de localização relativa, conforme os encontrados, por exemplo, em um carro ou em um avião, até sistemas de escala global envolvendo milhões de nós; desde serviços voltados para dados até tarefas que exigem uso intenso do processador; desde sistemas construídos a partir de sensores muito pequenos e relativamente primitivos até aqueles que incorporam elementos computacionais poderosos; desde sistemas embarcados até os que suportam uma sofisticada experiência interativa do usuário e assim por diante.

Vamos ver agora exemplos de sistemas distribuídos mais específicos para ilustrar melhor a diversidade e a real complexidade da provisão de sistemas distribuídos atual.

1.2.1 Pesquisa na Web

A pesquisa na Web tem se destacado como um setor crescente na última década, com os valores recentes indicando que o número global de pesquisas subiu para mais de 10 bilhões por mês. A tarefa de um mecanismo de pesquisa na Web é indexar todo o conteúdo da World Wide Web, abrangendo uma grande variedade de estilos de informação, incluindo páginas Web, fontes de multimídia e livros (escaneados). Essa é uma tarefa muito complexa, pois as estimativas atuais mostram que a Web consiste em mais de 63 bilhões

<i>Finanças e comércio</i>	O crescimento do <i>e-Commerce</i> , exemplificado por empresas como Amazon e eBay, e as tecnologias de pagamento subjacentes, como PayPal; o surgimento associado de operações bancárias e negócios <i>online</i> e também os complexos sistemas de disseminação de informações para mercados financeiros.
<i>A sociedade da informação</i>	O crescimento da World Wide Web como repositório de informações e de conhecimento; o desenvolvimento de mecanismos de busca na Web, como Google e Yahoo, para pesquisar esse amplo repositório; o surgimento de bibliotecas digitais e a digitalização em larga escala de fontes de informação legadas, como livros (por exemplo, Google Books); a importância cada vez maior do conteúdo gerado pelos usuários por meio de sites como YouTube, Wikipedia e Flickr; o surgimento das redes sociais por meio de serviços como Facebook e MySpace.
<i>Setores de criação e entretenimento</i>	O surgimento de jogos <i>online</i> como uma forma nova e altamente interativa de entretenimento; a disponibilidade de músicas e filmes nos lares por meio de centros de mídia ligados em rede e mais amplamente na Internet, por intermédio de conteúdo que pode ser baixado ou em <i>streaming</i> ; o papel do conteúdo gerado pelos usuários (conforme mencionado anteriormente) como uma nova forma de criatividade, por exemplo por meio de serviços como YouTube; a criação de novas formas de arte e entretenimento permitidas pelas tecnologias emergentes (inclusive em rede).
<i>Assistência médica</i>	O crescimento da informática médica como disciplina, com sua ênfase nos registros eletrônicos de pacientes <i>online</i> e nos problemas de privacidade relacionados; o papel crescente da telemedicina no apoio ao diagnóstico remoto ou a serviços mais avançados, como cirurgias remotas (incluindo o trabalho colaborativo entre equipes médicas); a aplicação cada vez maior de interligação em rede e tecnologia de sistemas incorporados em vida assistida; por exemplo, para monitorar idosos em suas próprias residências.
<i>Educação</i>	O surgimento do <i>e-learning</i> por meio, por exemplo, de ferramentas baseadas na Web, como os ambientes de ensino virtual; suporte associado ao aprendizado à distância; suporte ao aprendizado colaborativo ou em comunidade.
<i>Transporte e logística</i>	O uso de tecnologias de localização, como o GPS, em sistemas de descoberta de rotas e sistemas de gerenciamento de tráfego mais gerais; o próprio carro moderno como exemplo de sistema distribuído complexo (também se aplica a outras formas de transporte, como a aviação); o desenvolvimento de serviços de mapa baseados na Web, como MapQuest, Google Maps e Google Earth.
<i>Ciências</i>	O surgimento das grades computacionais (<i>grid</i>) como tecnologia fundamental para eScience, incluindo o uso de redes de computadores complexas para dar suporte ao armazenamento, à análise e ao processamento de dados científicos (frequentemente em volumes muito grandes); o uso associado das grades como tecnologia capacitadora para a colaboração entre grupos de cientistas do mundo inteiro.
<i>Gerenciamento ambiental</i>	O uso de tecnologia de sensores (interligados em rede) para monitorar e gerenciar o ambiente natural; por exemplo, para emitir alerta precoce de desastres naturais, como terremotos, enchentes ou tsunamis, e para coordenar a resposta de emergência; o cotejamento e a análise de parâmetros ambientais globais para entender melhor fenômenos naturais complexos, como a mudança climática.

Figura 1.1 Domínios de aplicação selecionados e aplicações de rede associadas.

de páginas e um trilhão de endereços Web únicos. Como a maioria dos mecanismos de busca analisa todo o conteúdo da Web e depois efetua um processamento sofisticado nesse banco de dados enorme, essa tarefa representa, por si só, um grande desafio para o projeto de sistemas distribuídos.

O Google, líder de mercado em tecnologia de pesquisa na Web, fez um trabalho significativo no projeto de uma sofisticada infraestrutura de sistema distribuído para dar suporte à pesquisa (e, na verdade, a outros aplicativos e serviços do Google, como o Google Earth). Isso representa uma das maiores e mais complexas instalações de sistemas distribuídos da história da computação e, assim, exige um exame minucioso. Os destaques dessa infraestrutura incluem:

- Uma infraestrutura física subjacente consistindo em grandes números de computadores interligados em rede, localizados em centros de dados por todo o mundo.
- Um sistema de arquivos distribuído projetado para suportar arquivos muito grandes e fortemente otimizado para o estilo de utilização exigido pela busca e outros aplicativos do Google (especialmente a leitura de arquivos em velocidades altas e constantes).
- Um sistema associado de armazenamento distribuído estruturado que oferece rápido acesso a conjuntos de dados muito grandes.
- Um serviço de bloqueio que oferece funções de sistema distribuído, como bloqueio e acordo distribuídos.
- Um modelo de programação que suporta o gerenciamento de cálculos paralelos e distribuídos muito grandes na infraestrutura física subjacente.

Mais detalhes sobre os serviços de sistemas distribuídos do Google e o suporte de comunicação subjacente podem ser encontrados no Capítulo 21, que apresenta um interessante estudo de caso de um sistema distribuído moderno em ação.

1.2.2 Massively multiplayer online games (MMOGs)

Os jogos *online* com vários jogadores, ou MMOGs (Massively Multiplayer Online Games), oferecem uma experiência imersiva com a qual um número muito grande de usuários interage com um mundo virtual persistente pela Internet. Os principais exemplos desses jogos incluem o EverQuest II, da Sony, e o EVE Online, da empresa finlandesa CCP Games. Esses mundos têm aumentado significativamente em termos de sofisticação e agora incluem, por exemplo, arenas de jogo complexas (o EVE Online, por exemplo, consiste em um universo com mais de 5.000 sistemas estelares) e variados sistemas sociais e econômicos. O número de jogadores também está aumentando, com os sistemas capazes de suportar mais de 50.000 usuários *online* simultâneos (e o número total de jogadores talvez seja dez vezes maior).

A engenharia dos MMOGs representa um grande desafio para as tecnologias de sistemas distribuídos, particularmente devido à necessidade de tempos de resposta rápidos para preservar a experiência dos usuários do jogo. Outros desafios incluem a propagação de eventos em tempo real para muitos jogadores e a manutenção de uma visão coerente do mundo compartilhado. Portanto, esse é um excelente exemplo dos desafios enfrentados pelos projetistas dos sistemas distribuídos modernos.

Foram propostas várias soluções para o projeto de MMOGs:

- Talvez surpreendentemente, o maior jogo *online*, o EVE Online, utiliza uma arquitetura *cliente-servidor* na qual uma única cópia do estado do mundo é mantida em um servidor centralizado e acessada por programas clientes em execução nos consoles

ou em outros equipamentos dos jogadores. Para suportar grandes números de clientes, por si só o servidor é uma entidade complexa, consistindo em uma arquitetura de agrregado (*cluster*) caracterizada por centenas de nós de computador (essa estratégia cliente-servidor está discutida com mais detalhes na Seção 1.4 e as estratégias de *cluster* estão discutidas na Seção 1.3.4). A arquitetura centralizada ajuda significativamente no gerenciamento do mundo virtual e a cópia única também diminui as preocupações com a coerência. Assim, o objetivo é garantir resposta rápida por meio da otimização de protocolos de rede e também para os eventos recebidos. Para suportar isso, a carga é particionada por meio da alocação de *sistemas estelares* individuais para computadores específicos dentro do *cluster*, com os sistemas estelares altamente carregados tendo seu próprio computador dedicado e outros compartilhando um computador. Os eventos recebidos são direcionados para os computadores certos dentro do *cluster* por intermédio do monitoramento da movimentação dos jogadores entre os sistemas estelares.

- Outros MMOGs adotam arquiteturas mais distribuídas, nas quais o universo é particionado por um número potencialmente muito grande de servidores, os quais também pode estar geograficamente distribuídos. Então, os usuários são alocados dinamicamente a um servidor em particular com base nos padrões de utilização momentâneos e também pelos atrasos de rede até o servidor (com base na proximidade geográfica, por exemplo). Esse estilo de arquitetura, que é adotado pelo EverQuest, é naturalmente extensível pela adição de novos servidores.
- A maioria dos sistemas comerciais adota um dos dois modelos apresentados anteriormente, mas agora os pesquisadores também estão examinando arquiteturas mais radicais, que não se baseiam nos princípios cliente-servidor, mas adotam estratégias completamente descentralizadas, baseadas em tecnologia *peer-to-peer*, em que cada participante contribui com recursos (armazenamento e processamento) para hospedar o jogo. Mais considerações sobre as soluções *peer-to-peer* estão nos Capítulos 2 e 10).

1.2.3 Negócios financeiros

Como um último exemplo, examinemos o suporte dos sistemas distribuídos para os mercados de negócios financeiros. Há muito tempo, o setor financeiro está na vanguarda da tecnologia de sistemas distribuídos, em particular por sua necessidade de acesso em tempo real a uma ampla variedade de fontes de informação (preços atuais e tendências de ações, mudanças econômicas e políticas, etc). O setor emprega monitoramento automatizado e aplicativos comerciais (veja a seguir).

Note que a ênfase de tais sistemas está na comunicação e no processamento de itens de interesse, conhecidos como *eventos* nos sistemas distribuídos, e também na necessidade de distribuir eventos de forma confiável e de maneira oportuna para uma quantidade de clientes que têm interesse declarado nesses itens de informação. Exemplos de tais eventos incluem queda no preço de uma ação, o comunicado dos últimos resultados do desemprego e assim por diante. Isso exige um estilo de arquitetura subjacente muito diferente dos estilos mencionados anteriormente (por exemplo, cliente-servidor), e esses sistemas normalmente empregam os que são conhecidos como *sistemas distribuídos baseados em eventos*. Apresentamos uma ilustração de uso típico de tais sistemas a seguir e voltaremos a esse importante tópico com mais profundidade no Capítulo 6.

A Figura 1.2 ilustra um sistema de negócios financeiros típico. Ela mostra uma série de fontes de evento envolvidas em determinada instituição financeira. Essas fontes de

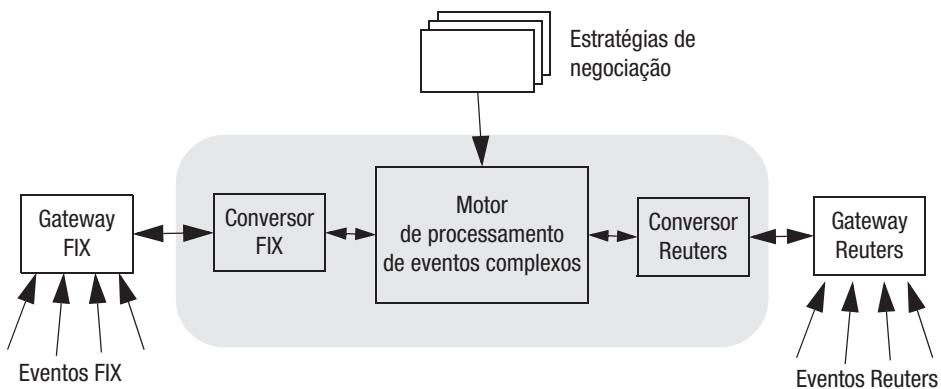


Figura 1.2 Um exemplo de sistema de negócios financeiros.

evento compartilham as seguintes características. Primeiramente, as fontes normalmente aparecem em formatos variados, como eventos de dados de mercado da Reuters e eventos FIX (que seguem o formato específico do protocolo Financial Information eXchange), e provêm de diferentes tecnologias de evento, ilustrando assim o problema da heterogeneidade encontrado na maioria dos sistemas distribuídos (consulte também a Seção 1.5.1). A figura mostra o uso de conversores que transformam formatos heterogêneos em um formato interno comum. Em segundo lugar, o sistema de negócios deve lidar com uma variedade de fluxos de evento, todos chegando em alta velocidade e frequentemente exigindo processamento em tempo real para se detectar padrões que indiquem oportunidades de negócios. Esse processo costumava ser manual, mas as pressões competitivas levaram a uma automação cada vez maior, nos termos do que é conhecido como CEP (Complex Event Processing), o qual oferece uma maneira de reunir ocorrências de evento em padrões lógicos, temporais ou espaciais.

Essa abordagem é usada principalmente no desenvolvimento de estratégias de negócios personalizadas, abrangendo tanto a compra como a venda de ações, em particular procurando padrões que indiquem uma oportunidade de negócio, respondendo automaticamente por fazer e gerenciar pedidos. Como exemplo, considere o seguinte *script*:

```

WHEN
    MSFT price moves outside 2% of MSFT Moving Average
FOLLOWED-BY (
    MyBasket moves up by 0.5%
    AND
        HPQ's price moves up by 5%
        OR
        MSFT's price moves down by 2%
    )
)
ALL WITHIN
    any 2 minute time period
THEN
    BUY MSFT
    SELL HPQ

```

Esse *script* é baseado na funcionalidade fornecida pelo Apama [www.progress.com], um produto comercial do mundo financeiro desenvolvido originalmente pela pesquisa feita na Universidade de Cambridge. Esse *script* detecta uma sequência temporal complexa, baseada nos preços de ações da Microsoft, da HP e uma série de outros preços de ação, resultando em decisões no sentido de comprar ou vender ações específicas.

Esse estilo de tecnologia está sendo cada vez mais usado em outras áreas dos sistemas financeiros, incluindo o monitoramento de atividade do negócio para o gerenciamento de risco (em particular, o controle de exposição), a garantia do cumprimento de normas e o monitoramento de padrões de atividade que possam indicar transações fraudulentas. Nesses sistemas, os eventos normalmente são interceptados e submetidos ao que é equivalente a um *firewall* de cumprimento e risco, antes de serem processados (consulte também a discussão sobre *firewalls* na Seção 1.3.1, a seguir).

1.3 Tendências em sistemas distribuídos

Os sistemas distribuídos estão passando por um período de mudança significativa e isso pode ser consequência de diversas tendências influentes:

- O surgimento da tecnologia de redes pervasivas.
- O surgimento da computação ubíqua, combinado ao desejo de suportar mobilidade do usuário em sistemas distribuídos.
- A crescente demanda por serviços multimídia.
- A visão dos sistemas distribuídos como um serviço público.

1.3.1 Interligação em rede pervasiva e a Internet moderna

A Internet moderna é um conjunto de redes de computadores interligadas, com uma variedade de tipos que aumenta cada vez mais e que agora inclui, por exemplo, uma grande diversidade de tecnologias de comunicação sem fio, como WiFi, WiMAX, Bluetooth (consulte o Capítulo 3) e redes de telefonia móvel de 3^a e 4^a geração. O resultado é que a interligação em rede se tornou um recurso pervasivo, e os dispositivos podem ser conectados (se assim for desejado) a qualquer momento e em qualquer lugar. A Figura 1.3 ilustra uma parte típica da Internet. Os programas que estão em execução nos computadores conectados a ela interagem enviando mensagens através de um meio de comunicação comum. O projeto e a construção dos mecanismos de comunicação da Internet (os protocolos Internet) são uma realização técnica importante, permitindo que um programa em execução em qualquer lugar envie mensagens para programas em qualquer outro lugar e abstraindo a grande variedade de tecnologias mencionadas anteriormente.

A Internet é um sistema distribuído muito grande. Ela permite que os usuários, onde quer que estejam, façam uso de serviços como a World Wide Web, *e-mail* e transferência de arquivos. Na verdade, às vezes, a Web é confundida como sendo a Internet. O conjunto de serviços da Internet é aberto – ele pode ser ampliado pela adição de computadores servidores e de novos tipos de serviços. A Figura 1.3 mostra um conjunto de intranets – sub-redes operadas por empresas e outras organizações, normalmente protegidas por *firewalls*. A função de um *firewall* é proteger uma intranet, impedindo a entrada

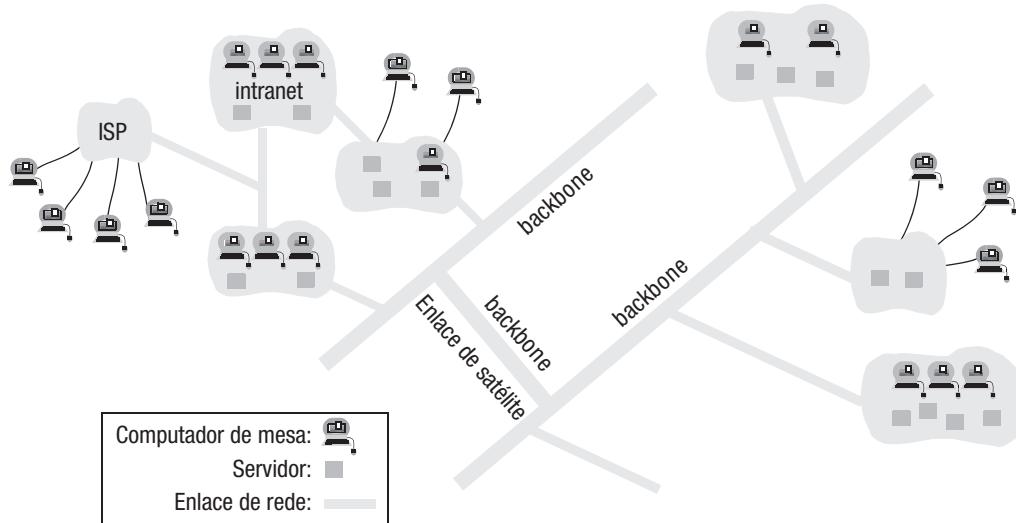


Figura 1.3 Uma parte típica da Internet.

ou a saída de mensagens não autorizadas. Um *firewall* é implementado pela filtragem de mensagens recebidas e enviadas, por exemplo, de acordo com sua origem ou destino. Um *firewall* poderia, por exemplo, permitir que somente mensagens relacionadas com *e-mail* e acesso a Web entrem ou saiam da intranet que ele está protegendo. Os provedores de serviços de Internet (ISPs, Internet Service Providers) são empresas que fornecem, através de *links* de banda larga e de outros tipos de conexões, o acesso a usuários individuais, ou organizações, à Internet. Esse acesso permite que os usuários utilizem os diferentes serviços disponibilizados pela Internet. Os provedores fornecem ainda alguns serviços locais, como *e-mail* e hospedagem de páginas Web. As intranets são interligadas por meio de *backbones*. Um *backbone* é um enlace de rede com uma alta capacidade de transmissão, empregando conexões via satélite, cabos de fibra óptica ou outros meios físicos de transmissão que possuam uma grande largura de banda.

Note que algumas empresas talvez não queiram conectar suas redes internas na Internet. Por exemplo, a polícia e outros órgãos de segurança e ordem pública provavelmente têm pelo menos algumas intranets internas que estão isoladas do mundo exterior (o *firewall* mais eficiente possível – a ausência de quaisquer conexões físicas com a Internet). Os *firewalls* também podem ser problemáticos em sistemas distribuídos por impedir o acesso legítimo a serviços, quando é necessário o compartilhamento de recursos entre usuários internos e externos. Assim, os *firewalls* frequentemente devem ser complementados por mecanismos e políticas mais refinados, conforme discutido no Capítulo 11.

A implementação da Internet e dos serviços que ela suporta tem acarretado o desenvolvimento de soluções práticas para muitos problemas dos sistemas distribuídos (incluindo a maior parte daqueles definidos na Seção 1.5). Vamos destacar essas soluções ao longo de todo o livro, apontando sua abrangência e suas limitações quando for apropriado.

1.3.2 Computação móvel e ubíqua

Os avanços tecnológicos na miniaturização de dispositivos e interligação em rede sem fio têm levado cada vez mais à integração de equipamentos de computação pequenos e portáteis com sistemas distribuídos. Esses equipamentos incluem:

- Computadores *notebook*.
- Aparelhos portáteis, incluindo telefones móveis, *smartphones*, equipamentos com GPS, *pgers*, assistentes digitais pessoais (PDAs), câmeras de vídeo e digitais.
- Aparelhos acoplados ao corpo, como relógios de pulso inteligentes com funcionalidade semelhante ao de um PDA.
- Dispositivos incorporados em aparelhos, como máquinas de lavar, aparelhos de som de alta fidelidade, carros, geladeiras, etc.

A portabilidade de muitos desses dispositivos, junto a sua capacidade de se conectar convenientemente com redes em diferentes lugares, tornam a *computação móvel* possível. Computação móvel é a execução de tarefas de computação enquanto o usuário está se deslocando de um lugar a outro ou visitando lugares diferentes de seu ambiente usual. Na computação móvel, os usuários que estão longe de suas intranets “de base” (a intranet do trabalho ou de sua residência) podem acessar recursos por intermédio dos equipamentos que carregam consigo. Eles podem continuar a acessar a Internet, os recursos em sua intranet de base e, cada vez mais, existem condições para que os usuários utilizem recursos como impressoras ou mesmo pontos de venda, que estão convenientemente próximos, enquanto transitam. Esta última possibilidade também é conhecida como *computação com reconhecimento de localização* ou *com reconhecimento de contexto*. A mobilidade introduz vários desafios para os sistemas distribuídos, incluindo a necessidade de lidar com a conectividade variável e mesmo com a desconexão, e a necessidade de manter o funcionamento em face da mobilidade do aparelho (consulte a discussão sobre transparência da mobilidade, na Seção 1.5.7).

A *computação ubíqua*, também denominada *computação pervasiva*, é a utilização de vários dispositivos computacionais pequenos e baratos, que estão presentes nos ambientes físicos dos usuários, incluindo suas casas, escritórios e até na rua. O termo “pervasivo” se destina a sugerir que pequenos equipamentos de computação finalmente se tornarão tão entrinhados nos objetos diários que mal serão notados. Isto é, seu comportamento computacional será transparente e intimamente vinculado à sua função física. Por sua vez, o termo “ubíquo” dá a noção de que o acesso a serviços de computação está onipresente, isto é, disponível em qualquer lugar.

A presença de computadores em toda parte só se torna útil quando eles podem se comunicar uns com os outros. Por exemplo, pode ser conveniente para um usuário controlar sua máquina de lavar e seu aparelho de som de alta fidelidade a partir de um único dispositivo universal de controle remoto em sua casa. Da mesma forma, seria cômodo a máquina de lavar enviar uma mensagem quando a lavagem tiver terminado.

A computação ubíqua e a computação móvel se sobrepõem, pois, em princípio, o usuário móvel pode usar computadores que estejam em qualquer lugar. Porém, elas são distintas, de modo geral. A computação ubíqua pode ajudar os usuários enquanto estão em um único local, como em casa ou em um hospital. Do mesmo modo, a computação móvel tem vantagens, mesmo envolvendo apenas computadores e equipamentos convencionais isolados, como *notebooks* e impressoras.

A Figura 1.4 mostra um usuário, visitante em uma organização anfitriã, que pode acessar serviços locais a intranet dessa organização, a Internet ou a sua intranet doméstica.

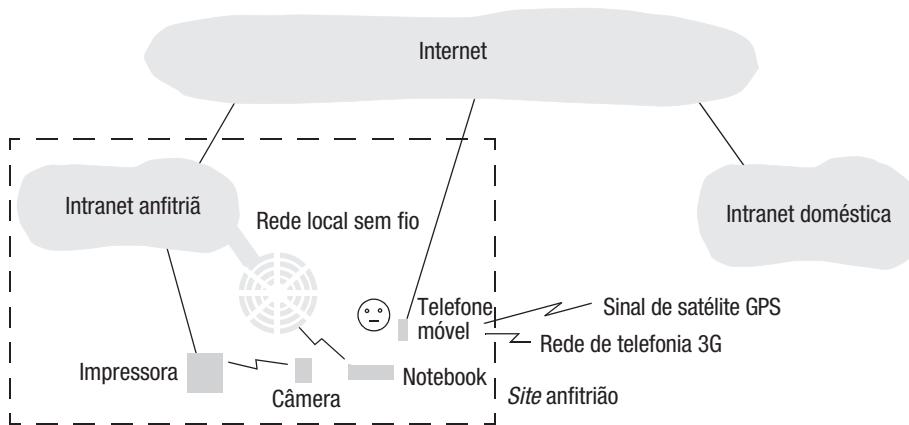


Figura 1.4 Equipamentos portáteis em um sistema distribuído.

O usuário tem acesso a três formas diferentes de conexão sem fio. Seu *notebook* tem uma maneira de se conectar com a rede local sem fio da organização anfitriã. Essa rede fornece cobertura de algumas centenas de metros (um andar de um edifício, digamos) e se conecta ao restante da intranet anfitriã por meio de um *gateway* ou ponto de acesso. O usuário também tem um telefone móvel (celular) que tem acesso à Internet, permitindo acesso à Web e a outros serviços de Internet, restrito apenas pelo que pode ser apresentado em sua pequena tela. O telefone também fornece informações locais por meio da funcionalidade de GPS incorporada. Por fim, o usuário está portando uma câmera digital, que se comunica por intermédio de uma rede sem fio pessoal (com alcance de cerca de 10 m) com outro equipamento, como, por exemplo, uma impressora.

Com uma infraestrutura conveniente, o usuário pode executar algumas tarefas no *site* anfitrião, usando os equipamentos que carrega consigo. Enquanto está no *site* anfitrião, o usuário pode buscar os preços de ações mais recentes a partir de um servidor Web, utilizando seu celular, e também pode usar o GPS incorporado e o *software* de localização de rota para obter instruções para a localização do *site*. Durante uma reunião com seus anfitriões, o usuário pode lhes mostrar uma fotografia, enviando-a a partir da câmera digital, diretamente para uma impressora ou projetor (local) adequadamente ativado na sala de reuniões (descoberto por meio de um serviço de localização). Isso exige apenas a comunicação sem fio entre a câmera e a impressora. Em princípio, eles podem enviar um documento de seus *notebooks* para a mesma impressora ou projetor, utilizando a rede local sem fio e as conexões Ethernet cabeadas com a impressora.

Esse cenário demonstra a necessidade de suportar *operação conjunta espontânea*, por meio da qual associações entre dispositivos são criadas e destruídas rotineiramente, por exemplo, localizando e utilizando os equipamentos da anfitriã (como as impressoras). O principal desafio que se aplica a tais situações é tornar a operação conjunta rápida e conveniente (isto é, espontânea), mesmo que o usuário esteja em um ambiente onde nunca esteve. Isso significa permitir que o equipamento do visitante se comunique com a rede da anfitriã e associá-lo a serviços locais convenientes – um processo chamado de *descoberta de serviço*.

A computação móvel e a computação ubíqua representam áreas de pesquisa ativa, e as várias dimensões anteriores serão discutidas em profundidade no Capítulo 19.

1.3.3 Sistemas multimídia distribuídos

Outra tendência importante é o requisito de suportar serviços multimídia em sistemas distribuídos. Uma definição útil de multimídia é a capacidade de suportar diversos tipos de mídia de maneira integrada. É de se esperar que um sistema distribuído suporte o armazenamento, a transmissão e a apresentação do que frequentemente são referidos como tipos de mídia distintos, como imagens ou mensagens de texto. Um sistema multimídia distribuído deve ser capaz de executar as mesmas funções para tipos de mídia contínuos, como áudio e vídeo, assim como armazenar e localizar arquivos de áudio ou vídeo, transmiti-los pela rede (possivelmente em tempo real, à medida que os fluxos saem de uma câmera de vídeo), suportar a apresentação dos tipos de mídia para o usuário e, opcionalmente, também compartilhar os tipos de mídia por um grupo de usuários.

A característica fundamental dos tipos de mídia contínuos é que eles incluem uma dimensão temporal e, de fato, a integridade do tipo de mídia depende fundamentalmente da preservação das relações em tempo real entre seus elementos. Por exemplo, em uma apresentação de vídeo, é necessário preservar determinado ritmo de transferência em termos de quadros por segundo e, para fluxos em tempo real, determinado atraso ou latência máxima para o envio dos quadros (esse é um exemplo da qualidade do serviço, discutida com mais detalhes na Seção 1.5.8, a seguir).

As vantagens da computação multimídia distribuída são consideráveis, pois uma ampla variedade de novos serviços (multimídia) e aplicativos pode ser fornecida na área de trabalho, incluindo o acesso a transmissões de televisão ao vivo ou gravadas, o acesso a bibliotecas de filmes que oferecem serviços de vídeo sob demanda, o acesso a bibliotecas de músicas, o fornecimento de recursos de áudio e teleconferência e os recursos de telefonia integrados, incluindo telefonia IP ou tecnologias relacionadas, como o Skype – uma alternativa *peer-to-peer* à telefonia IP (a infraestrutura de sistema distribuído que serve de base para o Skype está discutida na Seção 4.5.2). Note que essa tecnologia é revolucionária, desafiando os fabricantes a repensar muitos aparelhos. Por exemplo, qual é o equipamento de entretenimento doméstico básico do futuro – o computador, a televisão ou os consoles de games?

Webcasting é uma aplicação da tecnologia de multimídia distribuída. *Webcasting* é a capacidade de transmitir mídia contínua (normalmente áudio ou vídeo) pela Internet. Atualmente, é normal eventos esportivos ou musicais importantes serem transmitidos dessa maneira, frequentemente atraindo um grande número de espectadores (por exemplo, o concerto Live8 de 2005 atraiu cerca de 170.000 usuários simultâneos em seu momento de pico).

Sistemas multimídia distribuídos como o *Webcasting* impõem exigências consideráveis na infraestrutura distribuída subjacente, em termos de:

- Dar suporte a uma variedade (extensível) de formatos de codificação e criptografia, como a série MPEG de padrões (incluindo, por exemplo, o popular padrão MP3, conhecido como MPEG-1, Audio Layer 3) e HDTV.
- Fornecer diversos mecanismos para garantir que a qualidade de serviço desejada possa ser obtida.
- Fornecer estratégias de gerenciamento de recursos associadas, incluindo políticas de programação apropriadas para dar suporte à qualidade de serviço desejada.
- Fornecer estratégias de adaptação para lidar com a inevitável situação, em sistemas abertos, em que a qualidade do serviço não pode ser obtida ou mantida.

Mais discussões sobre tais mecanismos podem ser encontradas no Capítulo 20.

1.3.4 Computação distribuída como um serviço público

Com a crescente maturidade da infraestrutura dos sistemas distribuídos, diversas empresas estão promovendo o conceito dos recursos distribuídos como uma *commodity* ou um serviço público, fazendo a analogia entre recursos distribuídos e outros serviços públicos, como água ou eletricidade. Nesse modelo, os recursos são supridos por fornecedores de serviço apropriados e efetivamente alugados, em vez de pertencerem ao usuário final. Esse modelo se aplica tanto a recursos físicos como a serviços lógicos:

- Recursos físicos, como armazenamento e processamento, podem se tornar disponíveis para computadores ligados em rede, eliminando a necessidade de possuírem, eles próprios, esses recursos. Em uma extremidade do espectro, um usuário pode optar por um recurso de armazenamento remoto para requisitos de armazenamento de arquivos (por exemplo, para dados multimídia, como fotografias, música ou vídeo) e/ou para *backups*. Analogamente, essa estratégia permitiria a um usuário alugar um ou mais nós computacionais, para atender a suas necessidades de computação básicas ou para fazer computação distribuída. Na outra extremidade do espectro, os usuários podem acessar sofisticados *data centers* (recursos interligados em rede que dão acesso a repositórios de volumes de dados frequentemente grandes para usuários ou organizações) ou mesmo infraestrutura computacional, usando os serviços agora fornecidos por empresas como Amazon e Google. A virtualização do sistema operacional é uma tecnologia importante para essa estratégia, significando que os usuários podem ser atendidos por serviços em um nó virtual, em vez de físico, oferecendo maior flexibilidade ao fornecedor de serviço em termos de gerenciamento de recursos (a virtualização do sistema operacional é discutida com mais detalhes no Capítulo 7).
- Serviços de *software* (conforme definidos na Seção 1.4) também podem se tornar disponíveis pela Internet global por meio dessa estratégia. De fato, diversas empresas agora oferecem uma ampla variedade deles para aluguel, incluindo serviços como *e-mail* e calendários distribuídos. O Google, por exemplo, empacota diversos serviços empresariais sob o banner Google Apps [www.google.com II]. Esse avanço é possível graças a padrões combinados para serviços de *software*, como, por exemplo, os fornecidos pelos serviços Web (consulte o Capítulo 9).

O termo *computação em nuvem* é usado para capturar essa visão da computação como um serviço público. Uma nuvem é definida como um conjunto de serviços de aplicativo, armazenamento e computação baseados na Internet, suficientes para suportar as necessidades da maioria dos usuários, permitindo assim que eles prescindam, em grande medida ou totalmente, do *software* local de armazenamento de dados ou de aplicativo (consulte a Figura 1.5). O termo também promove a visão de tudo como um serviço de infraestrutura física ou virtual por meio de *software*, frequentemente pago com base na utilização, em vez de na aquisição. Note que a computação em nuvem reduz os requisitos dos equipamentos dos usuários, permitindo que aparelhos de mesa ou portáteis muito simples acessem uma variedade potencialmente ampla de recursos e serviços.

Geralmente, as nuvens são implementadas em *cluster* de computadores para fornecer a escala e o desempenho necessários exigidos por tais serviços. Um *cluster de computadores* é um conjunto de computadores interligados que cooperam estreitamente para fornecer um único recurso de computação integrado de alto desempenho. Ampliando projetos como o NOW (Network of Workstations) Project da Berkeley [Anderson *et al.* 1995, now.cs.berkeley.edu] e o Beowulf da NASA [www.beowulf.org], a tendência é no sentido de utilizar *hardware* de prateleira tanto para os computadores como para a interligação

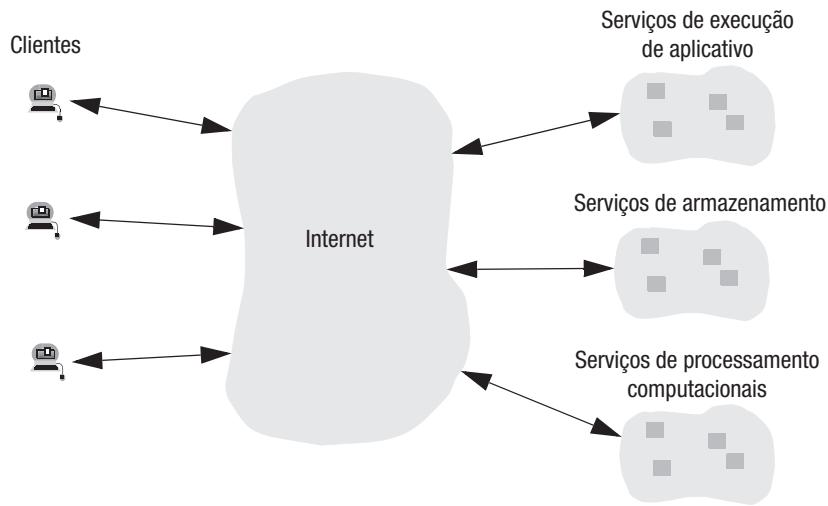


Figura 1.5 Computação em nuvem.

de redes. A maioria dos grupos consiste em PCs convencionais executando uma versão padrão (às vezes reduzida) de um sistema operacional, como o GNU/Linux, interligados por uma rede local. Empresas como HP, Sun e IBM oferecem soluções *blade*. Os servidores *blade* são elementos computacionais mínimos, contendo, por exemplo, recursos de processamento e armazenamento (memória principal). Um sistema *blade* consiste em um número potencialmente grande de servidores *blade* contidos em gabinetes e armários específicos, também denominados *rack*, mantidos em uma mesma instalação física. Outros elementos, como energia, refrigeração, armazenamento persistente (discos), ligação em rede e telas, são fornecidos pela instalação física ou por meio de soluções virtualizadas (discutidas no Capítulo 7). Com essa solução, os servidores *blade* individuais podem ser muito menores e também mais baratos de produzir do que os PCs convencionais.

O objetivo geral dos grupos de computadores é fornecer serviços na nuvem, incluindo recursos de computação de alto desempenho, mas também diversos outros serviços, englobando armazenamento de massa (por exemplo, por intermédio de centros de dados) ou serviços de aplicativo mais elaborados, como pesquisa na Web (o Google, por exemplo, conta com uma arquitetura de grupo de computadores volumosa para implementar seu mecanismo de busca e outros serviços, conforme discutido no Capítulo 21).

A *computação em grade* (discutida no Capítulo 9, Seção 9.7.2) também pode ser vista como uma forma de computação em nuvem. Em grande medida, os termos são sinônimos e, às vezes, mal definidos, mas a computação em grade geralmente pode ser vista como precursora do paradigma mais geral da computação em nuvem, com tendência para o suporte para aplicativos científicos.

1.4 Enfoque no compartilhamento de recursos

Os usuários estão tão acostumados às vantagens do compartilhamento de recursos que podem facilmente ignorar seu significado. Rotineiramente, compartilhamos recursos de *hardware* (como impressoras), recursos de dados (como arquivos) e recursos com funcionalidade mais específica (como os mecanismos de busca).

Do ponto de vista do *hardware*, compartilhamos equipamentos como impressoras e discos para reduzir os custos. Contudo, a maior importância para os usuários é o compartilhamento dos recursos em um nível de abstração mais alto, como informações necessárias às suas aplicações, ao seu trabalho diário e em suas atividades sociais. Por exemplo, os usuários se preocupam em compartilhar informações através de um banco de dados ou de um conjunto de páginas Web – e não com discos ou processadores em que eles estão armazenados. Analogamente, os usuários pensam em termos de recursos compartilhados, como um mecanismo de busca ou um conversor de moeda corrente, sem considerar o servidor (ou servidores) que os fornecem.

Na prática, os padrões de compartilhamento de recursos variam amplamente na abrangência e no quanto os usuários trabalham em conjunto. Em um extremo, temos um mecanismo de busca na Web que fornece um recurso para usuários de todo o mundo, usuários estes que nunca entram em contato diretamente. No outro extremo, no *trabalho cooperativo apoiado por computador*, um grupo de usuários colabora diretamente compartilhando recursos, como documentos, em um pequeno grupo fechado. O padrão de compartilhamento e a distribuição geográfica dos usuários determinam quais mecanismos o sistema deve fornecer para coordenar as ações dos usuários.

O termo *serviço* é usado para designar uma parte distinta de um sistema computacional que gerencia um conjunto de recursos relacionados e apresenta sua funcionalidade para usuários e aplicativos. Por exemplo, acessamos arquivos compartilhados por intermédio de um serviço de sistema de arquivos; enviamos documentos para impressoras por meio de um serviço de impressão; adquirimos bens por meio de um serviço de pagamento eletrônico. O único acesso que temos ao serviço é por intermédio do conjunto de operações que ele exporta. Por exemplo, um serviço de sistema de arquivos fornece operações de *leitura, escrita e exclusão* dos arquivos.

Em parte, o fato de os serviços restringirem o acesso ao recurso a um conjunto de operações bem definidas é uma prática padrão na engenharia de *software*, mas isso também reflete a organização física dos sistemas distribuídos. Em um sistema distribuído, os recursos são fisicamente encapsulados dentro dos computadores e só podem ser acessados a partir de outros computadores por intermédio de mecanismos de comunicação. Para se obter um compartilhamento eficiente, cada recurso deve ser gerenciado por um programa que ofereça uma interface de comunicação, permitindo ao recurso ser acessado e atualizado de forma confiável e consistente.

O termo *servidor* provavelmente é conhecido da maioria dos leitores. Ele se refere a um programa em execução (um *processo*) em um computador interligado em rede, que aceita pedidos de programas em execução em outros computadores para efetuar um serviço e responder apropriadamente. Os processos que realizam os pedidos são referidos como *clientes* e a estratégia geral é conhecida como *computação cliente-servidor*. Nessa estratégia, os pedidos são enviados em mensagens dos clientes para um servidor, e as respostas são enviadas do servidor para os clientes. Quando o cliente envia um pedido para que uma operação seja efetuada, dizemos que o cliente *requisita uma operação* no servidor. Uma interação completa entre um cliente e um servidor, desde quando o cliente envia seu pedido até o momento em que recebe a resposta do servidor, é chamada de *requisição remota*.

O mesmo processo pode ser tanto cliente como servidor, pois, às vezes, os servidores solicitam operações em outros servidores. Os termos cliente e servidor só se aplicam às funções desempenhadas em um único pedido. Os clientes são ativos (fazendo pedidos) e os servidores são passivos (sendo ativados somente ao receberem pedidos); os servidores funcionam continuamente, enquanto os clientes duram apenas enquanto os aplicativos dos quais fazem parte estiverem ativos.

Note que, por padrão, os termos *cliente* e *servidor* se referem a *processos* e não aos computadores em que são executados, embora no jargão comum esses termos também se refiram aos computadores em si. Outra distinção, que discutiremos no Capítulo 5, é que, em um sistema distribuído escrito em uma linguagem orientada a objetos, os recursos podem ser encapsulados como objetos e acessados por objetos clientes; nesse caso, falamos em um *objeto cliente* invocando um método em um *objeto servidor*.

Muitos sistemas distribuídos (mas certamente não todos) podem ser totalmente construídos na forma de clientes e servidores. A World Wide Web, o *e-mail* e as impressoras interligadas em rede são exemplos que se encaixam nesse modelo. Discutiremos alternativas para os sistemas cliente-servidor no Capítulo 2.

Um navegador Web em execução é um exemplo de cliente. O navegador Web se comunica com um servidor Web para solicitar páginas Web. Examinaremos a Web com mais detalhes na Seção 1.6.

1.5 Desafios

Os exemplos da Seção 1.2 ilustram a abrangência dos sistemas distribuídos e os problemas que surgem em seu projeto. Em muitos deles, desafios significativos foram encontrados e superados. Como a abrangência e a escala dos sistemas distribuídos e dos aplicativos são maiores, é provável que eles apresentem os mesmos desafios e ainda outros. Nesta seção, descrevemos os principais desafios dos sistemas distribuídos.

1.5.1 Heterogeneidade

A Internet permite aos usuários acessarem serviços e executarem aplicativos por meio de um conjunto heterogêneo de computadores e redes. A heterogeneidade (isto é, variedade e diferença) se aplica aos seguintes aspectos:

- redes;
- *hardware* de computador;
- sistemas operacionais;
- linguagens de programação;
- implementações de diferentes desenvolvedores.

Embora a Internet seja composta de muitos tipos de redes (como ilustrado na Figura 1.3), suas diferenças são mascaradas pelo fato de que todos os computadores ligados a elas utilizam protocolos Internet para se comunicar. Por exemplo, um computador que possui uma placa Ethernet tem uma implementação dos protocolos Internet, enquanto um computador em um tipo diferente de rede tem uma implementação dos protocolos Internet para essa rede. O Capítulo 3 explica como os protocolos Internet são implementados em uma variedade de redes diferentes.

Os tipos de dados, como os inteiros, podem ser representados de diversas maneiras em diferentes tipos de *hardware*; por exemplo, existem duas alternativas para a ordem em que os bytes de valores inteiros são armazenados: uma iniciando a partir do byte mais significativo e outra, a partir do byte menos significativo. Essas diferenças na representação devem ser consideradas, caso mensagens devam ser trocadas entre programas sendo executados em diferentes *hardwares*.

Embora os sistemas operacionais de todos os computadores na Internet precisem incluir uma implementação dos protocolos Internet, nem todos fornecem, necessariamente,

a mesma interface de programação de aplicativos para esses protocolos. Por exemplo, as chamadas para troca de mensagens no UNIX são diferentes das chamadas no Windows.

Diferentes linguagens de programação usam diferentes representações para caracteres e estruturas de dados, como vetores e registros. Essas diferenças devem ser consideradas, caso programas escritos em diferentes linguagens precisem se comunicar.

Os programas escritos por diferentes desenvolvedores não podem se comunicar, a menos que utilizem padrões comuns; por exemplo, para realizar a comunicação via rede e usar uma mesma representação de tipos de dados primitivos e estruturas de dados nas mensagens. Para que isso aconteça, padrões precisam ser estabelecidos e adotados. Assim é o caso dos protocolos Internet.

Middleware • O termo *middleware* se aplica a uma camada de *software* que fornece uma abstração de programação, assim como o mascaramento da heterogeneidade das redes, do *hardware*, dos sistemas operacionais e das linguagens de programação subjacentes. O CORBA (Common Object Request Broker), que será descrito nos Capítulos 4, 5 e 8, é um exemplo. Alguns *middlewares*, como o Java RMI (Remote Method Invocation) (veja o Capítulo 5), suportam apenas uma linguagem de programação. A maioria é implementada sobre os protocolos Internet, os quais escondem a diferença existente entre redes subjacentes. Todo *middleware*, em si, trata das diferenças em nível dos sistemas operacionais e do *hardware* – o modo como isso é feito será o tópico principal do Capítulo 4.

Além de resolver os problemas de heterogeneidade, o *middleware* fornece um modelo computacional uniforme para ser usado pelos programadores de serviços e de aplicativos distribuídos. Os modelos possíveis incluem a invocação remota de objetos, a notificação remota de eventos, o acesso remoto a banco de dados e o processamento de transação distribuído. Por exemplo, o CORBA fornece invocação remota de objetos, a qual permite que um objeto, em um programa sendo executado em um computador, invoque um método de um objeto em um programa executado em outro computador. Sua implementação oculta o fato de que as mensagens passam por uma rede para enviar o pedido de invocação e sua resposta.

Heterogeneidade e migração de código • O termo *migração de código*, ou ainda, *código móvel*, é usado para se referir ao código de programa que pode ser transferido de um computador para outro e ser executado no destino – os *applets* Java são um exemplo. Um código destinado à execução em um computador não é necessariamente adequado para outro computador, pois, normalmente, os programas executáveis são específicos a um conjunto de instruções e a um sistema operacional.

A estratégia de *máquina virtual* oferece uma maneira de tornar um código executável em uma variedade de computadores hospedeiros: o compilador de uma linguagem em particular gera código para uma máquina virtual, em vez de código para um processador e sistema operacional específicos. Por exemplo, o compilador Java produz código para uma máquina virtual Java (JVM, Java Virtual Machine), a qual o executa por meio de interpretação. A máquina virtual Java precisa ser implementada uma vez para cada tipo de computador a fim de que os programas Java sejam executados.

Atualmente, a forma mais usada de código móvel é a inclusão de programas *JavaScript* em algumas páginas Web carregadas nos navegadores clientes. Essa extensão da tecnologia da Web será discutida com mais detalhes na Seção 1.6, a seguir.

1.5.2 Sistemas abertos

Diz-se que um sistema computacional é aberto quando ele pode ser estendido e reimplementado de várias maneiras. O fato de um sistema distribuído ser ou não um sistema

aberto é determinado principalmente pelo grau com que novos serviços de compartilhamento de recursos podem ser adicionados e disponibilizados para uso por uma variedade de programas clientes.

A característica de sistema aberto é obtida a partir do momento em que a especificação e a documentação das principais interfaces de *software* dos componentes de um sistema estão disponíveis para os desenvolvedores de *software*. Em uma palavra, as principais interfaces são *publicadas*. Esse processo é similar àquele realizado por organizações de padronização, porém, frequentemente, ignora os procedimentos oficiais, os quais normalmente são pesados e lentos.

Entretanto, a publicação de interfaces é apenas o ponto de partida para adicionar e estender serviços em um sistema distribuído. O maior desafio para os projetistas é encarar a complexidade de sistemas distribuídos compostos por muitos componentes e elaborados por diferentes pessoas.

Os projetistas dos protocolos Internet elaboraram uma série de documentos, chamados *Requests For Comments*, ou RFCs, cada um identificado por um número. No início dos anos 80, as especificações dos protocolos de comunicação Internet foram publicadas nessa série, acompanhadas de especificações de aplicativos executados com eles, como transferência de arquivos, *e-mail* e telnet. Essa prática continua e forma a base da documentação técnica da Internet. Essa série inclui discussões, assim como as especificações dos protocolos (pode-se obter cópias das RFCs no endereço [www.ietf.org]). Dessa forma, com a publicação dos protocolos de comunicação Internet, permitiu-se a construção de uma variedade de sistemas e aplicativos para a Internet, incluindo a Web. As RFCs não são os únicos meios de publicação. Por exemplo, o W3C (World Wide Web Consortium) desenvolve e publica padrões relacionados ao funcionamento da Web (veja [www.w3.org]).

Os sistemas projetados a partir de padrões públicos são chamados de *sistemas distribuídos abertos*, para reforçar o fato de que eles são extensíveis. Eles podem ser ampliados em nível de *hardware*, pela adição de computadores em uma rede, e em nível de *software*, pela introdução de novos serviços ou pela reimplementação dos antigos, permitindo aos programas aplicativos compartilharem recursos. Uma vantagem adicional, frequentemente mencionada para sistemas abertos, é sua independência de fornecedores individuais.

Resumindo:

- Os sistemas abertos são caracterizados pelo fato de suas principais interfaces serem publicadas.
- Os sistemas distribuídos abertos são baseados na estipulação de um mecanismo de comunicação uniforme e em interfaces publicadas para acesso aos recursos compartilhados.
- Os sistemas distribuídos abertos podem ser construídos a partir de *hardware* e *software* heterogêneo, possivelmente de diferentes fornecedores. Para que um sistema funcione corretamente, a compatibilidade de cada componente com o padrão publicado deve ser cuidadosamente testada e verificada.

1.5.3 Segurança

Muitos recursos de informação que se tornam disponíveis e são mantidos em sistemas distribuídos têm um alto valor intrínseco para seus usuários. Portanto, sua segurança é de considerável importância. A segurança de recursos de informação tem três componentes: confidencialidade (proteção contra exposição para pessoas não autorizadas), integridade (proteção contra alteração ou dano) e disponibilidade (proteção contra interferência com os meios de acesso aos recursos).

A Seção 1.1 mostrou que, embora a Internet permita que um programa em um computador se comunique com um programa em outro computador, independentemente de sua localização, existem riscos de segurança associados ao livre acesso a todos os recursos em uma intranet. Embora um *firewall* possa ser usado para formar uma barreira em torno de uma intranet, restringindo o tráfego que pode entrar ou sair, isso não garante o uso apropriado dos recursos pelos usuários de dentro da intranet, nem o uso apropriado dos recursos na Internet, que não são protegidos por *firewalls*.

Em um sistema distribuído, os clientes enviam pedidos para acessar dados gerenciados por servidores, o que envolve o envio de informações em mensagens por uma rede. Por exemplo:

1. Um médico poderia solicitar acesso aos dados dos pacientes de um hospital ou enviar mais informações sobre esses pacientes.
2. No comércio eletrônico e nos serviços bancários, os usuários enviam seus números de cartão de crédito pela Internet.

Nesses dois exemplos, o desafio é enviar informações sigilosas em uma ou mais mensagens, por uma rede, de maneira segura. No entanto, a segurança não é apenas uma questão de ocultar o conteúdo das mensagens – ela também envolve saber com certeza a identidade do usuário, ou outro agente, em nome de quem uma mensagem foi enviada. No primeiro exemplo, o servidor precisa saber se o usuário é realmente um médico e, no segundo exemplo, o usuário precisa ter certeza da identidade da loja ou do banco com o qual está tratando. O segundo desafio, neste caso, é identificar corretamente um usuário ou outro agente remoto. Esses dois desafios podem ser resolvidos com o uso de técnicas de criptografia desenvolvidas para esse propósito. Elas são amplamente utilizadas na Internet e serão discutidas no Capítulo 11.

Entretanto, dois desafios de segurança, descritos a seguir, ainda não foram totalmente resolvidos:

Ataque de negação de serviço (Denial of Service): ocorre quando um usuário interrompe um serviço por algum motivo. Isso pode ser conseguido bombardeando o serviço com um número tão grande de pedidos sem sentido, que os usuários sérios não são capazes de utilizá-lo. Isso é chamado de *ataque de negação de serviço*. De tempos em tempos, surgem ataques de negação de serviços contra alguns serviços e servidores Web bastante conhecidos. Atualmente, tais ataques são controlados buscando-se capturar e punir os responsáveis após o evento, mas essa não é uma solução geral para o problema. Medidas para se opor a isso, baseadas em melhorias no gerenciamento das redes, estão sendo desenvolvidas e serão assunto do Capítulo 3.

Segurança de código móvel: um código móvel precisa ser manipulado com cuidado. Considere alguém que receba um programa executável como um anexo de correio eletrônico: os possíveis efeitos da execução do programa são imprevisíveis; por exemplo, pode parecer que ele está apenas exibindo uma figura interessante, mas, na realidade, está acessando recursos locais ou, talvez, fazendo parte de um ataque de negação de serviço. Algumas medidas para tornar seguro um código móvel serão esboçadas no Capítulo 11.

1.5.4 Escalabilidade

Os sistemas distribuídos funcionam de forma efetiva e eficaz em muitas escalas diferentes, variando desde uma pequena intranet até a Internet. Um sistema é descrito como *escalável* se permanece eficiente quando há um aumento significativo no número de re-

cursos e no número de usuários. O número de computadores e de serviços na Internet aumentou substancialmente. A Figura 1.6 mostra o aumento no número de computadores e servidores Web durante os 12 anos de história da Web até 2005 (veja [[zakon.org](#)]). É interessante notar o aumento significativo tanto no número de computadores como no serviços Web nesse período, mas também a porcentagem relativa, que cresce rapidamente, uma tendência explicada pelo crescimento da computação pessoal fixa e móvel. Um único servidor Web também pode, cada vez mais, ser hospedado em vários computadores.

O projeto de sistemas distribuídos escaláveis apresenta os seguintes desafios:

Controlar o custo dos recursos físicos: à medida que a demanda por um recurso aumenta, deve ser possível, a um custo razoável, ampliar o sistema para atendê-la. Por exemplo, a frequência com que os arquivos são acessados em uma intranet provavelmente vai crescer à medida que o número de usuários e de computadores aumentar. Deve ser possível adicionar servidores de arquivos de forma a evitar o gargalo de desempenho que haveria caso um único servidor de arquivos tivesse de tratar todos os pedidos de acesso a arquivos. Em geral, para que um sistema com n usuários seja escalável, a quantidade de recursos físicos exigida para suportá-los deve ser, no máximo, $O(n)$ – isto é, proporcional a n . Por exemplo, se um único servidor de arquivos pode suportar 20 usuários, então dois servidores deverão suportar 40 usuários. Embora isso pareça um objetivo óbvio, não é necessariamente fácil atingi-lo, como mostraremos no Capítulo 12.

Controlar a perda de desempenho: considere o gerenciamento de um conjunto de dados, cujo tamanho é proporcional ao número de usuários ou recursos presentes no sistema; por exemplo, a tabela de correspondência entre os nomes de domínio dos computadores e seus endereços IP mantidos pelo Domain Name System (DNS), que é usado principalmente para pesquisar nomes, como www.amazon.com. Os algoritmos que utilizam estruturas hierárquicas têm melhor escalabilidade do que os que usam estruturas lineares. Porém, mesmo com as estruturas hierárquicas, um aumento no tamanho resulta em alguma perda de desempenho: o tempo que leva para acessar dados hierarquicamente estruturados é $O(\log n)$, onde n é o tamanho do conjunto de dados. Para que um sistema seja escalável, a perda de desempenho máxima não deve ser maior do que isso.

Impedir que os recursos de software se esgotem: um exemplo de falta de escalabilidade é mostrado pelos números usados como endereços IP (endereços de computador na Internet). No final dos anos 70, decidiu-se usar 32 bits para esse propósito,

Data	Computadores	Servidores Web	Percentual
Julho de 1993	1.776.000	130	0,008
Julho de 1995	6.642.000	23.500	0,4
Julho de 1997	19.540.000	1.203.096	6
Julho de 1999	56.218.000	6.598.697	12
Julho de 2001	125.888.197	31.299.592	25
Julho de 2003	~200.000.000	42.298.371	21
Julho de 2005	353.284.187	67.571.581	19

Figura 1.6 Crescimento da Internet (computadores e servidores Web).

mas, conforme será explicado no Capítulo 3, a quantidade de endereços IP disponíveis está se esgotando. Por isso, uma nova versão do protocolo, com endereços IP em 128 bits, está sendo adotada e isso exige modificações em muitos componentes de *software*. Para sermos justos com os primeiros projetistas da Internet, não há uma solução correta para esse problema. É difícil prever, com anos de antecedência, a demanda que será imposta sobre um sistema. Além disso, superestimar o crescimento futuro pode ser pior do que se adaptar para uma mudança quando fôrmos obrigados a isso – por exemplo, endereços IP maiores ocupam espaço extra no armazenamento de mensagens e no computador.

Evitar gargalos de desempenho: em geral, os algoritmos devem ser descentralizados para evitar a existência de gargalos de desempenho. Ilustramos esse ponto com referência ao predecessor do Domain Name System, no qual a tabela de correspondência entre endereços IP e nomes era mantida em um único arquivo central, cujo *download* podia ser feito em qualquer computador que precisasse dela. Isso funcionava bem quando havia apenas algumas centenas de computadores na Internet, mas logo se tornou um sério gargalo de desempenho e de administração. Com milhares de computadores na Internet, imagine o tamanho e o acesso a esse arquivo central. O Domain Name System eliminou esse gargalo, particionando a tabela de correspondência de nomes entre diversos servidores localizados em toda a Internet e administrados de forma local – veja os Capítulos 3 e 13. Alguns recursos compartilhados são acessados com muita frequência; por exemplo, muitos usuários podem acessar uma mesma página Web, causando uma queda no desempenho. No Capítulo 2, veremos que o uso de cache e de replicação melhora o desempenho de recursos que são pesadamente utilizados.

De preferência, o *software* de sistema e de aplicativo não deve mudar quando a escala do sistema aumentar, mas isso é difícil de conseguir. O problema da escala é um tema central no desenvolvimento de sistemas distribuídos. As técnicas que têm obtido sucesso serão discutidas extensivamente neste livro. Elas incluem o uso de dados replicados (Capítulo 18), a técnica associada de uso de cache (Capítulos 2 e 12) e a distribuição de vários servidores para manipular as tarefas comumente executadas, permitindo que várias tarefas semelhantes sejam executadas concorrentemente.

1.5.5 Tratamento de falhas

Às vezes, os sistemas de computador falham. Quando ocorrem falhas no *hardware* ou no *software*, os programas podem produzir resultados incorretos ou podem parar antes de terem concluído a computação pretendida. No Capítulo 2, discutiremos e classificaremos uma variedade de tipos de falhas possíveis, que podem ocorrer nos processos e nas redes que compõem um sistema distribuído.

As falhas em um sistema distribuído são parciais – isto é, alguns componentes falham, enquanto outros continuam funcionando. Portanto, o tratamento de falhas é particularmente difícil. As técnicas para tratamento de falhas a seguir serão discutidas ao longo de todo o livro:

Detecção de falhas: algumas falhas podem ser detectadas. Por exemplo, somas de verificação podem ser usadas para detectar dados corrompidos em uma mensagem ou em um arquivo. O Capítulo 2 mostra que é difícil, ou mesmo impossível, detectar algumas outras falhas, como um servidor remoto danificado na Internet. O desafio é gerenciar a ocorrência de falhas que não podem ser detectadas, mas que podem ser suspeitas.

Mascaramento de falhas: algumas falhas detectadas podem ser ocultas ou se tornar menos sérias. Dois exemplos de ocultação de falhas:

1. Mensagens podem ser retransmitidas quando não chegam.
2. Dados de arquivos podem ser gravados em dois discos, para que, se um estiver danificado, o outro ainda possa estar correto.

Simplesmente eliminar uma mensagem corrompida é um exemplo de como tornar uma falha menos grave – ela pode ser retransmitida. O leitor provavelmente perceberá que as técnicas descritas para o mascaramento de falhas podem não funcionar nos piores casos; por exemplo, os dados no segundo disco também podem estar danificados ou a mensagem pode não chegar em um tempo razoável e, contudo, ser retransmitida frequentemente.

Tolerância a falhas: a maioria dos serviços na Internet apresenta falhas – não seria prático para eles tentar detectar e mascarar tudo que possa ocorrer em uma rede grande assim, com tantos componentes. Seus clientes podem ser projetados de forma a tolerar falhas, o que geralmente envolve a tolerância também por parte dos usuários. Por exemplo, quando um navegador não consegue contatar um servidor Web, ele não faz o usuário esperar indefinidamente, enquanto continua tentando – ele informa o usuário sobre o problema, deixando-o livre para tentar novamente. Os serviços que toleram falhas serão discutidos no item sobre redundância, logo a seguir.

Recuperação de falhas: a recuperação envolve projetar *software* de modo que o estado dos dados permanentes possa ser recuperado ou “retrocedido” após a falha de um servidor. Em geral, as computações realizadas por alguns programas ficarão incompletas quando ocorrer uma falha, e os dados permanentes que eles atualizam (arquivos em disco) podem não estar em um estado consistente. A recuperação de falhas será estudada no Capítulo 17.

Redundância: os serviços podem se tornar tolerantes a falhas com o uso de componentes redundantes. Considere os exemplos a seguir:

1. Sempre deve haver pelo menos duas rotas diferentes entre dois roteadores quaisquer na Internet.
2. No Domain Name System, toda tabela de correspondência de nomes é replicada em pelo menos dois servidores diferentes.
3. Um banco de dados pode ser replicado em vários servidores, para garantir que os dados permaneçam acessíveis após a falha de qualquer servidor. Os servidores podem ser projetados de forma a detectar falhas em seus pares; quando uma falha é detectada em um servidor, os clientes são redirecionados para os servidores restantes.

O projeto de técnicas eficazes para manter réplicas atualizadas de dados que mudam rapidamente, sem perda de desempenho excessiva, é um desafio. Várias estratégias para isso serão discutidas no Capítulo 18.

Os sistemas distribuídos fornecem um alto grau de disponibilidade perante falhas de *hardware*. A *disponibilidade* de um sistema é a medida da proporção de tempo em que ele está pronto para uso. Quando um dos componentes de um sistema distribuído falha, apenas o trabalho que estava usando o componente defeituoso é afetado. Um usuário pode passar para outro computador, caso aquele que estava sendo utilizado falhe; um processo servidor pode ser iniciado em outro computador.

1.5.6 Concorrência

Tanto os serviços como os aplicativos fornecem recursos que podem ser compartilhados pelos clientes em um sistema distribuído. Portanto, existe a possibilidade de que vários clientes tentem acessar um recurso compartilhado ao mesmo tempo. Por exemplo, uma estrutura de dados que registra lances de um leilão pode ser acessada com muita frequência, quando o prazo final se aproximar.

O processo que gerencia um recurso compartilhado poderia aceitar e tratar um pedido de cliente por vez. Contudo, essa estratégia limita o desempenho do tratamento de pedidos. Portanto, os serviços e aplicativos geralmente permitem que vários pedidos de cliente sejam processados concorrentemente. Para tornar isso mais concreto, suponha que cada recurso seja encapsulado como um objeto e que as chamadas sejam executadas em diferentes fluxos de execução, processos ou *threads*, concorrentes. Nesta situação, é possível que vários fluxos de execução estejam simultaneamente dentro de um objeto e, eventualmente, suas operações podem entrar em conflito e produzir resultados inconsistentes. Por exemplo, se dois lances em um leilão forem Smith: \$122 e Jones: \$111 e as operações correspondentes fossem entrelaçadas sem nenhum controle, eles poderiam ser armazenados como Smith: \$111 e Jones: \$122.

A moral da história é que qualquer objeto que represente um recurso compartilhado em um sistema distribuído deve ser responsável por garantir que ele opere corretamente em um ambiente concorrente. Isso se aplica não apenas aos servidores, mas também aos objetos nos aplicativos. Portanto, qualquer programador que implemente um objeto que não foi destinado para uso em um sistema distribuído deve fazer o que for necessário para garantir que, em um ambiente concorrente, ele não assuma resultados inconsistentes.

Para que um objeto mantenha coerência em um ambiente concorrente, suas operações devem ser sincronizadas de tal maneira que seus dados permaneçam consistentes. Isso pode ser obtido por meio de técnicas padrão, como semáforos, que são disponíveis na maioria dos sistemas operacionais. Esse assunto, e sua extensão para coleções de objetos compartilhados distribuídos, serão discutidos nos Capítulos 7 e 17.

1.5.7 Transparência

A transparência é definida como a ocultação, para um usuário final ou para um programador de aplicativos, da separação dos componentes em um sistema distribuído, de modo que o sistema seja percebido como um todo, em vez de como uma coleção de componentes independentes. As implicações da transparência têm grande influência sobre o projeto do *software* de sistema.

O ANSA *Reference Manual* [ANSA 1989] e o RM-ODP (Reference Model for Open Distributed Processing) da *International Organization for Standardization* [ISO 1992] identificam oito formas de transparência. Parafraseamos as definições originais da ANSA, substituindo transparência de migração por transparência de mobilidade, cujo escopo é mais abrangente:

Transparência de acesso permite que recursos locais e remotos sejam acessados com o uso de operações idênticas.

Transparência de localização permite que os recursos sejam acessados sem conhecimento de sua localização física ou em rede (por exemplo, que prédio ou endereço IP).

Transparência de concorrência permite que vários processos operem concorrentemente, usando recursos compartilhados sem interferência entre eles.

Transparência de replicação permite que várias instâncias dos recursos sejam usadas para aumentar a confiabilidade e o desempenho, sem conhecimento das réplicas por parte dos usuários ou dos programadores de aplicativos.

Transparência de falhas permite a ocultação de falhas, possibilitando que usuários e programas aplicativos concluam suas tarefas, a despeito da falha de componentes de hardware ou software.

Transparência de mobilidade permite a movimentação de recursos e clientes dentro de um sistema, sem afetar a operação de usuários ou de programas.

Transparência de desempenho permite que o sistema seja reconfigurado para melhorar o desempenho à medida que as cargas variam.

Transparência de escalabilidade permite que o sistema e os aplicativos se expandam em escala, sem alterar a estrutura do sistema ou os algoritmos de aplicação.

As duas transparências mais importantes são a de acesso e a de localização; sua presença ou ausência afeta mais fortemente a utilização de recursos distribuídos. Às vezes, elas são referidas em conjunto como *transparência de rede*.

Como exemplo da transparência de acesso, considere o uso de uma interface gráfica em um sistema de arquivo que organiza diretórios e subdiretórios em pastas; o que o usuário enxerga é a mesma coisa, independentemente de os arquivos serem contidos em uma pasta local ou remota. Outro exemplo é uma interface de programação para arquivos que usa as mesmas operações para acessar tanto arquivos locais como remotos (veja o Capítulo 12). Como exemplo de falta de transparência de acesso, considere um sistema distribuído que não permite acessar arquivos em um computador remoto, a não ser que você utilize o programa FTP para isso.

Os nomes de recurso na Web, isto é, os URLs, são transparentes à localização, pois a parte do URL que identifica o nome de um servidor Web se refere a um nome de computador em um domínio, em vez de seu endereço IP. Entretanto, os URLs não são transparentes à mobilidade, pois se a página Web de alguém mudar para o seu novo local de trabalho, em um domínio diferente, todos as referências (*links*) nas outras páginas ainda apontarão para a página original.

Em geral, identificadores como os URLs, que incluem os nomes de domínio dos computadores, impedem a transparência de replicação. Embora o DNS permita que um nome de domínio se refira a vários computadores, ele escolhe apenas um deles ao pesquisar um nome. Como um esquema de replicação geralmente precisa acessar todos os computadores participantes, ele precisará acessar cada uma das entradas de DNS pelo nome.

Para ilustrar a transparência de rede, considere o uso de um endereço de correio eletrônico, como *Fred.Flintstone@stoneit.com*. O endereço consiste em um nome de usuário e um nome de domínio. O envio de correspondência para tal usuário não envolve o conhecimento de sua localização física ou na rede, nem o procedimento de envio de uma mensagem de correio depende da localização do destinatário. Assim, o correio eletrônico, dentro da Internet, oferece tanto transparência de localização como de acesso (isto é, transparência de rede).

A transparência de falhas também pode ser vista no contexto do correio eletrônico, que finalmente é entregue, mesmo quando os servidores ou os enlaces de comunicação falham. As falhas são mascaradas pela tentativa de retransmitir as mensagens, até que elas sejam enviadas com êxito, mesmo que isso demore vários dias. Geralmente, o *middleware* converte as falhas de redes e os processos em exceções em nível de programação (veja uma explicação no Capítulo 5).

Para ilustrar a transparência de mobilidade, considere o caso dos telefones móveis. Suponha que quem faz a ligação e quem recebe a ligação estejam viajando de trem em diferentes partes de um país. Consideramos o telefone de quem chama como cliente e o telefone de quem recebe, como recurso. Os dois usuários de telefone que compõem a ligação não estão cientes da mobilidade dos telefones (o cliente e o recurso).

O uso dos diferentes tipos de transparência oculta e transforma em anônimos os recursos que não têm relevância direta para a execução de uma tarefa por parte de usuários e de programadores de aplicativos. Por exemplo, geralmente é desejável que recursos de *hardware* semelhantes sejam alocados de maneira permutável para executar uma tarefa – a identidade do processador usado para executar um processo geralmente fica oculta do usuário e permanece anônima. Conforme mencionado na Seção 1.3.2, nem sempre isso pode ser atingido. Por exemplo, um viajante que liga um *notebook* na rede local de cada escritório visitado faz uso de serviços locais, como o envio de correio eletrônico, usando diferentes servidores em cada local. Mesmo dentro de um prédio, é normal enviar um documento para que ele seja impresso em uma impressora específica configurada no sistema, normalmente a que está mais próxima do usuário.

1.5.8 Qualidade de serviço

Uma vez fornecida a funcionalidade exigida de um serviço (como o serviço de arquivo em um sistema distribuído), podemos passar a questionar a qualidade do serviço fornecido. As principais propriedades não funcionais dos sistemas que afetam a qualidade do serviço experimentada pelos clientes e usuários são a *confiabilidade*, a *segurança* e o *desempenho*. A *adaptabilidade* para satisfazer as configurações de sistema variáveis e a disponibilidade de recursos tem sido reconhecida como um aspecto muito importante da qualidade do serviço.

Os problemas de confiabilidade e de segurança são fundamentais no projeto da maioria dos sistemas de computador. O aspecto do desempenho da qualidade de serviço foi definido originalmente em termos da velocidade de resposta e do rendimento computacional, mas foi redefinido em termos da capacidade de satisfazer garantias de pontualidade, conforme discutido nos parágrafos a seguir.

Alguns aplicativos, incluindo os multimídia, manipulam *dados críticos quanto ao tempo* – fluxos de dados que precisam ser processados ou transferidos de um processo para outro em velocidade constante. Por exemplo, um serviço de filmes poderia consistir em um programa cliente que estivesse recuperando um filme de um servidor de vídeo e o apresentando na tela do usuário. Para se obter um resultado satisfatório, os sucessivos quadros de vídeo precisam ser exibidos para o usuário dentro de alguns limites de tempo especificados.

Na verdade, a abreviação QoS (Quality of Service) foi imprópria para se referir à capacidade dos sistemas de satisfazer tais prazos finais. Seu sucesso depende da disponibilidade dos recursos de computação e de rede necessários nos momentos apropriados. Isso implica um requisito para o sistema de fornecer recursos de computação e comunicação garantidos, suficientes para permitir que os aplicativos terminem cada tarefa a tempo (por exemplo, a tarefa de exibir um quadro de vídeo).

As redes normalmente usadas hoje têm alto desempenho; por exemplo, o iPlayer, da BBC, geralmente tem desempenho aceitável, mas quando as redes estão muito carregadas, seu desempenho pode se deteriorar – e não é dada nenhuma garantia. A qualidade de serviço (QoS) se aplica tanto aos sistemas operacionais quanto às redes. Cada recurso fundamental deve ser reservado pelos aplicativos que exigem QoS e deve existir gerenciadores de recursos que deem garantias. Os pedidos de reserva que não podem ser atendidos são rejeitados. Esses problemas serão tratados com mais profundidade no Capítulo 20.

1.6 Estudo de caso: a World Wide Web

A World Wide Web [www.w3.org I, Berners-Lee 1991] é um sistema em evolução para a publicação e para o acesso a recursos e serviços pela Internet. Por meio de navegadores Web (*browsers*) comumente disponíveis, os usuários recuperam e veem documentos de muitos tipos, ouvem fluxos de áudio, assistem a fluxos de vídeo e interagem com um vasto conjunto de serviços.

A Web nasceu no centro europeu de pesquisa nuclear (CERN), na Suíça, em 1989, como um veículo para troca de documentos entre uma comunidade de físicos conectados pela Internet [Berners-Lee 1999]. Uma característica importante da Web é que ela fornece uma estrutura de *hipertexto* entre os documentos que armazena, refletindo a necessidade dos usuários de organizar seus conhecimentos. Isso significa que os documentos contêm *links* (ou *hyperlinks*) – referências para outros documentos e recursos que também estão armazenados na Web.

É fundamental para um usuário da Web que, ao encontrar determinada imagem ou texto dentro de um documento, isso seja frequentemente acompanhado de *links* para documentos e outros recursos relacionados. A estrutura de *links* pode ser arbitrariamente complexa, e o conjunto de recursos que podem ser adicionados é ilimitado – a “teia” (Web) de *links* é realmente mundial. Bush [1945] imaginou estruturas de hipertexto há mais de 50 anos; foi com o desenvolvimento da Internet que essa ideia pôde se manifestar em escala mundial.

A Web é um sistema *aberto*: ela pode ser ampliada e implementada de novas maneiras, sem perturbar a funcionalidade já existente (consulte a Seção 1.5.2). A operação da Web é baseada em padrões de comunicação e de documentos livremente publicados. Por exemplo, existem muitos tipos de navegadores, em muitos casos, implementados em várias plataformas; e existem muitas implementações de servidores Web. Qualquer navegador pode recuperar recursos de qualquer servidor, desde que ambos utilizem os mesmos padrões em suas implementações. Os usuários têm acesso a navegadores na maioria dos equipamentos que utilizam, desde telefones móveis até computadores de mesa.

A Web é aberta no que diz respeito aos tipos de recursos que nela podem ser publicados e compartilhados. Em sua forma mais simples, um recurso da Web é uma página ou algum outro tipo de *conteúdo* que possa ser armazenado em um arquivo e apresentado ao usuário, como arquivos de programa, arquivos de mídia e documentos em PostScript ou Portable Document Format (pdf). Se alguém inventar, digamos, um novo formato de armazenamento de imagem, então as imagens que tenham esse formato poderão ser publicadas na Web imediatamente. Os usuários necessitam de meios para ver imagens nesse novo formato, mas os navegadores são projetados de maneira a acomodar nova funcionalidade de apresentação de conteúdo, na forma de aplicativos “auxiliares” e “plugins”.

A Web já foi além desses recursos de dados simples e hoje abrange serviços como a aquisição eletrônica de bens. Ela tem evoluído sem mudar sua arquitetura básica e é baseada em três componentes tecnológicos padrão principais:

- HTML (HyperText Markup Language), que é uma linguagem para especificar o conteúdo e o layout de páginas de forma que elas possam ser exibidas pelos navegadores Web.
- URLs (Uniform Resource Locators), que identificam os documentos e outros recursos armazenados como parte da Web.
- Uma arquitetura de sistema cliente-servidor, com regras padrão para interação (o protocolo HTTP – HyperText Transfer Protocol) por meio das quais os navegadores e outros clientes buscam documentos e outros recursos dos servidores Web. A

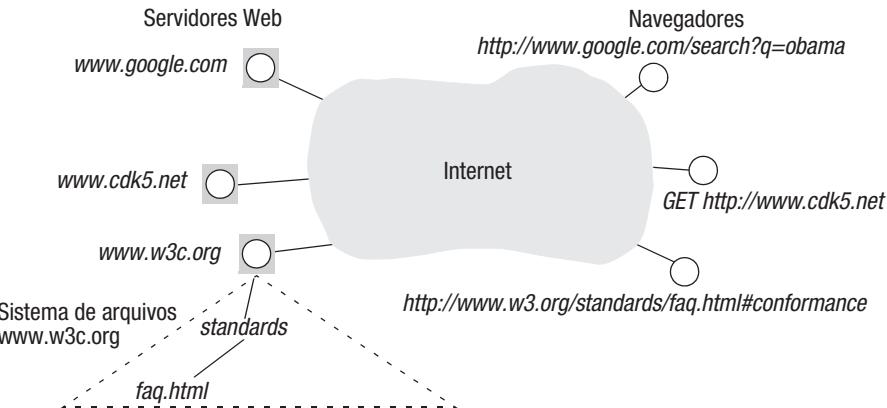


Figura 1.7 Servidores e navegadores Web.

Figura 1.7 mostra alguns navegadores realizando pedidos para servidores Web. É uma característica importante o fato de os usuários poderem localizar e gerenciar seus próprios servidores Web em qualquer parte da Internet.

Discussiremos agora cada um desses componentes e, ao fazermos isso, explicaremos o funcionamento dos navegadores e servidores Web quando um usuário busca páginas Web e clica nos *links* existentes.

HTML • A HyperText Markup Language [[www.w3.org II](http://www.w3.org/HTML)] é usada para especificar o texto e as imagens que compõem o conteúdo de uma página Web e para especificar como eles são dispostos e formatados para apresentação ao usuário. Uma página Web contém itens estruturados como cabeçalhos, parágrafos, tabelas e imagens. A HTML também é usada para especificar *links* e os recursos associados a eles.

Os usuários produzem código HTML manualmente, usando um editor de textos padrão ou um editor *wysiwyg* (*what you see is what you get*) com reconhecimento de HTML, que gera código HTML a partir de um layout criado graficamente. Um texto em HTML típico aparece a seguir:

<pre> <P> Welcome to Earth! Visitors may also be interested in taking a look at the Moon. </P></pre>	1 2 3 4 5
--	-----------------------

Esse texto em HTML é armazenado em um arquivo que um servidor Web pode acessar – digamos que seja o arquivo *earth.html*. Um navegador recupera o conteúdo desse arquivo a partir de um servidor Web – neste caso específico, um servidor em um computador chamado www.cdk5.net. O navegador lê o conteúdo retornado pelo servidor e o apresenta como um texto formatado com as imagens que o compõe, disposto em uma página Web na forma que conhecemos. Apenas o navegador – não o servidor – interpreta o texto em HTML. Contudo, o servidor informa ao navegador sobre o tipo de conteúdo que está retornando, para distingui-lo de, digamos, um documento em Portable Document Format. O servidor pode deduzir o tipo de conteúdo a partir da extensão do nome de arquivo ‘.html’.

Note que as diretivas HTML, conhecidas como *tags*, são incluídas entre sinais de menor e maior, como em `<P>`. A linha 1 do exemplo identifica um arquivo contendo uma imagem de apresentação. Seu URL é `http://www.cdk5.net/WebExample/Images/earth.jpg`. As linhas 2 e 5 possuem as diretivas para iniciar e terminar um parágrafo, respectivamente. As linhas 3 e 4 contêm o texto a ser exibido na página Web, no formato padrão de parágrafo.

A linha 4 especifica um *link* na página Web. Ele contém a palavra *Moon*, circundada por duas tags HTML relacionadas `<A HREF...>` e ``. O texto entre essas tags é o que aparece no *link* quando ele é apresentado na página Web. Por padrão, a maioria dos navegadores é configurada de modo a mostrar o texto de *links* sublinhado; portanto, o que o usuário verá nesse parágrafo será:

Welcome to Earth! Visitors may also be interested in taking a look at the Moon.

O navegador grava a associação entre o texto exibido do *link* e o URL contido na tag `<A HREF...>` – neste caso:

`http://www.cdk5.net/WebExample/moon.html`

Quando o usuário clica no texto, o navegador recupera o recurso identificado pelo URL correspondente e o apresenta para o usuário. No exemplo, o recurso está em um arquivo HTML que especifica uma página Web sobre a Lua.

URLs • O objetivo de um URL (Uniform Resource Locator) [www.w3.org III] é identificar um recurso. Na verdade, o termo usado em documentos que descrevem a arquitetura da Web é URI (Uniform Resource Identifier), mas, neste livro, o termo mais conhecido, URL, será usado quando não houver uma possível ambiguidade. Os navegadores examinam os URLs para acessar os recursos correspondentes. Às vezes, o usuário digita um URL no navegador. Mais comumente, o navegador pesquisa o URL correspondente quando o usuário clica em um *link*, quando seleciona um URL de sua lista de *bookmarks* ou quando o navegador busca um recurso incorporado em uma página Web, como uma imagem.

Todo URL, em sua forma completa e absoluta, tem dois componentes de nível superior:

esquema: identificador-específico-do-esquema

O primeiro componente, o “esquema”, declara qual é o tipo desse URL. Os URLs são obrigados a identificar uma variedade de recursos. Por exemplo, `mailto:joe@anISP.net` identifica o endereço de *e-mail* de um usuário; `ftp://ftp.downloadIt.com/software/aProg.exe` identifica um arquivo que deve ser recuperado com o protocolo FTP (File Transfer Protocol), em vez do protocolo mais comumente usado, HTTP. Outros exemplos de esquemas são “tel” (usado para especificar um número de telefone a ser discado, o que é particularmente útil ao se navegar em um telefone celular) e “tag” (usado para identificar uma entidade arbitrária).

A Web não tem restrições com relação aos tipos de recursos que pode usar para acesso, graças aos designadores de esquema presentes nos URLs. Se alguém inventar um novo tipo de recurso, por exemplo, *widget* – talvez com seu próprio esquema de endereçamento para localizar elementos em uma janela gráfica e seu próprio protocolo para acessá-los –, então o mundo poderá começar a usar URLs da forma `widget:...` É claro que os navegadores devem ter a capacidade de usar o novo protocolo *widget*, mas isso pode ser feito pela adição de um *plugin*.

Os URLs HTTP são os mais comuns para acessar recursos usando o protocolo HTTP padrão. Um URL HTTP tem duas tarefas principais a executar: identificar qual servidor Web mantém o recurso e qual dos recursos está sendo solicitado a esse servidor. A Figura 1.7 mostra três navegadores fazendo pedidos de recursos, gerenciados por três servidores

Web. O navegador que está mais acima está fazendo uma consulta em um mecanismo de busca. O navegador do meio solicita a página padrão de outro *site*. O navegador que está mais abaixo solicita uma página Web especificada por completo, incluindo um nome de caminho relativo para o servidor. Os arquivos de determinado servidor Web são mantidos em uma ou mais subárvore (diretórios) do sistema de arquivos desse servidor, e cada recurso é identificado por um nome e um caminho relativo (*pathname*) para esse servidor.

Em geral, os URLs HTTP são da seguinte forma:

http://nomedoservidor [:porta] [/nomedeCaminho] [?consulta][#fragmento]

onde os itens entre colchetes são opcionais. Um URL HTTP completo sempre começa com o *string* *http://*, seguido de um nome de servidor, expresso como um nome DNS (Domain Name System) (consulte a Seção 13.2). Opcionalmente, o nome DNS do servidor é seguido do número da porta em que o servidor recebe os pedidos (consulte o Capítulo 4) – que, por padrão, é a porta 80. Em seguida, aparece um nome de caminho opcional do recurso no servidor. Se ele estiver ausente, então a página Web padrão do servidor será solicitada. Por fim, o URL termina, também opcionalmente, com um componente de consulta – por exemplo, quando um usuário envia entradas de um formulário, como a página de consulta de um mecanismo de busca – e/ou um identificador de fragmento, que identifica um componente de um determinado recurso.

Considere os URLs a seguir:

http://www.cdk5.net
http://www.w3.org/standards/faq.html#conformance
http://www.google.com/search?q=obama

Eles podem ser decompostos, como segue:

<i>Server DNS name</i>	<i>Path name</i>	<i>Query</i>	<i>Fragment</i>
<i>www.cdk5.net</i>	(default)	(none)	(none)
<i>www.w3.org</i>	<i>standards/faq.html</i>	(none)	intro
<i>www.google.com</i>	<i>search</i>	<i>q=obama</i>	(none)

O primeiro URL designa a página padrão fornecida por *www.cdk5.net*. O segundo identifica um fragmento de um arquivo HTML cujo nome de caminho é *standards/faq.html*, relativo ao servidor *www.w3.org*. O identificador do fragmento (especificado após o caractere # no URL) é *conformance* e um navegador procurará esse identificador de fragmento dentro do texto HTML, após ter baixado o arquivo inteiro. O terceiro URL especifica uma consulta para um mecanismo de busca. O caminho identifica um programa chamado de “search” e o *string* após o caractere ? codifica um *string* de consulta fornecido como argumento para esse programa. Discutiremos os URLs que identificam recursos de programa com mais detalhes a seguir, quando considerarmos os recursos mais avançados.

Publicação de um recurso: Embora a Web tenha um modelo claramente definido para acessar um recurso a partir de seu URL, os métodos exatos para publicação de recursos dependem da implementação do servidor. Em termos de mecanismos de baixo nível, o método mais simples de publicação de um recurso na Web é colocar o arquivo correspondente em um diretório que o servidor Web possa acessar. Sabendo o nome do servidor, *S*, e um nome de caminho para o arquivo, *C*, que o servidor possa reconhecer, o usuário constrói o URL como *http://S/C*. O usuário coloca esse URL em um *link* de um documento já existente ou informa esse URL para outros usuários de diversas formas como, por exemplo, por *e-mail*.

É comum tais preocupações ficarem ocultas dos usuários, quando eles geram conteúdo. Por exemplo, os *bloggers* normalmente usam ferramentas de *software*, elas próprias implementadas como páginas Web, para criar coleções de páginas de diário organizadas. As páginas de produtos do *site* de uma empresa normalmente são criadas com um *sistema de gerenciamento de conteúdo*; novamente, pela interação direta com o *site*, por meio de páginas Web administrativas. O banco de dados, ou sistema de arquivos, no qual as páginas de produtos estão baseadas é transparente.

Por fim, Huang *et al.* [2000] fornece um modelo para inserir conteúdo na Web com mínima intervenção humana. Isso é particularmente relevante quando os usuários precisam extrair conteúdo de uma variedade de equipamentos, como câmeras, para publicação em páginas Web.

HTTP • O protocolo HyperText Transfer Protocol [[www.w3.org IV](http://www.w3.org/Protocols/HTTP/)] define as maneiras pelas quais os navegadores e outros tipos de cliente interagem com os servidores Web. O Capítulo 5 examina o protocolo HTTP com mais detalhes, mas destacaremos aqui suas principais características (restringindo nossa discussão à recuperação de recursos em arquivos):

Interações requisição-resposta: o protocolo HTTP é do tipo requisição-resposta. O cliente envia uma mensagem de requisição para o servidor, contendo o URL do recurso solicitado. O servidor pesquisa o nome de caminho e, se ele existir, envia de volta para o cliente o conteúdo do recurso em uma mensagem de resposta. Caso contrário, ele envia de volta uma resposta de erro, como a conhecida mensagem *404 Not Found*. O protocolo HTTP define um conjunto reduzido de operações ou *métodos* que podem ser executados em um recurso. Os mais comuns são GET, para recuperar dados do recurso, e POST, para fornecer dados para o recurso.

Tipos de conteúdo: os navegadores não são necessariamente capazes de manipular todo tipo de conteúdo. Quando um navegador faz uma requisição, ele inclui uma lista dos tipos de conteúdo que prefere – por exemplo, em princípio, ele poderia exibir imagens no formato GIF, mas não no formato JPEG. O servidor pode levar isso em conta ao retornar conteúdo para o navegador. O servidor inclui o tipo de conteúdo na mensagem de resposta para que o navegador saiba como processá-lo. Os *strings* que denotam o tipo de conteúdo são chamados de tipos MIME e estão padronizados na RFC 1521 [Freed e Borenstein 1996]. Por exemplo, se o conteúdo é de tipo *text/html*, então um navegador interpreta o texto como HTML e o exibe; se o conteúdo é de tipo *image/GIF*, o navegador o representa como uma imagem no formato GIF; se é do tipo *application/zip*, dados compactados no formato zip, o navegador ativa um aplicativo auxiliar externo para descompactá-lo. O conjunto de ações a serem executadas pelo navegador para determinado tipo de conteúdo pode ser configurado, e os leitores devem verificar essas configurações em seus próprios navegadores.

Um recurso por requisição: os clientes especificam um recurso por requisição HTTP. Se uma página Web contém, digamos, nove imagens, o navegador emite um total de 10 requisições separadas para obter o conteúdo inteiro da página. Normalmente, os navegadores fazem vários pedidos concorrentes para reduzir o atraso global para o usuário.

Controle de acesso simplificado: por padrão, qualquer usuário com conectividade de rede para um servidor Web pode acessar qualquer um de seus recursos publicados. Se for necessário restringir o acesso a determinados recursos, isso pode ser feito configurando o servidor de modo a emitir um pedido de identificação para qualquer cliente que o solicite. Então, o usuário precisa provar que tem direito de acessar o recurso, por exemplo, digitando uma senha.

Páginas dinâmicas • Até aqui, descrevemos como os usuários podem publicar páginas Web e outros tipos de conteúdos armazenados em arquivos na Web. Entretanto, grande parte da experiência dos usuários na Web é a interação com serviços, em vez da simples recuperação de informações. Por exemplo, ao adquirir um item em uma loja *online*, o usuário frequentemente preenche um *formulário* para fornecer seus dados pessoais ou para especificar exatamente o que deseja adquirir. Um formulário Web é uma página contendo instruções para o usuário e elementos de janela para entrada de dados, como campos de texto e caixas de seleção. Quando o usuário envia o formulário (normalmente, clicando sobre um botão no próprio formulário ou pressionando a tecla *return*), o navegador envia um pedido HTTP para um servidor Web, contendo os valores inseridos pelo usuário.

Como o resultado do pedido depende da entrada do usuário, o servidor precisa *processar* a entrada do usuário. Portanto, o URL, ou seu componente inicial, designa um *programa* no servidor, e não um arquivo. Se os dados de entrada fornecidos pelo usuário forem um conjunto de parâmetros razoavelmente curto, então normalmente eles são enviados como o componente de *consulta* do URL, usando o método GET; alternativamente, eles são enviados como dados adicionais no pedido, usando o método POST. Por exemplo, um pedido contendo o URL a seguir ativa um programa chamado “search” no endereço www.google.com e especifica o *string* de consulta “obama”: <http://www.google.com/search?q=obama>.

Esse programa “search” produz texto em HTML na saída, e o usuário vê uma listagem das páginas que contêm a palavra “obama”. (O leitor pode inserir uma consulta em seu mecanismo de busca predileto e observar o URL exibido pelo navegador, quando o resultado for retornado.) O servidor retorna o texto em HTML gerado pelo programa, exatamente como se tivesse sido recuperado de um arquivo. Em outras palavras, a diferença entre o conteúdo estático recuperado a partir de um arquivo e o conteúdo gerado dinamicamente é transparente para o navegador.

Um programa que os servidores Web executam para gerar conteúdo para seus clientes é referido como programa CGI (Common Gateway Interface). Um programa CGI pode ter qualquer funcionalidade específica do aplicativo, desde que possa analisar os argumentos fornecidos pelo cliente e produzir conteúdo do tipo solicitado (normalmente, texto HTML). Frequentemente, o programa consulta ou atualiza um banco de dados no processamento do pedido.

Código baixado: um programa CGI é executado no servidor. Às vezes, os projetistas de serviços Web exigem que algum código relacionado ao serviço seja executado pelo navegador no computador do usuário. Em particular, código escrito em Javascript [www.netscape.com] muitas vezes é baixado com uma página contendo um formulário para proporcionar uma interação de melhor qualidade com o usuário, em vez daquela suportada pelos elementos de janela padrão do protocolo HTML. Uma página aprimorada com Javascript pode dar ao usuário retorno imediato sobre entradas inválidas (em vez de obrigar o usuário a verificar os valores no servidor, o que seria muito mais demorado).

O código Javascript pode ser usado para atualizar partes do conteúdo de uma página Web, sem a busca de uma nova versão inteira da página e sem sua reapresentação. Essas atualizações dinâmicas ocorrem devido a uma ação do usuário (como um clique em um *link* ou em um botão de opção) ou quando o navegador recebe novos dados do servidor que forneceu a página Web. Neste último caso, como a temporização da chegada de dados é desconectada de qualquer ação do usuário no próprio navegador, ela é chamada de *assíncrona*, em que é usada uma técnica conhecida como *AJAX (Asynchronous Javascript And XML)*. AJAX está descrita mais completamente na Seção 2.3.2.

Uma alternativa para um programa em Javascript é um *applet*: um aplicativo escrito na linguagem Java que o navegador baixa e executa automaticamente ao buscar a página

Web correspondente. Os *applets* podem acessar a rede e fornecer interfaces de usuário personalizadas. Por exemplo, às vezes, os aplicativos de bate-papo são implementados como *applets* que são executados nos navegadores dos usuários, junto a um programa servidor. Nesse caso, os *applets* enviam o texto dos usuários para o servidor, o qual, por sua vez, o redistribui para os demais *applets* para serem apresentados aos outros usuários. Discutiremos os *applets* com mais detalhes na Seção 2.3.1.

Serviços Web (Web services) • Até aqui, discutimos a Web do ponto de vista de um usuário operando um navegador. No entanto, outros programas, além dos navegadores, também podem ser clientes Web; na verdade, o acesso por meio de programas aos recursos da Web é muito comum.

Entretanto, sozinho, o padrão HTML é insuficiente para realizar interações por meio de programas. Há uma necessidade cada vez maior na Web de trocar dados de tipos estruturados, mas o protocolo HTML não possui essa capacidade – ele está limitado a realizar a navegação em informações. O protocolo HTML tem um conjunto estático de estruturas, como parágrafos, e elas estão restritas às maneiras de apresentar dados aos usuários. A linguagem XML (Extensible Markup Language) (consule a Seção 4.3.3) foi projetada como um modo de representar dados em formas padronizadas, estruturadas e específicas para determinado aplicativo. Em princípio, os dados expressos em XML são portáveis entre os aplicativos, pois são *autodescritivos* – eles contêm os nomes, os tipos e a estrutura dos seus elementos. Por exemplo, XML pode ser usada para descrever os recursos de dispositivos ou informações sobre usuários para muitos serviços ou aplicações diferentes. Na Web, ela é usada para fornecer e recuperar dados de recursos em operações POST e GET do protocolo HTTP, assim como de outros protocolos. No AJAX, ela é usada para fornecer dados para programas Javascript em navegadores.

Os recursos Web fornecem operações específicas para um serviço. Por exemplo, na loja eletrônica amazon.com, as operações do serviço Web incluem pedidos de livros e verificação da situação atual de um pedido. Conforme mencionamos, o protocolo HTTP fornece um conjunto reduzido de operações aplicáveis a qualquer recurso. Isso inclui principalmente os métodos GET e POST em recursos já existentes e as operações PUT e DELETE, para criar e excluir recursos Web, respectivamente. Qualquer operação em um recurso pode ser ativada com o método GET ou POST, com o conteúdo estruturado para especificar os parâmetros da operação, os resultados e as respostas a erros. A assim chamada arquitetura REST (REpresentational State Transfer) para serviços Web [Fielding 2000] adota essa estratégia com base em sua capacidade de extensão: todo recurso na Web tem um URL e responde ao mesmo conjunto de operações, embora o processamento das operações possa variar bastante de um recurso para outro. O outro lado da moeda dessa capacidade de extensão pode ser a falta de robustez no funcionamento do *software*. O Capítulo 9 descreve melhor a arquitetura REST e examina com mais profundidade a estrutura dos serviços Web, a qual permite aos projetistas de serviços Web descrever mais especificamente para os programadores quais são as operações específicas do serviço disponíveis e como os clientes devem acessá-las.

Discussão sobre a Web • O sucesso fenomenal da Web baseia-se na relativa facilidade com que muitas fontes individuais e organizacionais podem publicar recursos, na conveniência de sua estrutura de hipertexto para organizar muitos tipos de informação e no fato de sua arquitetura ser um sistema aberto. Os padrões nos quais sua arquitetura está baseada são simples e foram amplamente publicados desde seu início. Eles têm permitido a integração de muitos tipos novos de recursos e serviços.

O sucesso da Web esconde alguns problemas de projeto. Primeiramente, seu modelo de hipertexto é deficiente sob alguns aspectos. Se um recurso é excluído ou movido, os assim

chamados *links* “pendentes” para esse recurso ainda podem permanecer, causando frustração para os usuários. E há o conhecido problema de usuários “perdidos no hiperespaço”. Frequentemente, os usuários ficam confusos, seguindo muitos *links* distintos, referenciando páginas de um conjunto de fontes discrepantes e, em alguns casos, de confiabilidade duvidosa.

Os mecanismos de busca são uma alternativa para localizar informações na Web, mas são imperfeitos para obter o que o usuário pretende especificamente. Um modo de tratar desse problema, exemplificado no Resource Description Framework [www.w3.org/V], é produzir vocabulários, sintaxe e semântica padrões para expressar metadados sobre as coisas de nosso mundo e encapsular esses metadados nos recursos Web correspondentes para acesso por meio de programas. Em vez de pesquisar palavras que ocorrem em páginas Web, os programas de busca podem então, em princípio, realizar pesquisas nos metadados para compilar listas de *links* relacionados com base na correspondência semântica. Coleтивamente, esse emaranhado de recursos de metadados vinculados é conhecido como *Web semântica*.

Como uma arquitetura de sistema, a Web enfrenta problemas de escalabilidade. Os servidores Web mais populares podem ter muitos acessos (*hits*) por segundo e, como resultado, a resposta para os usuários pode ser lenta. O Capítulo 2 descreve o uso de cache em navegadores e de servidores *proxies* para melhorar o tempo de resposta e a divisão da carga de processamento por um grupo de computadores.

1.7 Resumo

Os sistemas distribuídos estão em toda parte. A Internet permite que usuários de todo o mundo acessem seus serviços onde quer que estejam. Cada organização gerencia uma intranet, a qual fornece serviços locais e serviços de Internet para usuários locais e remotos. Sistemas distribuídos de pequena escala podem ser construídos a partir de computadores móveis e outros dispositivos computacionais portáteis interligados através de redes sem fio.

O compartilhamento de recursos é o principal fator de motivação para a construção de sistemas distribuídos. Recursos como impressoras, arquivos, páginas Web ou registros de banco de dados são gerenciados por servidores de tipo apropriado; por exemplo, servidores Web gerenciam páginas Web. Os recursos são acessados por clientes específicos; por exemplo, os clientes dos servidores Web geralmente são chamados de navegadores.

A construção de sistemas distribuídos gera muitos desafios:

Heterogeneidade: eles devem ser construídos a partir de uma variedade de redes, sistemas operacionais, *hardware* e linguagens de programação diferentes. Os protocolos de comunicação da Internet mascaram a diferença existente nas redes e o *middleware* pode cuidar das outras diferenças.

Sistemas abertos: os sistemas distribuídos devem ser extensíveis – o primeiro passo é publicar as interfaces dos componentes, mas a integração de componentes escritos por diferentes programadores é um desafio real.

Segurança: a criptografia pode ser usada para proporcionar proteção adequada para os recursos compartilhados e para manter informações sigilosas em segredo, quando são transmitidas em mensagens por uma rede. Os ataques de negação de serviço ainda são um problema.

Escalabilidade: um sistema distribuído é considerado escalável se o custo da adição de um usuário for um valor constante, em termos dos recursos que devem ser adicionados. Os algoritmos usados para acessar dados compartilhados devem evitar gargalos

de desempenho, e os dados devem ser estruturados hierarquicamente para se obter os melhores tempos de acesso. Os dados acessados frequentemente podem ser replicados.

Tratamento de falhas: qualquer processo, computador ou rede pode falhar, independentemente dos outros. Portanto, cada componente precisa conhecer as maneiras possíveis pelas quais os componentes de que depende podem falhar e ser projetado de forma a tratar de cada uma dessas falhas apropriadamente.

Concorrência: a presença de múltiplos usuários em um sistema distribuído é uma fonte de pedidos concorrentes para seus recursos. Em um ambiente concorrente, cada recurso deve ser projetado para manter a consistência nos estados de seus dados.

Transparência: o objetivo é tornar certos aspectos da distribuição invisíveis para o programador de aplicativos, para que este se preocupe apenas com o projeto de seu aplicativo em particular. Por exemplo, ele não precisa estar preocupado com sua localização ou com os detalhes sobre como suas operações serão acessadas por outros componentes, nem se será replicado ou migrado. As falhas de rede e de processos podem ser apresentadas aos programadores de aplicativos na forma de exceções – mas elas devem ser tratadas.

Qualidade do serviço: Não basta fornecer acesso aos serviços em sistemas distribuídos. Em particular, também é importante dar garantias das qualidades associadas ao acesso aos serviços. Exemplos dessas qualidades incluem parâmetros relacionados ao desempenho, à segurança e à confiabilidade.

Exercícios

- 1.1 Cite cinco tipos de recurso de *hardware* e cinco tipos de recursos de dados ou de *software* que possam ser compartilhados com sucesso. Dê exemplos práticos de seu compartilhamento em sistemas distribuídos. *páginas 2, 14*
- 1.2 Como os relógios de dois computadores ligados por uma rede local podem ser sincronizados sem referência a uma fonte de hora externa? Quais fatores limitam a precisão do procedimento que você descreveu? Como os relógios de um grande número de computadores conectados pela Internet poderiam ser sincronizados? Discuta a precisão desse procedimento. *página 2*
- 1.3 Considere as estratégias de implementação de MMOG (massively multiplayer online games) discutidas na Seção 1.2.2. Em particular, quais vantagens você vê em adotar a estratégia de servidor único para representar o estado do jogo para vários jogadores? Quais problemas você consegue identificar e como eles poderiam ser resolvidos? *página 5*
- 1.4 Um usuário chega a uma estação de trem que nunca havia visitado, portando um PDA capaz de interligação em rede sem fio. Sugira como o usuário poderia receber informações sobre serviços locais e comodidades dessa estação, sem digitar o nome ou os atributos da estação. Quais desafios técnicos devem ser superados? *página 13*
- 1.5 Compare e contraste a computação em nuvem com a computação cliente-servidor mais tradicional. O que há de novo em relação à computação em nuvem como conceito? *páginas 13, 14*
- 1.6 Use a World Wide Web como exemplo para ilustrar o conceito de compartilhamento de recursos, cliente e servidor. Quais são as vantagens e desvantagens das tecnologias básicas HTML, URLs e HTTP para navegação em informações? Alguma dessas tecnologias é conveniente como base para a computação cliente-servidor em geral? *páginas 14, 26*

- 1.7 Um programa servidor escrito em uma linguagem (por exemplo, C++) fornece a implementação de um objeto BLOB destinado a ser acessado por clientes que podem estar escritos em outra linguagem (por exemplo, Java). Os computadores cliente e servidor podem ter *hardware* diferente, mas todos eles estão ligados em uma rede. Descreva os problemas devidos a cada um dos cinco aspectos da heterogeneidade que precisam ser resolvidos para que seja possível um objeto cliente invocar um método no objeto servidor. *página 16*
- 1.8 Um sistema distribuído aberto permite que novos serviços de compartilhamento de recursos (como o objeto BLOB do Exercício 1.7) sejam adicionados e acessados por diversos programas clientes. Discuta, no contexto desse exemplo, até que ponto as necessidades de abertura do sistema diferem das necessidades da heterogeneidade. *página 17*
- 1.9 Suponha que as operações do objeto BLOB sejam separadas em duas categorias – operações públicas que estão disponíveis para todos os usuários e operações protegidas, que estão disponíveis somente para certos usuários nomeados. Indique todos os problemas envolvidos para se garantir que somente os usuários nomeados possam usar uma operação protegida. Supondo que o acesso a uma operação protegida forneça informações que não devem ser reveladas para todos os usuários, quais outros problemas surgem? *página 18*
- 1.10 O serviço INFO gerencia um conjunto potencialmente muito grande de recursos, cada um dos quais podendo ser acessado por usuários de toda a Internet por intermédio de uma chave (um nome de *string*). Discuta uma estratégia para o projeto dos nomes dos recursos que cause a mínima perda de desempenho à medida que o número de recursos no serviço aumenta. Sugira como o serviço INFO pode ser implementado de modo a evitar gargalos de desempenho quando o número de usuários se torna muito grande. *página 19*
- 1.11 Liste os três principais componentes de *software* que podem falhar quando um processo cliente chama um método em um objeto servidor, dando um exemplo de falha em cada caso. Sugira como os componentes podem ser feitos de modo a tolerar as falhas uns dos outros. *página 21*
- 1.12 Um processo servidor mantém um objeto de informação compartilhada, como o objeto BLOB do Exercício 1.7. Dê argumentos contra permitir que os pedidos do cliente sejam executados de forma concorrente pelo servidor e a favor disso. No caso de serem executados de forma concorrente, dê um exemplo de uma possível “interferência” que pode ocorrer entre as operações de diferentes clientes. Sugira como essa interferência pode ser evitada. *página 22*
- 1.13 Um serviço é implementado por vários servidores. Explique por que recursos poderiam ser transferidos entre eles. Seria satisfatório para os clientes fazer *multicast* (difusão seletiva) de todos os pedidos para o grupo de servidores, como uma maneira de proporcionar transparência de mobilidade para os clientes? *página 23*
- 1.14 Os recursos na World Wide Web e outros serviços são nomeados por URLs. O que denotam as iniciais URL? Dê exemplos de três diferentes tipos de recursos da Web que podem ser nomeados por URLs. *página 26*
- 1.15 Cite um exemplo de URL HTTP. Liste os principais componentes de um URL HTTP, dizendo como seus limites são denotados e ilustrando cada um, a partir de seu exemplo. Até que ponto um URL HTTP tem transparência de localização? *página 26*

2

Modelos de Sistema

- 2.1 Introdução
- 2.2 Modelos físicos
- 2.3 Modelos de arquitetura para sistemas distribuídos
- 2.4 Modelos fundamentais
- 2.5 Resumo

Este capítulo fornece uma explicação sobre maneiras importantes e complementares pelas quais o projeto de sistemas distribuídos pode ser descrito e discutido:

Os *modelos físicos* consideram os tipos de computadores e equipamentos que constituem um sistema e sua interconectividade, sem os detalhes das tecnologias específicas.

Os *modelos de arquitetura* descrevem um sistema em termos das tarefas computacionais e de comunicação realizadas por seus elementos computacionais – os computadores individuais ou conjuntos deles interligados por conexões de rede apropriadas. Os *modelos cliente-servidor* e *peer-to-peer* são duas das formas mais usadas de arquitetura para sistemas distribuídos.

Os *modelos fundamentais* adotam uma perspectiva abstrata para descrever soluções para os problemas individuais enfrentados pela maioria dos sistemas distribuídos.

Não existe a noção de relógio global em um sistema distribuído; portanto, os relógios de diferentes computadores não fornecem necessariamente a mesma hora. Toda comunicação entre processos é obtida por meio de troca de mensagens. A comunicação por troca de mensagens em uma rede de computadores pode ser afetada por atrasos, sofrer uma variedade de falhas e ser vulnerável a ataques contra a segurança. Esses problemas são tratados por três modelos:

- O modelo de interação, que trata do desempenho e da dificuldade de estabelecer limites de tempo em um sistema distribuído, por exemplo, para entrega de mensagens.
- O modelo de falha, que visa a fornecer uma especificação precisa das falhas que podem ser exibidas por processos e canais de comunicação. Ele define a noção de comunicação confiável e da correção dos processos.
- O modelo de segurança, que discute as possíveis ameaças aos processos e aos canais de comunicação. Ele apresenta o conceito de canal seguro, que é protegido dessas ameaças.

2.1 Introdução

Os sistemas destinados ao uso em ambientes do mundo real devem ser projetados para funcionar corretamente na maior variedade possível de circunstâncias e perante muitas dificuldades e ameaças possíveis (para alguns exemplos, veja o quadro abaixo). A discussão e os exemplos do Capítulo 1 sugerem que sistemas distribuídos de diferentes tipos compartilham importantes propriedades subjacentes e dão origem a problemas de projeto comuns. Neste capítulo, mostramos como as propriedades e os problemas de projeto de sistemas distribuídos podem ser capturados e discutidos por meio do uso de modelos descritivos. Cada tipo de modelo é destinado a fornecer uma descrição abstrata e simplificada, mas consistente, de um aspecto relevante do projeto de um sistema distribuído.

Os modelos físicos são a maneira mais explícita de descrever um sistema; eles capturam a composição de *hardware* de um sistema, em termos dos computadores (e outros equipamentos, incluindo os móveis) e suas redes de interconexão.

Os modelos de arquitetura descrevem um sistema em termos das tarefas computacionais e de comunicação realizadas por seus elementos computacionais – os computadores individuais ou seus agregados (*clusters*) suportados pelas interconexões de rede apropriadas.

Os modelos fundamentais adotam uma perspectiva abstrata para examinar os aspectos individuais de um sistema distribuído. Neste capítulo, vamos apresentar modelos fundamentais que examinam três importantes aspectos dos sistemas distribuídos: *modelos de interação*, que consideram a estrutura e a ordenação da comunicação entre os elementos do sistema; *modelos de falha*, que consideram as maneiras pelas quais um sistema pode deixar de funcionar corretamente; e *modelos de segurança*, que consideram como o sistema está protegido contra tentativas de interferência em seu funcionamento correto ou de roubo de seus dados.

Dificuldades e ameaças para os sistemas distribuídos • Aqui estão alguns dos problemas que os projetistas de sistemas distribuídos enfrentam:

Variados modos de uso: os componentes dos sistemas estão sujeitos a amplos variações na carga de trabalho – por exemplo, algumas páginas Web são acessadas vários milhões de vezes por dia. Alguns componentes de um sistema podem estar desconectados ou mal conectados em parte do tempo – por exemplo, quando computadores móveis são incluídos em um sistema. Alguns aplicativos têm requisitos especiais de necessitar de grande largura de banda de comunicação e baixa latência – por exemplo, aplicativos multimídia.

Ampla variedade de ambientes de sistema: um sistema distribuído deve acomodar *hardware*, sistemas operacionais e redes heterogêneas. As redes podem diferir muito no desempenho – as redes sem fio operam a uma taxa de transmissão inferior a das redes locais cabeadas. Os sistemas computacionais podem apresentar ordens de grandeza totalmente diferentes – variando desde dezenas até milhões de computadores –, devendo ser igualmente suportados.

Problemas internos: relógios não sincronizados, atualizações conflitantes de dados, diferentes modos de falhas de *hardware* e de *software* envolvendo os componentes individuais de um sistema.

Ameaças externas: ataques à integridade e ao sigilo dos dados, negação de serviço, etc.

2.2 Modelos físicos

Um modelo físico é uma representação dos elementos de *hardware* de um sistema distribuído, de maneira a abstrair os detalhes específicos do computador e das tecnologias de rede empregadas.

Modelo físico básico: no Capítulo 1, um sistema distribuído foi definido como aquele no qual os componentes de *hardware* ou *software* localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens. Isso leva a um modelo físico mínimo de um sistema distribuído como sendo um conjunto extensível de nós de computador interconectados por uma rede de computadores para a necessária passagem de mensagens.

Além desse modelo básico, podemos identificar três gerações de sistemas distribuídos:

Sistemas distribuídos primitivos: esses sistemas surgiram no final dos anos 1970 e início dos anos 1980 em resposta ao surgimento da tecnologia de redes locais, normalmente Ethernet (consulte a Seção 3.5). Esses sistemas normalmente consistiam em algo entre 10 e 100 nós interconectados por uma rede local, com conectividade de Internet limitada, e suportavam uma pequena variedade de serviços, como impressoras locais e servidores de arquivos compartilhados, assim como transferências de *e-mail* e arquivos pela Internet. Os sistemas individuais geralmente eram homogêneos e não havia muita preocupação com o fato de serem abertos. O fornecimento de qualidade de serviço ainda se encontrava em um estado muito inicial e era um ponto de atenção de grande parte da pesquisa feita em torno desses sistemas primitivos.

Sistemas distribuídos adaptados para a Internet: aproveitando essa base, sistemas distribuídos de maior escala começaram a surgir nos anos 1990, em resposta ao enorme crescimento da Internet durante essa época (por exemplo, o mecanismo de busca do Google foi lançado em 1996). Nesses sistemas, a infraestrutura física subjacente consiste em um modelo físico (conforme ilustrado no Capítulo 1, Figura 1.3) que é um conjunto extensível de nós interconectados por uma *rede de redes* (a Internet). Esses sistemas exploravam a infraestrutura oferecida pela Internet para desenvolver sistemas distribuídos realmente globais, envolvendo potencialmente grandes números de nós. Surgiram sistemas que forneciam serviços distribuídos para organizações globais e fora dos limites organizacionais. Como resultado, o nível de heterogeneidade nesses sistemas era significativo em termos de redes, arquiteturas de computador, sistemas operacionais, linguagens empregadas e também equipes de desenvolvimento envolvidas. Isso levou a uma ênfase cada vez maior em padrões abertos e tecnologias de *middleware* associadas, como CORBA e, mais recentemente, os serviços Web. Também foram empregados serviços sofisticados para fornecer propriedades de qualidade de serviço nesses sistemas globais.

Sistemas distribuídos contemporâneos: nos sistemas anteriores, os nós normalmente eram computadores de mesa e, portanto, relativamente estáticos (isto é, permaneciam em um local físico por longos períodos de tempo), separados (não incorporados dentro de outras entidades físicas) e autônomos (em grande parte, independentes de outros computadores em termos de sua infraestrutura física). As principais tendências identificadas na Seção 1.3 resultaram em desenvolvimentos significativos nos modelos físicos:

- O surgimento da computação móvel levou a modelos físicos em que nós como *notebooks* ou *smartphones* podem mudar de um lugar para outro em um sistema distribuído, levando à necessidade de mais recursos, como a descoberta de serviço e o suporte para operação conjunta espontânea.

- O surgimento da computação ubíqua levou à mudança de nós distintos para arquiteturas em que os computadores são incorporados em objetos comuns e no ambiente circundante (por exemplo, em lavadoras ou em casas inteligentes de modo geral).
- O surgimento da computação em nuvem e, em particular, das arquiteturas de agregados (*clusters*), levou a uma mudança de nós autônomos – que executavam determinada tarefa – para conjuntos de nós que, juntos, fornecem determinado serviço (por exemplo, um serviço de busca como o oferecido pelo Google).

O resultado final é a uma arquitetura física com um aumento significativo no nível de heterogeneidade, compreendendo, por exemplo, os menores equipamentos incorporados utilizados na computação ubíqua, por meio de elementos computacionais complexos encontrados na computação em grade (*Grid*). Esses sistemas aplicam um conjunto cada vez mais diversificado de tecnologias de interligação em rede em um ampla variedade de aplicativos e serviços oferecidos por essas plataformas. Tais sistemas também podem ser de grande escala, explorando a infraestrutura oferecida pela Internet para desenvolver sistemas distribuídos verdadeiramente globais, envolvendo, potencialmente, números de nós que chegam a centenas de milhares.

Sistemas distribuídos de sistemas • Um relatório recente discute o surgimento de sistemas distribuídos ULS (Ultra Large Scale) [www.sei.cmu.edu]. O relatório captura a complexidade dos sistemas distribuídos modernos, referindo-se a essas arquiteturas (físicas) como *sistemas de sistemas* (espelhando a visão da Internet como uma rede de redes). Um sistema de sistemas pode ser definido como um sistema complexo, consistindo em uma série de subsistemas, os quais são, eles próprios, sistemas que se reúnem para executar uma ou mais tarefas em particular.

Como exemplo de sistema de sistemas, considere um sistema de gerenciamento ambiental para previsão de enchentes. Nesse cenário, existirão redes de sensores implantadas para monitorar o estado de vários parâmetros ambientais relacionados a rios, terrenos propensos à inundação, efeitos das marés, etc. Isso pode, então, ser acoplado a sistemas responsáveis por prever a probabilidade de enchentes, fazendo simulações (frequentemente complexas) em, por exemplo, *clusters computacionais* (conforme discutido no Capítulo 1). Outros sistemas podem ser estabelecidos para manter e analisar dados históricos ou para fornecer sistemas de alerta precoce para partes interessadas fundamentais, via telefones celulares.

Resumo • A evolução histórica global mostrada nesta seção está resumida na Figura 2.1, com a tabela destacando os desafios significativos associados aos sistemas distribuídos contemporâneos, em termos de gerenciamento dos níveis de heterogeneidade e de fornecimento de propriedades importantes, como sistemas abertos e qualidade de serviço.

2.3 Modelos de arquitetura para sistemas distribuídos

A arquitetura de um sistema é sua estrutura em termos de componentes especificados separadamente e suas inter-relações. O objetivo global é garantir que a estrutura atenda às demandas atuais e, provavelmente, às futuras demandas impostas sobre ela. As maiores preocupações são tornar o sistema confiável, gerenciável, adaptável e rentável. O projeto arquitetônico de um prédio tem aspectos similares – ele determina não apenas sua aparência, mas também sua estrutura geral e seu estilo arquitetônico (gótico, neoclássico, moderno), fornecendo um padrão de referência coerente para seu projeto.

<i>Sistemas distribuídos</i>	<i>Primitivos</i>	<i>Adaptados para Internet</i>	<i>Contemporâneos</i>
<i>Escala</i>	Pequenos	Grandes	Ultragrandes
<i>Heterogeneidade</i>	Limitada (normalmente, configurações relativamente homogêneas)	Significativa em termos de plataformas, linguagens e <i>middleware</i>	Maiores dimensões introduzidas, incluindo estilos de arquitetura radicalmente diferentes
<i>Sistemas abertos</i>	Não é prioridade	Prioridade significativa, com introdução de diversos padrões	Grande desafio para a pesquisa, com os padrões existentes ainda incapazes de abranger sistemas complexos
<i>Qualidade de serviço</i>	Em seu início	Prioridade significativa, com introdução de vários serviços	Grande desafio para a pesquisa, com os serviços existentes ainda incapazes de abranger sistemas complexos

Figura 2.1 Gerações de sistemas distribuídos.

Nesta seção, vamos descrever os principais modelos de arquitetura empregados nos sistemas distribuídos – os estilos arquitetônicos desses sistemas. Em particular, preparamos o terreno para um completo entendimento de estratégias como os modelos cliente-servidor, estratégias *peer-to-peer*, objetos distribuídos, componentes distribuídos, sistemas distribuídos baseados em eventos e as principais diferenças entre esses estilos.

A seção adota uma estratégia de três estágios:

- Examinaremos os principais elementos arquitetônicos que servem de base para os sistemas distribuídos modernos, destacando a diversidade de estratégias agora existentes.
- Em seguida, examinaremos os padrões arquitetônicos compostos que podem ser usados isoladamente ou, mais comumente, em combinação, no desenvolvimento de soluções de sistemas distribuídos mais sofisticados.
- Por fim, consideraremos as plataformas de *middleware* que estão disponíveis para suportar os vários estilos de programação que surgem a partir dos estilos arquitetônicos anteriores.

Note que existem muitos compromissos associados à escolhas identificadas neste capítulo, em termos dos elementos arquitetônicos empregados, dos padrões adotados e (quando apropriado) do *middleware* utilizado, afetando, por exemplo, o desempenho e a eficiência do sistema resultante; portanto, entender esses compromissos com certeza é uma habilidade fundamental para se projetar sistemas distribuídos.

2.3.1 Elementos arquitetônicos

Para se entender os elementos fundamentais de um sistema distribuído, é necessário considerar quatro perguntas básicas:

- Quais são as entidades que estão se comunicando no sistema distribuído?
- Como elas se comunicam ou, mais especificamente, qual é o *paradigma de comunicação* utilizado?
- Quais funções e responsabilidades (possivelmente variáveis) estão relacionadas a eles na arquitetura global?
- Como eles são mapeados na infraestrutura distribuída física (qual é sua *localização*)?

Entidades em comunicação • As duas primeiras perguntas anteriores são absolutamente vitais para se entender os sistemas distribuídos: o que está se comunicando e como se comunica, junto à definição de um rico espaço de projeto para o desenvolvedor de sistemas distribuídos considerar. Com relação à primeira pergunta, é útil tratar disso dos pontos de vista do sistema e do problema.

Do ponto de vista do sistema, a resposta normalmente é muito clara, pois as entidades que se comunicam em um sistema distribuído normalmente são *processos*, levando à visão habitual de um sistema distribuído como processos acoplados a paradigmas de comunicação apropriados entre processos (conforme discutido, por exemplo, no Capítulo 4), com duas advertências:

- Em alguns ambientes primitivos, como nas redes de sensores, os sistemas operacionais talvez não suportem abstrações de processo (ou mesmo qualquer forma de isolamento) e, assim, as entidades que se comunicam nesses sistemas são *nós*.
- Na maioria dos ambientes de sistema distribuído, os processos são complementados por *threads*; portanto, rigorosamente falando, as *threads* é que são os pontos extremos da comunicação.

Em um nível, isso é suficiente para modelar um sistema distribuído e, na verdade, os modelos fundamentais considerados na Seção 2.4 adotam essa visão. Contudo, do ponto de vista da programação, isso não basta, e foram propostas abstrações mais voltadas para o problema:

Objetos: os objetos foram introduzidos para permitir e estimular o uso de estratégias orientadas a objeto nos sistemas distribuídos (incluindo o projeto orientado a objeto e as linguagens de programação orientadas a objeto). Nas estratégias baseadas em objetos distribuídos, uma computação consiste em vários objetos interagindo, representando unidades de decomposição naturais para o domínio do problema dado. Os objetos são acessados por meio de interfaces, com uma linguagem de definição de interface (ou IDL, interface definition language) associada fornecendo uma especificação dos métodos definidos nesse objeto. Os objetos distribuídos se tornaram uma área de estudo importante nos sistemas distribuídos e mais considerações serão feitas nos Capítulos 5 e 8.

Componentes: desde sua introdução, vários problemas significativos foram identificados nos objetos distribuídos, e o uso de tecnologia de componente surgiu como uma resposta direta a essas deficiências. Os componentes se parecem com objetos, pois oferecem abstrações voltadas ao problema para a construção de sistemas distribuídos e também são acessados por meio de interfaces. A principal diferença é que os componentes especificam não apenas suas interfaces (fornecidas), mas também as suposições que fazem em termos de outros componentes/interfaces que devem estar presentes para que o componente cumpra sua função, em outras palavras, tornando todas as dependências explícitas e fornecendo um contrato mais completo para a construção do sistema. Essa estratégia mais contratual estimula e permite o desenvolvimento de componentes por terceiros e também promove uma abordagem de composição mais pura para a construção de sistemas distribuídos, por remover dependências ocultas. O *middleware* baseado em componentes frequentemente fornece suporte adicional para áreas importantes, como a implantação e o suporte para programação no lado do servidor [Heineman e Councill 2001]. Mais detalhes sobre as estratégias baseadas em componentes podem ser encontrados no Capítulo 8.

Serviços Web: os serviços Web representam o terceiro paradigma importante para o desenvolvimento de sistemas distribuídos [Alonso *et al.* 2004]. Eles estão intimamente relacionados aos objetos e aos componentes, novamente adotando uma estratégia baseada no encapsulamento de comportamento e no acesso por meio de interfaces. Em contraste, contudo, os serviços Web são intrinsicamente integrados na World Wide Web, usando padrões da Web para representar e descobrir serviços. O W3C (World Wide Web Consortium) define um serviço Web como:

... um aplicativo de *software* identificado por um URI, cujas interfaces e vínculos podem ser definidos, descritos e descobertos como artefatos da XML. Um serviço Web suporta interações diretas com outros agentes de *software*, usando trocas de mensagens baseadas em XML por meio de protocolos Internet.

Em outras palavras, os serviços Web são parcialmente definidos pelas tecnologias baseadas na Web que adotam. Outra distinção importante resulta do estilo de uso da tecnologia. Enquanto os objetos e componentes são frequentemente usados dentro de uma organização para o desenvolvimento de aplicativos fortemente acoplados, os serviços Web geralmente são vistos como serviços completos por si sós, os quais podem, então, ser combinados para se obter serviços de valor agregado, frequentemente ultrapassando os limites organizacionais e, assim, obtendo integração de empresa para empresa. Os serviços Web podem ser implementados por diferentes provedores e usar diferentes tecnologias. Eles serão considerados com mais detalhes no Capítulo 9.

Paradigmas de comunicação • Voltemos agora nossa atenção para como as entidades se comunicam em um sistema distribuído e consideremos três tipos de paradigma de comunicação:

- comunicação entre processos;
- invocação remota;
- comunicação indireta.

Comunicação entre processos: se refere ao suporte de nível relativamente baixo para comunicação entre processos nos sistemas distribuídos, incluindo primitivas de passagem de mensagens, acesso direto à API oferecida pelos protocolos Internet (programação de soquetes) e também o suporte para comunicação em grupo* (*multicast*). Tais serviços serão discutidos em detalhes no Capítulo 4.

A invocação remota: representa o paradigma de comunicação mais comum nos sistemas distribuídos, cobrindo uma variedade de técnicas baseadas na troca bilateral entre as entidades que se comunicam em um sistema distribuído e resultando na chamada de uma operação, um procedimento ou método remoto, conforme melhor definido a seguir (e considerado integralmente no Capítulo 5):

Protocolos de requisição-resposta: os protocolos de requisição-resposta são efetivamente um padrão imposto em um serviço de passagem de mensagens para suportar computação cliente-servidor. Em particular, esses protocolos normalmente envolvem uma troca por pares de mensagens do cliente para o servidor e, então, do servidor de volta para o cliente, com a primeira mensagem contendo uma codificação da operação a ser executada no servidor e também um vetor de bytes contendo argumentos

* N. de R. T.: O envio de uma mensagem pode ter como destino uma única entidade (*unicast*), um subconjunto ou grupo de entidades de um sistema (*multicast*), ou todas as entidades desse sistema (*broadcast*). É comum encontrar tradução apenas para o termo *multicast*, por isso, por coerência, manteremos todos os termos no seu original, em inglês. As traduções normalmente encontradas para *multicast* são: comunicação em grupo ou difusão seletiva.

associados. A segunda mensagem contém os resultados da operação, novamente codificados como um vetor de bytes. Esse paradigma é bastante primitivo e utilizado somente em sistemas em que o desempenho é fundamental. A estratégia também é usada no protocolo HTTP, descrito na Seção 5.2. No entanto, a maioria dos sistemas distribuídos opta por usar chamadas de procedimento remoto ou invocação de método remoto, conforme discutido a seguir; contudo, observe que as duas estratégias são suportadas pelas trocas de requisição-resposta.

Chamada de procedimento remoto: o conceito de chamada de procedimento remoto (RPC, Remote Procedure Call), inicialmente atribuído a Birrell e Nelson [1984], representa uma inovação intelectual importante na computação distribuída. Na RPC, procedimentos nos processos de computadores remotos podem ser chamados como se fossem procedimentos no espaço de endereçamento local. Então, o sistema de RPC subjacente oculta aspectos importantes da distribuição, incluindo a codificação e a decodificação de parâmetros e resultados, a passagem de mensagens e a preservação da semântica exigida para a chamada de procedimento. Essa estratégia suporta a computação cliente-servidor de forma direta e elegante, com os servidores oferecendo um conjunto de operações por meio de uma interface de serviço e os clientes chamando essas operações diretamente, como se estivessem disponíveis de forma local. Portanto, os sistemas de RPC oferecem (no mínimo) transparência de acesso e localização.

Invocação de método remoto: a invocação de método remoto (RMI, Remote Method Invocation) é muito parecida com as chamadas de procedimento remoto, mas voltada para o mundo dos objetos distribuídos. Com essa estratégia, um objeto chamarador pode invocar um método em um objeto potencialmente remoto, usando invocação de método remoto. Assim como na RPC, os detalhes subjacentes geralmente ficam ocultos do usuário. Contudo, as implementações de RMI podem ir mais além, suportando a identidade de objetos e a capacidade associada de passar identificadores de objeto como parâmetros em chamadas remotas. De modo geral, elas também se beneficiam de uma integração mais forte com as linguagens orientadas a objetos, conforme será discutido no Capítulo 5.

Todas as técnicas do grupo anterior têm algo em comum: a comunicação representa uma relação bilateral entre um remetente e um destinatário, com os remetentes direcionando explicitamente as mensagens/invocações para os destinatários associados. Geralmente, os destinatários conhecem a identidade dos remetentes e, na maioria dos casos, as duas partes devem existir ao mesmo tempo. Em contraste, têm surgido várias técnicas nas quais a comunicação é indireta, por intermédio de uma terceira entidade, possibilitando um alto grau de desacoplamento entre remetentes e destinatários. Em particular:

- Os remetentes não precisam saber para quem estão enviando (*desacoplamento espacial*).
- Os remetentes e os destinatários não precisam existir ao mesmo tempo (*desacoplamento temporal*).

A comunicação indireta será discutida com mais detalhes no Capítulo 6.

As principais técnicas de comunicação indireta incluem:

Comunicação em grupo: a comunicação em grupo está relacionada à entrega de mensagens para um conjunto de destinatários e, assim, é um paradigma de comunicação de várias partes, suportando comunicação de um para muitos. A comunicação em grupo conta com a abstração de um grupo, que é representado no sistema por um identificador. Os destinatários optam por receber as mensagens enviadas

para um grupo ingressando nesse grupo. Então, os remetentes enviam mensagens para o grupo por meio do identificador de grupo e, assim, não precisam conhecer os destinatários da mensagem. Normalmente, os grupos também mantêm o registro de membros e incluem mecanismos para lidar com a falha de seus membros.

Sistemas publicar-assinar: muitos sistemas, como o exemplo de negócios financeiros do Capítulo 1, podem ser classificados como sistemas de disseminação de informações, por meio dos quais um grande número de produtores (ou publicadores) distribui itens de informação de interesse (eventos) para um número semelhantemente grande de consumidores (ou assinantes). Seria complicado e ineficiente empregar qualquer um dos paradigmas de comunicação básicos discutidos anteriormente e, assim, surgiram os sistemas publicar-assinar (também chamados de sistemas baseados em eventos distribuídos) para atender a essa importante necessidade [Muhl *et al.* 2006]. Todos os sistemas publicar-assinar compartilham característica fundamental de fornecer um serviço intermediário, o qual garante, eficientemente, que as informações geradas pelos produtores sejam direcionadas para os consumidores que as desejam.

Filas de mensagem: enquanto os sistemas publicar-assinar oferecem um estilo de comunicação de um para muitos, as filas de mensagem oferecem um serviço ponto a ponto por meio do qual os processos produtores podem enviar mensagens para uma fila específica e os processos consumidores recebem mensagens da fila ou são notificados da chegada de novas mensagens na fila. Portanto, as filas oferecem uma indireção entre os processos produtores e consumidores.

Espaços de tupla: os espaços de tupla oferecem mais um serviço de comunicação indireta, suportando um modelo por meio do qual os processos podem colocar itens de dados estruturados arbitrários, chamados tuplas, em um espaço de tupla persistente e outros processos podem ler ou remover tais tuplas desse espaço, especificando padrões de interesse. Como o espaço de tupla é persistente, os leitores e escritores não precisam existir ao mesmo tempo. Esse estilo de programação, também conhecido como comunicação generativa, foi apresentado por Gelernter [1985] como um paradigma para a programação paralela. Também foram desenvolvidas várias implementações distribuídas, adotando um estilo cliente-servidor ou uma estratégia *peer-to-peer* mais descentralizada.

Memória compartilhada distribuída: os sistemas de memória compartilhada distribuída (DSM, Distributed Shared Memory) fornecem uma abstração para compartilhamento de dados entre processos que não compartilham a memória física. Contudo, é apresentada aos programadores uma abstração de leitura ou de escrita de estruturas de dados (compartilhadas) conhecida, como se estivessem em seus próprios espaços de endereçamento locais, apresentando, assim, um alto nível de transparência de distribuição. A infraestrutura subjacente deve garantir o fornecimento de uma cópia de maneira oportuna e também deve tratar dos problemas relacionados ao sincronismo e à consistência dos dados. Uma visão geral da memória compartilhada distribuída pode ser encontrada no Capítulo 6.

As escolhas de arquitetura discutidas até aqui estão resumidas na Figura 2.2.

Funções e responsabilidades • Em um sistema distribuído, os processos (ou, na verdade, os objetos), componentes ou serviços, incluindo serviços Web (mas, por simplicidade, usamos o termo processo em toda esta seção), interagem uns com os outros para realizar uma atividade útil; por exemplo, para suportar uma sessão de bate-papo. Ao fazer isso, os processos assumem determinadas funções e, de fato, esse estilo de função é fundamental

Entidades em comunicação (o que se comunica)		Paradigmas de comunicação (como se comunicam)		
Orientados a sistemas	Orientados a problemas	Entre processos	Invocação remota	Comunicação indireta
Nós	Objetos	Passagem de mensagem	Requisição-resposta	Comunicação em grupo
Processos	Componentes	Soquetes	RPC	Publicar-assinar
	Serviços Web	Multicast	RMI	Fila de mensagem Espaço de tupla DSM

Figura 2.2 Entidades e paradigmas de comunicação.

no estabelecimento da arquitetura global a ser adotada. Nesta seção, examinaremos dois estilos de arquitetura básicos resultantes da função dos processos individuais: cliente-servidor e *peer-to-peer*.

Cliente-servidor: essa é a arquitetura mais citada quando se discute os sistemas distribuídos. Historicamente, ela é a mais importante e continua sendo amplamente empregada. A Figura 2.3 ilustra a estrutura simples na qual os processos assumem os papéis de clientes ou servidores. Em particular, os processos clientes interagem com processos servidores, localizados possivelmente em distintos computadores hospedeiros, para acessar os recursos compartilhados que estes gerenciam.

Os servidores podem, por sua vez, ser clientes de outros servidores, conforme a figura indica. Por exemplo, um servidor Web é frequentemente um cliente de um servidor de arquivos local que gerencia os arquivos nos quais as páginas Web estão armazenadas. Os servidores Web, e a maioria dos outros serviços Internet, são clientes do serviço DNS, que mapeia nomes de domínio Internet a endereços de rede (IP). Outro exemplo relacionado à Web diz respeito aos *mecanismos de busca*, os quais permitem aos usuários pesquisar resumos de informações disponíveis em páginas Web em *sites* de toda a Internet. Esses resumos são feitos por programas chamados *Web crawlers** , que são executados em segundo plano (*background*) em um *site* de mecanismo de busca, usando pedidos HTTP para acessar servidores Web em toda a Internet. Assim, um mecanismo de busca é tanto um servidor como um cliente: ele responde às consultas de clientes navegadores e executa *Web crawlers* que atuam como clientes de outros servidores Web. Nesse exemplo, as tarefas do servidor (responder às consultas dos usuários) e as tarefas do *Web crawler* (fazer pedidos para outros servidores Web) são totalmente independentes; há pouca necessidade de sincronizá-las e elas podem ser executadas concomitantemente. Na verdade, um mecanismo de busca típico, normalmente, é feito por muitas *threads* concorrentes, algumas servindo seus clientes e outras executando *Web crawlers*. No Exercício 2.5, o leitor é convidado a refletir sobre o problema de sincronização que surge para um mecanismo de busca concorrente do tipo aqui esboçado.

Peer-to-peer:** nessa arquitetura, todos os processos envolvidos em uma tarefa ou atividade desempenham funções semelhantes, interagindo cooperativamente como *pares*

* N. de R.T.: Também denominados *spiders* (aranhas), em analogia ao fato de que passeiam sobre a *Web* (teia); entretanto, é bastante comum o uso do termo *Web crawler* e, por isso, preferimos não traduzi-lo.

** N. de R. T.: Sistemas par-a-par; por questões de clareza, manteremos o termo técnico *peer-to-peer*, em inglês, para denotar a arquitetura na qual os processos (*peers*) não possuem hierarquia entre si.

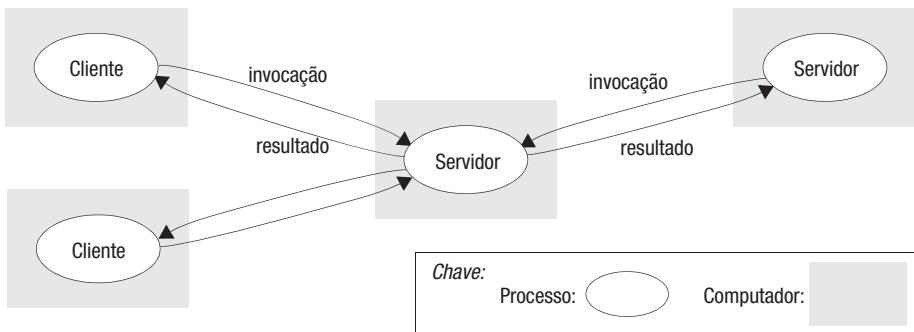


Figura 2.3 Os clientes chamam o servidor individual.

(*peers*), sem distinção entre processos clientes e servidores, nem entre os computadores em que são executados. Em termos práticos, todos os processos participantes executam o mesmo programa e oferecem o mesmo conjunto de interfaces uns para os outros. Embora o modelo cliente-servidor ofereça uma estratégia direta e relativamente simples para o compartilhamento de dados e de outros recursos, ele não é flexível em termos de escalabilidade. A centralização de fornecimento e gerenciamento de serviços, acarretada pela colocação de um serviço em um único computador, não favorece um aumento de escala além daquela limitada pela capacidade do computador que contém o serviço e da largura de banda de suas conexões de rede.

Várias estratégias de posicionamento evoluíram como uma resposta a esse problema (consulte a seção sobre *Posicionamento*, a seguir), mas nenhuma delas trata do problema fundamental – a necessidade de distribuir recursos compartilhados de uma forma mais ampla para dividir as cargas de computação e de comunicação entre um número muito grande de computadores e de conexões de rede. A principal ideia que levou ao desenvolvimento de sistemas *peer-to-peer* foi que a rede e os recursos computacionais pertencentes aos usuários de um serviço também poderiam ser utilizados para suportar esse serviço. Isso tem a consequência vantajosa de que os recursos disponíveis para executar o serviço aumentam com o número de usuários.

A capacidade do *hardware* e a funcionalidade do sistema operacional dos computadores do tipo *desktops* atuais ultrapassam aquelas dos servidores antigos e ainda, a maioria desses computadores, está equipada com conexões de rede de banda larga e sempre ativas. O objetivo da arquitetura *peer-to-peer* é explorar os recursos (tanto dados como de *hardware*) de um grande número de computadores para o cumprimento de uma dada tarefa ou atividade. Tem-se construído, com sucesso, aplicativos e sistemas *peer-to-peer* que permitem a dezenas, ou mesmo, a centenas de milhares de computadores, fornecerem acessos a dados e a outros recursos que eles armazenam e gerenciam coletivamente. Um dos exemplos mais antigos desse tipo de arquitetura é o aplicativo Napster, empregado para o compartilhamento de arquivos de música digital. Embora tenha se tornado famoso por outro motivo que não a sua arquitetura, sua demonstração de exequibilidade resultou no desenvolvimento desse modelo de arquitetura em muitas direções importantes. Um exemplo desse tipo de arquitetura mais recente e amplamente utilizado é o sistema de compartilhamento de arquivos BitTorrent (discutido com mais profundidade na Seção 20.6.2).

A Figura 2.4a ilustra o formato de um aplicativo *peer-to-peer*. Os aplicativos são compostos de grandes números de processos (*peers*) executados em diferentes com-

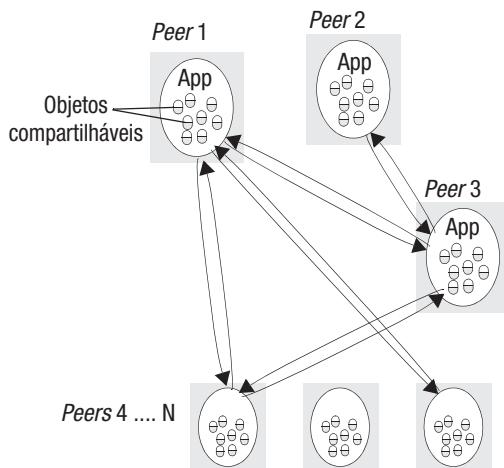


Figura 2.4a Arquitetura peer-to-peer.

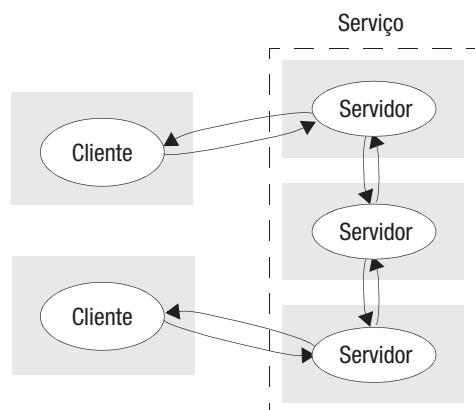


Figura 2.4b Um serviço fornecido por vários servidores.

putadores, e o padrão de comunicação entre eles depende totalmente dos requisitos do aplicativo. Um grande número de objetos de dados são compartilhados, um computador individual contém apenas uma pequena parte do banco de dados do aplicativo e as cargas de armazenamento, processamento e comunicação para acessar os objetos são distribuídas por muitos computadores e conexões de rede. Cada objeto é replicado em vários computadores para distribuir a carga ainda mais e para fornecer poder de recuperação no caso de desconexão de computadores individuais (como, inevitavelmente, acontece nas redes grandes e heterogêneas a que os sistemas *peer-to-peer* se destinam). A necessidade de colocar objetos individuais, recuperá-los e manter réplicas entre muitos computadores torna essa arquitetura significativamente mais complexa do que a arquitetura cliente-servidor.

O desenvolvimento de aplicativos *peer-to-peer* e *middleware* para suportá-los está descrito com profundidade no Capítulo 10.

Posicionamento • O último problema a ser considerado é de que modo entidades como objetos ou serviços são mapeadas na infraestrutura física distribuída subjacente, que possivelmente vai consistir em um grande número de máquinas interconectadas por uma rede de complexidade arbitrária. O posicionamento é fundamental em termos de determinar as propriedades do sistema distribuído, mas obviamente relacionadas ao desempenho, mas também a outros aspectos, como confiabilidade e segurança.

A questão de onde colocar determinado cliente ou servidor em termos de máquinas e os processos dentro delas é uma questão de projeto cuidadoso. O posicionamento precisa levar em conta os padrões de comunicação entre as entidades, a confiabilidade de determinadas máquinas e sua carga atual, a qualidade da comunicação entre as diferentes máquinas, etc. Isso deve ser determinado com forte conhecimento dos aplicativos, sendo que existem algumas diretrizes universais para se obter a melhor solução. Portanto, focamos principalmente as estratégias de posicionamento adicionais a seguir, as quais podem alterar significativamente as características de determinado projeto (embora retornemos ao problema fundamental do mapeamento na infraestrutura física na Seção 2.3.2, a seguir, sob o tema arquitetura em camadas):

- mapeamento de serviços em vários servidores;
- uso de cache;
- código móvel;
- agentes móveis.

Mapeamento de serviços em vários servidores: os serviços podem ser implementados como vários processos servidores em diferentes computadores hospedeiros, interagindo conforme for necessário, para fornecer um serviço para processos clientes (Figura 2.4b). Os servidores podem particionar o conjunto de objetos nos quais o serviço é baseado e distribuí-los entre eles mesmos ou podem, ainda, manter cópias duplicadas deles em vários outros hospedeiros. Essas duas opções são ilustradas pelos exemplos a seguir.

A Web oferece um exemplo comum de particionamento de dados no qual cada servidor Web gerencia seu próprio conjunto de recursos. Um usuário pode usar um navegador para acessar um recurso em qualquer um desses servidores.

Um exemplo de serviço baseado em dados replicados é o NIS (Network Information Service), da Sun, usado para permitir que todos os computadores em uma rede local acessem os mesmos dados de autenticação quando os usuários se conectam. Cada servidor NIS tem sua própria cópia (réplica) de um arquivo de senhas que contém uma lista de nomes de *login* dos usuários e suas respectivas senhas criptografadas. O Capítulo 18 discute as técnicas de replicação em detalhes.

Um tipo de arquitetura em que ocorre uma interação maior entre vários servidores, e por isso denominada arquitetura fortemente acoplada, é o baseado em *cluster**¹, conforme apresentado no Capítulo 1. Um *cluster* é construído a partir de várias, às vezes milhares, de unidades de processamento, e a execução de um serviço pode ser particionada ou duplicada entre elas.

Uso de cache: uma *cache* consiste em realizar um armazenamento de objetos de dados recentemente usados em um local mais próximo a um cliente, ou a um conjunto de clientes em particular, do que a origem real dos objetos em si. Quando um novo objeto é recebido de um servidor, ele é adicionado na cache local, substituindo, se houver necessidade, alguns objetos já existentes. Quando um processo cliente requisita um objeto, o serviço de cache primeiro verifica se possui armazenado uma cópia atualizada desse objeto; caso esteja disponível, ele é entregue ao processo cliente. Se o objeto não estiver armazenado, ou se a cópia não estiver atualizada, ele é acessado diretamente em sua origem. As caches podem ser mantidas nos próprios clientes, ou localizadas em um servidor *proxy* que possa ser compartilhado por eles.

Na prática, o emprego de caches é bastante comum. Por exemplo, os navegadores Web mantêm no sistema de arquivos local uma cache das páginas recentemente visitadas e, antes de exibi-las, com o auxílio de uma requisição HTTP especial, verifica nos servidores originais se as páginas armazenadas na cache estão atualizadas. Um servidor *proxy* Web (Figura 2.5) fornece uma cache compartilhada de recursos Web para máquinas clientes de um ou vários *sites*. O objetivo dos servidores *proxies* é aumentar a disponibilidade e o desempenho do serviço, reduzindo a carga sobre a rede remota e sobre os servidores Web. Os servidores *proxies* podem assumir outras funções, como, por exemplo, serem usados para acessar servidores Web através de um *firewall*.

* N. de R.T.: É comum encontrarmos os termos agregado, ou agrupamento, como tradução da palavra *cluster*. Na realidade existem dois tipos de *clusters*. Os denominados fortemente acoplados são compostos por vários processadores e atuam como multiprocessadores. Normalmente, são empregados para atingir alta disponibilidade e balanceamento de carga. Os fracamente acoplados são formados por um conjunto de computadores interligados em rede e são comumente utilizados para processamento paralelo e de alto desempenho.

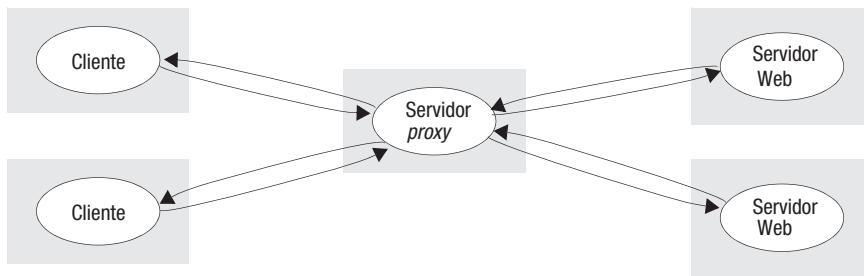


Figura 2.5 Servidor proxy Web.

Código móvel: o Capítulo 1 apresentou o conceito de código móvel. Os *applets* representam um exemplo bem conhecido e bastante utilizado de código móvel – o usuário, executando um navegador, seleciona um *link* que aponta para um *applet*, cujo código é armazenado em um servidor Web; o código é carregado no navegador e, como se vê na Figura 2.6, posteriormente executado. Uma vantagem de executar um código localmente é que ele pode dar uma boa resposta interativa, pois não sofre os atrasos nem a variação da largura de banda associada à comunicação na rede.

Acessar serviços significa executar código que pode ativar suas operações. Alguns serviços são tão padronizados que podemos acessá-los com um aplicativo já existente e bem conhecido – a Web é o exemplo mais comum disso; ainda assim, mesmo nela, alguns *sites* usam funcionalidades não disponíveis em navegadores padrão e exigem o *download* de código adicional. Esse código adicional pode, por exemplo, comunicar-se com um servidor. Considere uma aplicação que exige que os usuários precisem estar atualizados com relação às alterações que ocorrerem em uma fonte de informações em um servidor. Isso não pode ser obtido pelas interações normais com o servidor Web, pois elas são sempre iniciadas pelo cliente. A solução é usar *software* adicional que opere de uma maneira frequentemente referida como modelo *push* – no qual o servidor inicia as interações, em vez do cliente. Por exemplo, um corretor da bolsa de valores poderia fornecer um serviço personalizado para notificar os usuários sobre alterações nos preços das ações. Para usar esse serviço, cada indivíduo teria de fazer o *download* de um *applet* especial que recebesse atualizações do servidor do corretor, exibisse-as para o usuário e, talvez, executasse automaticamente operações de compra e venda, disparadas por condições preestabelecidas e armazenadas por uma pessoa em seu computador.

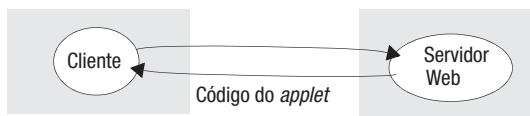
a) Requisição do cliente resulta no download do código de um *applet*b) O cliente interage com o *applet*

Figura 2.6 Applets Web.

O uso de código móvel é uma ameaça em potencial aos recursos locais do computador de destino. Portanto, os navegadores dão aos *applets* um acesso limitado a seus recursos locais usando um esquema discutido na Seção 11.1.1.

Agentes móveis: um agente móvel é um programa em execução (inclui código e dados) que passa de um computador para outro em um ambiente de rede, realizando uma tarefa em nome de alguém, como uma coleta de informações, e finalmente retornando com os resultados obtidos a esse alguém. Um agente móvel pode efetuar várias requisições aos recursos locais de cada *site* que visita como, por exemplo, acessar entradas de banco de dados. Se compararmos essa arquitetura com um cliente estático que solicita, via requisições remotas, acesso a alguns recursos, possivelmente transferindo grandes volumes de dados, há uma redução no custo e no tempo da comunicação, graças à substituição das requisições remotas por requisições locais.

Os agentes móveis podem ser usados para instalar e manter *software* em computadores dentro de uma empresa, ou para comparar os preços de produtos de diversos fornecedores, visitando o *site* de cada fornecedor e executando uma série de operações de consulta. Um exemplo já antigo de uma ideia semelhante é o chamado programa *worm*, desenvolvido no Xerox PARC [Shoch e Hupp 1982], projetado para fazer uso de computadores ociosos para efetuar cálculos intensivos.

Os agentes móveis (assim como o código móvel) são uma ameaça em potencial à segurança para os recursos existentes nos computadores que visitam. O ambiente que recebe um agente móvel deve decidir, com base na identidade do usuário em nome de quem o agente está atuando, qual dos recursos locais ele pode usar. A identidade deve ser incluída de maneira segura com o código e com os dados do agente móvel. Além disso, os agentes móveis, em si, podem ser vulneráveis – eles podem não conseguir completar sua tarefa, caso seja recusado o acesso às informações de que precisam. Para contornar esse problema, as tarefas executadas pelos agentes móveis podem ser feitas usando outras técnicas. Por exemplo, os *Web crawlers*, que precisam acessar recursos em servidores Web em toda a Internet, funcionam com muito sucesso, fazendo requisições remotas de processos servidores. Por esses motivos, a aplicabilidade dos agentes móveis é limitada.

2.3.2 Padrões arquitetônicos

Os padrões arquitetônicos baseiam-se nos elementos de arquitetura mais primitivos discutidos anteriormente e fornecem estruturas recorrentes compostas que mostraram bom funcionamento em determinadas circunstâncias. Eles não são, necessariamente, soluções completas em si mesmos, mas oferecem ideias parciais que, quando combinadas a outros padrões, levam o projetista a uma solução para determinado domínio de problema.

Esse é um assunto amplo e já foram identificados muitos padrões arquitetônicos para sistemas distribuídos. Nesta seção, apresentaremos vários padrões arquitetônicos importantes em sistemas distribuídos, incluindo as arquiteturas de camadas lógicas (*layer*) e de camadas físicas (*tier*), e o conceito relacionado de clientes “leves” (incluindo o mecanismo específico da computação em rede virtual). Examinaremos, também, os serviços Web como um padrão arquitetônico e indicaremos outros que podem ser aplicados em sistemas distribuídos.

Camadas lógicas • O conceito de camadas lógicas é bem conhecido e está intimamente relacionado à abstração. Em uma estratégia de camadas lógicas, um sistema complexo é particionado em várias camadas, com cada uma utilizando os serviços oferecidos pela camada lógica inferior. Portanto, determinada camada lógica oferece uma abstração de

software, com as camadas superiores desconhecendo os detalhes da implementação ou mesmo a existência das camadas lógicas que estão abaixo delas.

Em termos de sistemas distribuídos, isso se equipara a uma organização vertical de serviços em camadas lógicas. Um serviço distribuído pode ser fornecido por um ou mais processos servidores que interagem entre si e com os processos clientes para manter uma visão coerente dos recursos do serviço em nível de sistema. Por exemplo, um serviço de relógio na rede é implementado na Internet com base no protocolo NTP (Network Time Protocol) por processos servidores sendo executados em computadores hospedeiros em toda a Internet. Esses servidores fornecem a hora atual para qualquer cliente que a solicite e ajustam sua versão da hora atual como resultado de interações mútuas. Devido à complexidade dos sistemas distribuídos, frequentemente é útil organizar esses serviços em camadas lógicas. Apresentamos uma visão comum de arquitetura em camadas lógicas na Figura 2.7 e a desenvolveremos em detalhes nos Capítulos 3 a 6.

A Figura 2.7 apresenta os importantes termos *plataforma* e *middleware*, os quais definimos como segue:

- Uma plataforma para sistemas e aplicativos distribuídos consiste nas camadas lógicas de *hardware* e *software* de nível mais baixo. Essas camadas lógicas de baixo nível fornecem serviços para as camadas que estão acima delas, as quais são implementadas independentemente em cada computador, trazendo a interface de programação do sistema para um nível que facilita a comunicação e a coordenação entre os processos. Intel x86/Windows, Intel x86/Solaris, Intel x86/Mac OS X, Intel x86/Linux e ARM/Symbian são bons exemplos.
- O *middleware* foi definido na Seção 1.5.1 como uma camada de *software* cujo objetivo é mascarar a heterogeneidade e fornecer um modelo de programação conveniente para os programadores de aplicativo. O *middleware* é representado por processos ou objetos em um conjunto de computadores que interagem entre si para implementar o suporte para comunicação e compartilhamento de recursos para sistemas distribuídos. Seu objetivo é fornecer elementos básicos úteis para a construção de componentes de *software* que possam interagir em um sistema distribuído. Em particular, ele eleva o nível das atividades de comunicação de programas aplicativos por meio do suporte para abstrações, como a invocação de método remoto,

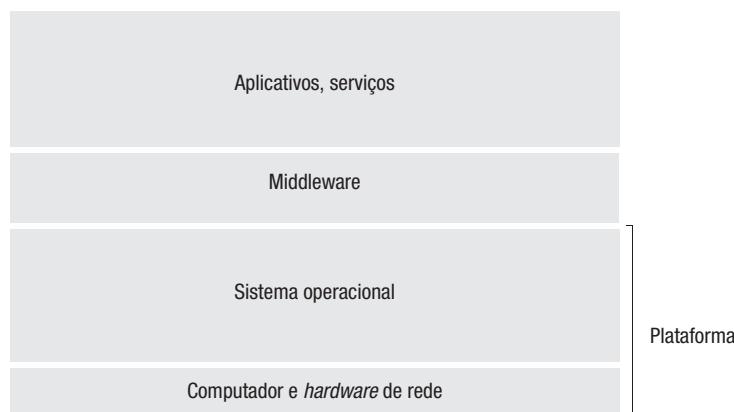


Figura 2.7 Camadas lógicas de serviço de *software* e *hardware* em sistemas distribuídos.

a comunicação entre um grupo de processos, a notificação de eventos, o particionamento, o posicionamento e a recuperação de objetos de dados compartilhados entre computadores colaboradores, a replicação de objetos de dados compartilhados e a transmissão de dados multimídia em tempo real. Vamos voltar a esse importante assunto na Seção 2.3.3, a seguir.

Arquitetura de camadas físicas • As arquiteturas de camadas físicas são complementares às camadas lógicas. Enquanto as camadas lógicas lidam com a organização vertical de serviços em camadas de abstração, as camadas físicas representam uma técnica para organizar a funcionalidade de determinada camada lógica e colocar essa funcionalidade nos servidores apropriados e, como uma consideração secundária, nos nós físicos. Essa técnica é mais comumente associada à organização de aplicativos e serviços, como na Figura 2.7, mas também se aplica a todas as camadas lógicas de uma arquitetura de sistema distribuído.

Vamos examinar primeiro os conceitos da arquitetura de duas e três camadas físicas. Para ilustrar isso, consideremos a decomposição funcional de determinada aplicação, como segue:

- a lógica de apresentação ligada ao tratamento da interação do usuário e à atualização da visão do aplicativo, conforme apresentada a ele;
- a lógica associada à aplicação ligada ao seu processamento detalhado (também referida como lógica do negócio, embora o conceito não esteja limitado apenas a aplicativos comerciais);
- a lógica dos dados ligada ao armazenamento persistente do aplicativo, normalmente em um sistema de gerenciamento de banco de dados.

Agora, vamos considerar a implementação de um aplicativo assim, usando tecnologia cliente-servidor. As soluções de duas e três camadas físicas associadas são apresentadas juntas na Figura 2.8 (a) e (b), respectivamente, para fins de comparação.

Na solução de duas camadas físicas, os três aspectos anteriores devem ser particionados em dois processos, o cliente e o servidor. Mais comumente, isso é feito dividindo-se a lógica da aplicação, com parte dela residindo no cliente e o restante no servidor (embora outras soluções também sejam possíveis). A vantagem desse esquema são as baixas latências em termos de interação, com apenas uma troca de mensagens para ativar uma operação. A desvantagem é a divisão da lógica da aplicação entre limites de processo, com a consequente restrição sobre quais partes podem ser chamadas diretamente de quais outras partes.

Na solução de três camadas físicas, existe um mapeamento de um-para-um de elementos lógicos para servidores físicos e, assim, por exemplo, a lógica da aplicação é mantida em um único lugar, o que, por sua vez, pode melhorar a manutenibilidade do *software*. Cada camada física também tem uma função bem definida; por exemplo, a terceira camada é simplesmente um banco de dados oferecendo uma interface de serviço relacional (possivelmente padronizada). A primeira camada também pode ser uma interface de usuário simples, permitindo suporte intrínseco para clientes magros (conforme discutido a seguir). Os inconvenientes são a maior complexidade do gerenciamento de três servidores e também o maior tráfego na rede e as latências associadas a cada operação.

Note que essa estratégia é generalizada em soluções de n (ou múltiplas) camadas físicas, em que determinado domínio de aplicação é partitionado em n elementos lógicos, cada um mapeado em determinado elemento servidor. Como exemplo, a Wikipedia,

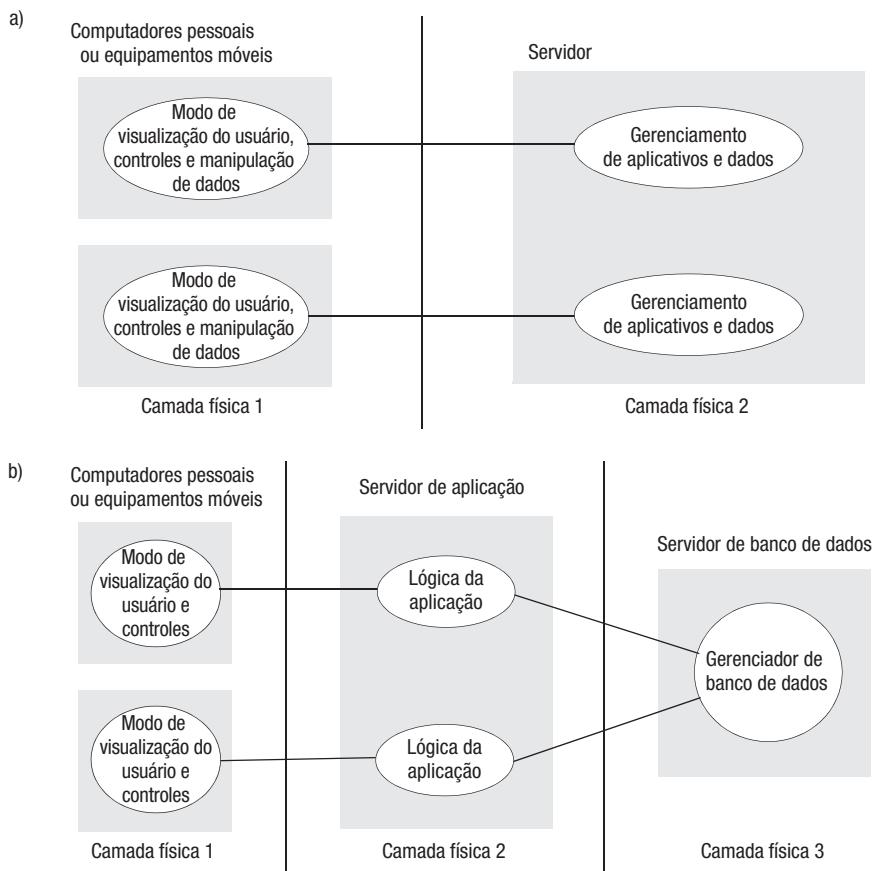


Figura 2.8 Arquitetura de duas e de três camadas físicas.

a enciclopédia baseada na Web que pode ser editada pelo público, adota uma arquitetura de múltiplas camadas físicas para lidar com o alto volume de pedidos Web (até 60.000 pedidos de página por segundo).

Na Seção 1.6, apresentamos o AJAX (Asynchronous Javascript And XML) como uma extensão estilo cliente-servidor padrão de interação, usada na World Wide Web. O AJAX atende às necessidades de comunicação entre um programa Javascript *front-end* sendo executado em um navegador Web e um programa de *back-end* no servidor, contendo dados que descrevem o estado do aplicativo. Para recapitular, no estilo Web padrão de interação, um navegador envia para um servidor uma requisição HTTP, solicitando uma página, uma imagem ou outro recurso com determinado URL. O servidor responde enviando uma página inteira, que foi lida de um arquivo seu ou gerada por um programa, dependendo do tipo de recurso identificado no URL. Quando o conteúdo resultante é recebido no cliente, o navegador o apresenta de acordo com o método de exibição relevante para seu tipo MIME (*text/html*, *image/jpg*, etc.). Embora uma página Web possa ser composta de vários itens de conteúdo de diferentes tipos, a página inteira é composta e apresentada pelo navegador da maneira especificada em sua definição de página HTML.

Esse estilo padrão de interação restringe o desenvolvimento de aplicativos Web de diversas maneiras significativas:

- Uma vez que o navegador tenha feito um pedido HTTP para uma nova página Web, o usuário não pode interagir com ela até que o novo conteúdo HTML seja recebido e apresentado pelo navegador. Esse intervalo de tempo é indeterminado, pois está sujeito a atrasos da rede e do servidor.
- Para atualizar mesmo uma pequena parte da página atual com dados adicionais do servidor, uma nova página inteira precisa ser solicitada e exibida. Isso resulta em uma resposta com atrasos para o usuário, em processamento adicional no cliente e no servidor e em tráfego de rede redundante.
- O conteúdo de uma página exibida em um cliente não pode ser atualizado em resposta a alterações feitas nos dados do aplicativo mantidos no servidor.

A introdução da Javascript, uma linguagem de programação independente de navegador e de plataforma, e que é baixada e executada no navegador, constituiu um primeiro passo na eliminação dessas restrições. Javascript é uma linguagem de propósito geral que permite programar e executar tanto a interface do usuário como a lógica da aplicação no contexto de uma janela de navegador.

O AJAX é o segundo passo inovador que foi necessário para permitir o desenvolvimento e a distribuição de importantes aplicativos Web interativos. Ele permite que programas Javascript *front-end* solicitem novos dados diretamente dos programas servidores. Quaisquer itens de dados podem ser solicitados e a página atual, atualizada seletivamente para mostrar os novos valores. De fato, o *front-end* pode reagir aos novos dados de qualquer maneira que seja útil para o aplicativo.

Muitos aplicativos Web permitem aos usuários acessar e atualizar conjuntos de dados compartilhados de grande porte que podem estar sujeitos à mudança em resposta à entrada de outros clientes ou às transmissões de dados recebidas por um servidor. Eles exigem um componente de *front-end* rápido na resposta executando em cada navegador cliente para executar ações de interface do usuário, como a seleção em um menu, mas também exigem acesso a um conjunto de dados que deve ser mantido no servidor para permitir o compartilhamento. Geralmente, tais conjuntos de dados são grandes e dinâmicos demais para permitir o uso de qualquer arquitetura baseada no *download* de uma cópia do estado do aplicativo inteiro no cliente, no início da sessão de um usuário, para manipulação por parte do cliente.

O AJAX é a “cola” que suporta a construção de tais aplicativos; ele fornece um mecanismo de comunicação que permite aos componentes de *front-end* em execução em um navegador fazer pedidos e receber resultados de componentes de *back-end* em execução em um servidor. Os clientes fazem os pedidos por meio do objeto *XmlHttpRequest* Javascript, o qual gerencia uma troca HTTP (consulte a Seção 1.6) com um processo servidor. Como o objeto *XmlHttpRequest* tem uma API complexa que também é um tanto dependente do navegador, normalmente é acessado por meio de uma das muitas bibliotecas Javascript que estão disponíveis para suportar o desenvolvimento de aplicativos Web. Na Figura 2.9, ilustramos seu uso na biblioteca Javascript *Prototype.js* [www.prototypejs.org].

O exemplo é um trecho de um aplicativo Web que exibe uma página listando placares atualizados de jogos de futebol. Os usuários podem solicitar atualizações dos placares de jogos individuais clicando na linha relevante da página, a qual executa a primeira linha do exemplo. O objeto *Ajax.Request* envia um pedido HTTP para um programa *scores.php*, o qual está localizado no mesmo servidor da página Web. Então, o objeto *Ajax.Request* retorna o controle, permitindo que o navegador continue a responder às outras

```
new Ajax.Request('scores.php?game=Arsenal:Liverpool',
{onSuccess: updateScore});

function updateScore(request) {
.....
  (request  contém o estado do pedido Ajax, incluindo o resultado retornado.
  O resultado é analisado para se obter um texto fornecendo o placar,
  o qual é usado para atualizar a parte relevante da página atual.)
.....
}
```

Figura 2.9 Exemplo de AJAX: atualizações de placar de futebol.

ações do usuário na mesma janela ou em outras. Quando o programa *scores.php* obtém o placar mais recente, ele o retorna em uma resposta HTTP. Então, o objeto *Ajax.Request* é reativado; ele chama a função *updateScore* (pois essa é a ação de *onSuccess*), a qual analisa o resultado e insere o placar na posição relevante da página atual. O restante da página permanece intacto e não é recarregado.

Isso ilustra o tipo de comunicação utilizada entre componentes de camada física 1 e camada física 2. Embora *Ajax.Request* (e o objeto *XmlHttpRequest* subjacente) ofereça comunicação síncrona e assíncrona, quase sempre a versão assíncrona é utilizada, pois o efeito na interface do usuário de respostas de servidor com atrasos é inaceitável.

Nosso exemplo simples ilustra o uso de AJAX em um aplicativo de duas camadas físicas. Em um aplicativo de três camadas físicas, o componente servidor (*scores.php*, em nosso exemplo) enviaria um pedido para um componente gerenciador de dados (normalmente, uma consulta SQL para um servidor de banco de dados) solicitando os dados exigidos. Esse pedido seria síncrono, pois não há motivo para retornar o controle para o componente servidor até que o pedido seja atendido.

O mecanismo AJAX constitui uma técnica eficiente para a construção de aplicativos Web de resposta rápida no contexto da latência indeterminada da Internet e tem sido amplamente implantado. O aplicativo Google Maps [www.google.com II] é um excelente exemplo. Mapas são exibidos como um vetor de imagens adjacentes de 256 x 256 pixels (chamadas de *áreas retangulares – tiles*). Quando o mapa é movido, as áreas retangulares visíveis são repositionadas no navegador por meio de código Javascript e as áreas retangulares adicionais necessárias para preencher a região visível são solicitadas com uma chamada AJAX para um servidor do Google. Elas são exibidas assim que são recebidas, mas o navegador continua a responder à interação do usuário, enquanto elas aguardam.

Clientes “magros”(thin) • A tendência da computação distribuída é retirar a complexidade do equipamento do usuário final e passá-la para os serviços da Internet. Isso fica mais aparente na mudança para a computação em nuvem, conforme discutido no Capítulo 1, mas também pode ser visto em arquiteturas de camadas físicas, conforme discutido anteriormente. Essa tendência despertou o interesse no conceito de *cliente magro (thin)*, dando acesso a sofisticados serviços interligados em rede, fornecidos, por exemplo, por uma solução em nuvem, com poucas suposições ou exigências para o equipamento cliente. Mais especificamente, o termo cliente magro se refere a uma camada de *software* que suporta uma interface baseada em janelas que é local para o usuário, enquanto executa programas aplicativos ou, mais geralmente, acessa serviços em um computador remoto. Por exemplo, a Figura 2.10 ilustra um cliente magro acessando um servidor pela Inter-

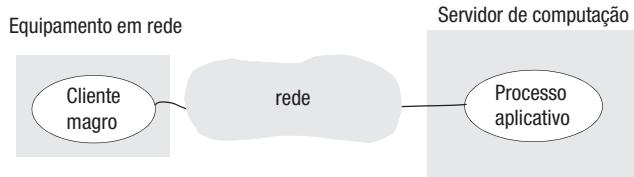


Figura 2.10 Clientes “magros” e servidores.

net. A vantagem dessa estratégia é que um equipamento local potencialmente simples (incluindo, por exemplo, *smartphones* e outros equipamentos com poucos recursos) pode ser melhorado significativamente com diversos serviços e recursos interligados em rede. O principal inconveniente da arquitetura de cliente magro aparece em atividades gráficas altamente interativas, como CAD e processamento de imagens, em que os atrasos experimentados pelos usuário chegam a níveis inaceitáveis, pela necessidade de transferir imagens e informações vetoriais entre o cliente magro e o processo aplicativo, devido às latências da rede e do sistema operacional.

Esse conceito levou ao surgimento da *computação de rede virtual* (VNC, *Virtual Network Computing*). Essa tecnologia foi apresentada pelos pesquisadores da Olivetti e do Oracle Research Laboratory [Richardson *et al.* 1998] e o conceito inicial evoluiu para o RealVNC [www.realvnc.com], que é uma solução de *software*, e também para o Adventiq [www.adventiq.com], que é uma solução baseada em *hardware* que suporta a transmissão de eventos de teclado, vídeo e mouse por meio de IP (KVM-over-IP). Outras soluções VNC incluem Apple Remote Desktop, TightVNC e Aqua Connect.

O conceito é simples: fornecer acesso remoto para interfaces gráficas do usuário. Nessa solução, um cliente VNC (ou visualizador) interage com um servidor VNC por intermédio de um protocolo VNC. O protocolo opera em um nível primitivo, em termos de suporte gráfico, baseado em *framebuffers* e apresentando apenas uma operação, que é o posicionamento de um retângulo de dados de *pixel* em determinado lugar na tela (outras soluções, como XenApp da Citrix, operam em um nível mais alto, em termos de operações de janela [www.citrix.com]). Essa estratégia de baixo nível garante que o protocolo funcione com qualquer sistema operacional ou aplicativo. Embora seja simples, as implicações são que os usuários podem acessar seus recursos de computador a partir de qualquer lugar, em uma ampla variedade de equipamentos, representando, assim, um passo significativo em direção à computação móvel.

A computação de rede virtual substituiu os computadores de rede, uma tentativa anterior de obter soluções de cliente magro por meio de dispositivos de *hardware* simples e baratos totalmente dependentes de serviços de rede, baixando seu sistema operacional e qualquer *software* aplicativo necessário para o usuário a partir de um servidor de arquivos remoto. Como todos os dados e código de aplicativo são armazenados por um servidor de arquivos, os usuários podem migrar de um computador da rede para outro. Na prática, a computação de rede virtual se mostrou uma solução mais flexível e agora domina o mercado.

Outros padrões que ocorrem comumente • Conforme mencionado anteriormente, um grande número de padrões arquitetônicos foi identificado e documentado. Vários exemplos importantes são fornecidos a seguir:

- O padrão *proxy* é recorrente em sistemas distribuídos projetados especificamente para suportar transparência de localização em chamadas de procedimento remoto ou invocação de método remoto. Com essa estratégia, um *proxy* é criado no espaço

de endereçamento local para representar o objeto remoto. Esse *proxy* oferece exatamente a mesma interface do objeto remoto. O programador faz chamadas nesse objeto *proxy* e, assim, não precisa conhecer a natureza distribuída da interação. A função dos objetos *proxy* no suporte para transparência de localização em RPC e RMI está discutida com mais detalhes no Capítulo 5. Note que os objetos *proxy* também podem ser usados para encapsular outra funcionalidade, como políticas de posicionamento de replicação ou uso de cache.

- O uso de *brokerage** em serviços Web pode ser visto como um padrão arquitetônico que suporta interoperabilidade em infraestrutura distribuída potencialmente complexa. Em particular, esse padrão consiste no trio provedor de serviço, solicitante de serviço e corretor de serviço (um serviço que combina os serviços fornecidos com os que foram solicitados), como mostrado na Figura 2.11. Esse padrão de *brokerage* é duplicado em muitas áreas dos sistemas distribuídos; por exemplo, no caso do registro em RMI Java e no serviço de atribuição de nomes do CORBA (conforme discutido nos Capítulos 5 e 8 respectivamente).
- *Reflexão* é um padrão cada vez mais usado em sistema distribuídos, como uma maneira de suportar introspecção (a descoberta dinâmica de propriedades do sistema) e intercessão (a capacidade de modificar estrutura ou comportamento dinamicamente). Por exemplo, os recursos de introspecção da linguagem Java são usados eficientemente na implementação de RMI para fornecer envio genérico (conforme discutido na Seção 5.4.2). Em um sistema refletivo, as interfaces de serviço padrão estão disponíveis em nível básico, mas também está disponível uma interface de meta-nível que dá acesso aos componentes e seus parâmetros envolvidos na obtenção dos serviços. Diversas técnicas estão disponíveis no meta-nível, incluindo a capacidade de interceptar mensagens recebidas ou invocações para descobrir dinamicamente a interface oferecida por determinado objeto e para descobrir e adaptar a arquitetura subjacente do sistema. A reflexão tem sido aplicada em diversas áreas nos sistemas distribuídos, particularmente no campo do *middleware* refletivo; por exemplo, para suportar arquiteturas de *middleware* com maior capacidade de configuração e reconfiguração [Kon *et al.* 2002].

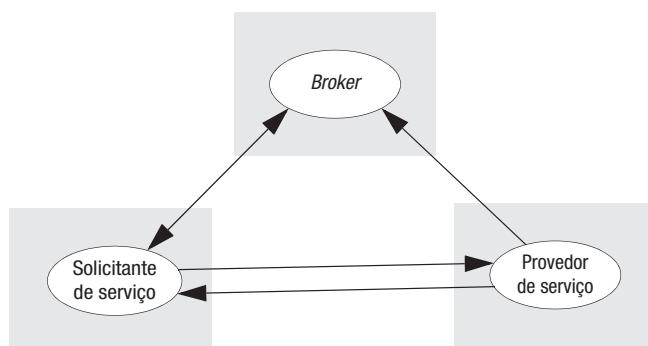


Figura 2.11 O padrão arquitetônico do serviço Web.

* N. de R.T.: Em analogia ao mercado financeiro, em que há um corretor (*broker*) que intermedia a relação entre clientes e empresas com ações no mercado, os termos *broker* e *brokerage* (corretagem) são empregados no provimento de aplicações distribuídas. Por não haver uma tradução aceita, manteremos os termos em inglês.

Mais exemplos de padrões arquitetônicos relacionados aos sistemas distribuídos podem ser encontrados em Buschmann *et al.* [2007].

2.3.3 Soluções de middleware associadas

O *middleware* já foi apresentado no Capítulo 1 e revisto na discussão sobre camadas lógicas, na Seção 2.3.2. A tarefa do *middleware* é fornecer uma abstração de programação de nível mais alto para o desenvolvimento de sistemas distribuídos e, por meio de camadas lógicas, abstrair a heterogeneidade da infraestrutura subjacente para promover a interoperabilidade e a portabilidade. As soluções de *middleware* se baseiam nos modelos arquitetônicos apresentados na Seção 2.3.1 e também suportam padrões arquitetônicos mais complexos. Nesta seção, examinaremos brevemente as principais classes de *middleware* que existem atualmente e preparamos o terreno para um estudo mais aprofundado dessas soluções no restante do livro.

Categorias de middleware • Os pacotes de chamada de procedimento remoto, como Sun RPC (Capítulo 5), e os sistemas de comunicação de grupo, como ISIS (Capítulos 6 e 18) aparecem como os primeiros exemplos de *middleware*. Desde então, uma ampla variedade de estilos de *middleware* tem aparecido, em grande medida baseada nos modelos arquitetônicos apresentados anteriormente. Apresentamos uma taxonomia dessas plataformas de *middleware* na Figura 2.12, incluindo referências cruzadas para outros capítulos que abordam as várias categorias com mais detalhes. Deve-se enfatizar que as classificações não são exatas e que as plataformas de *middleware* modernas tendem a oferecer soluções híbridas. Por exemplo, muitas plataformas de objeto distribuído oferecem serviços de evento distribuído para complementar o suporte mais tradicional para invocação de método remoto. Analogamente, muitas plataformas baseadas em componentes (e mesmo outras categorias de plataforma) também suportam interfaces e padrões de serviço Web por questões de interoperabilidade. Deve-se enfatizar que essa taxonomia não pretende ser completa em termos do conjunto de padrões e tecnologias de *middleware* disponíveis atualmente, mas se destina a indicar as principais classes de *middleware*. Outras soluções (não mostradas) tendem a ser mais específicas, por exemplo, oferecendo paradigmas de comunicação em particular, como passagem de mensagens, chamadas de procedimento remoto, memória compartilhada distribuída, espaços de tupla ou comunicação de grupo.

A classificação de alto nível do *middleware* na Figura 2.12 é motivada pela escolha de entidades que se comunicam e pelos paradigmas de comunicação associados, seguindo cinco dos principais modelos arquitetônicos: serviços Web, objetos distribuídos, componentes distribuídos, sistemas publicar-assinar e filas de mensagem. Isso é complementado por soluções *peer-to-peer*, um ramo distinto do *middleware* baseado na estratégia cooperativa, conforme capturado na discussão relevante da Seção 2.3.1. A subcategoria de componentes distribuídos mostrada como servidores de aplicação também fornece suporte direto para as arquiteturas de três camadas físicas. Em particular, os servidores de aplicação fornecem estrutura para suportar uma separação entre lógica da aplicação e armazenamento de dados, junto ao suporte para outras propriedades, como segurança e confiabilidade. Mais detalhes são deixados para o Capítulo 8.

Além das abstrações de programação, o *middleware* também pode fornecer serviços de sistemas distribuídos de infraestrutura para uso por parte de programas aplicativos ou outros serviços. Esses serviços de infraestrutura são fortemente ligados ao modelo de programação distribuída fornecido pelo *middleware*. Por exemplo, o CORBA (Capítulo 8) fornece aplicativos com uma variedade de serviços CORBA, incluindo suporte para tornar os aplicativos seguros e confiáveis. Conforme mencionado anteriormente, os ser-

<i>Principais categorias</i>	<i>Subcategoria</i>	<i>Exemplos de sistemas</i>
<i>Objetos distribuídos (Capítulos 5, 8)</i>	Padrão	RM-ODP
	Plataforma	CORBA
	Plataforma	Java RMI
<i>Componentes distribuídos (Capítulo 8)</i>	Componentes leves	Fractal
	Componentes leves	OpenCOM
	Servidores de aplicação	SUN EJB
	Servidores de aplicação	CORBA Component Model
<i>Sistemas publicar-assinar (Capítulo 6)</i>	Servidores de aplicação	JBoss
	–	CORBA Event Service
	–	Scribe
	–	JMS
<i>Filas de mensagem (Capítulo 6)</i>	–	Websphere MQ
	–	JMS
<i>Serviços web (Capítulo 9)</i>	Serviços web	Apache Axis
	Serviços de grade	The Globus Toolkit
<i>Peer-to-peer (Capítulo 10)</i>	Sobreposições de roteamento	Pastry
	Sobreposições de roteamento	Tapestry
	Específico da aplicação	Squirrel
	Específico da aplicação	OceanStore
	Específico da aplicação	Ivy
	Específico da aplicação	Gnutella

Figura 2.12 Categorias de middleware.

vidores de aplicação também fornecem suporte intrínseco para tais serviços (também discutido no Capítulo 8).

Limitações do middleware • Muitos aplicativos distribuídos contam completamente com os serviços fornecidos pelo *middleware* para satisfazer suas necessidades de comunicação e de compartilhamento de dados. Por exemplo, um aplicativo que segue o modelo cliente-servidor, como um banco de dados de nomes e endereços, pode ser construído com um *middleware* que forneça somente invocação de método remoto.

Por meio do desenvolvimento do suporte para *middleware*, muito se tem conseguido na simplificação da programação de sistemas distribuídos, mas alguns aspectos da confiabilidade dos sistemas exige suporte em nível de aplicação.

Considere a transferência de grandes mensagens de correio eletrônico, do computador do remetente ao destinatário. À primeira vista, essa é uma simples aplicação do protocolo de transmissão de dados TCP (discutido no Capítulo 3). No entanto, considere o problema de um usuário que tenta transferir um arquivo muito grande por meio de uma rede potencialmente não confiável. O protocolo TCP fornece certa capacidade de detecção e correção de erros, mas não consegue se recuperar de interrupções mais sérias na rede. Portanto, o serviço de transferência de correio eletrônico acrescenta outro nível de

tolerância a falhas, mantendo um registro do andamento e retomando a transmissão em uma nova conexão TCP, caso a original se desfaça.

Um artigo clássico de Saltzer, Reed e Clarke [Saltzer *et al.* 1984] apresenta uma ideia semelhante e valiosa sobre o projeto de sistemas distribuídos, a qual foi chamada de *princípio fim-a-fim*. Parafraseando seu enunciado:

Algumas funções relacionadas à comunicação podem ser completa e corretamente implementadas apenas com o conhecimento e a ajuda da aplicação que está nos pontos extremos de um sistema de comunicação. Portanto, fornecer essa função como um recurso do próprio sistema de comunicação nem sempre é sensato. (Embora uma versão mais simples da função fornecida pelo sistema de comunicação às vezes possa ser útil para melhorar o desempenho).

Pode-se notar que esse princípio vai contra a visão de que todas as atividades de comunicação podem ser abstraídas da programação de aplicações pela introdução de camadas de *middleware* apropriadas.

O ponto principal desse princípio é que o comportamento correto em programas distribuídos depende de verificações, de mecanismos de correção de erro e de medidas de segurança em muitos níveis, alguns dos quais exigindo acesso a dados dentro do espaço de endereçamento da aplicação. Qualquer tentativa de realizar verificações dentro do próprio sistema de comunicação garantirá apenas parte da correção exigida. Portanto, o mesmo trabalho vai ser feito nos programas aplicativos, desperdiçando esforço de programação e, o mais importante, acrescentando complexidade desnecessária e executando operações redundantes.

Não há espaço aqui para detalhar melhor os argumentos que embasam o princípio fim-a-fim; o artigo citado é fortemente recomendado para leitura – ele está repleto de exemplos esclarecedores. Um dos autores originais mostrou, recentemente, que as vantagens significativas trazidas pelo uso do princípio fim-a-fim no projeto da Internet são colocados em risco pelas atuais mudanças na especialização dos serviços de rede para atender aos requisitos dos aplicativos [www.reed.com].

Esse princípio representa um verdadeiro dilema para os projetistas de *middleware* e, sem dúvida, as dificuldades estão aumentando, dada a ampla variedade de aplicações (e condições ambientais associadas) nos sistemas distribuídos contemporâneos (consulte o Capítulo 1). Basicamente, o comportamento correto do *middleware* subjacente é uma função dos requisitos de determinado aplicação ou de um conjunto de aplicações e o contexto ambiental associado, como o estado e o estilo da rede subjacente. Isso está aumentando o interesse nas soluções com reconhecimento de contexto e adaptáveis, no debate sobre *middleware*, conforme discutido em Kon *et al* [2002].

2.4 Modelos fundamentais

Todos os modelos de arquitetura para sistemas distribuídos vistos anteriormente, apesar de bastante diferentes, apresentam algumas propriedades fundamentais idênticas. Em particular, todos são compostos de processos que se comunicam por meio do envio de mensagens através de uma rede de computadores. Ainda, é desejável que todos possuam os mesmos requisitos de projeto, que se preocupam com as características de desempenho e confiabilidade dos processos e das redes de comunicação e com a segurança dos recursos presentes no sistema. Nesta seção, apresentaremos modelos baseados nessas propriedades fundamentais, as quais nos permitem ser mais específicos a respeito de características e das falhas e riscos para a segurança que possam apresentar.

Genericamente, um modelo fundamental deve conter apenas os ingredientes essenciais que precisamos considerar para entender e raciocinar a respeito de certos aspectos do comportamento de um sistema. O objetivo de um modelo é:

- Tornar explícitas todas as suposições relevantes sobre os sistemas que estamos modelando.
- Fazer generalizações a respeito do que é possível ou impossível, dadas essas suposições. As generalizações podem assumir a forma de algoritmos de propósito geral ou de propriedades desejáveis a serem garantidas. Essas garantias dependem da análise lógica e, onde for apropriado, de prova matemática.

Há muito a lucrar com o fato de sabermos do que dependem e do que não dependem nossos projetos. Isso nos permite saber se um projeto funcionará se tentarmos implementá-lo em um sistema específico: só precisamos perguntar se nossas suposições são válidas para esse sistema. Além disso, tornando nossas suposições claras e explícitas, podemos provar matematicamente as propriedades do sistema. Essas propriedades valerão, então, para qualquer sistema que atenda a nossas suposições. Finalmente, a partir do momento que abstraiamo detalhes específicos, como, por exemplo, o *hardware* empregado, e nos concentramos apenas em entidades e características comportamentais essenciais do sistema, podemos compreendê-lo mais facilmente.

Os aspectos dos sistemas distribuídos que desejamos considerar em nossos modelos fundamentais se destinam a nos ajudar a discutir e raciocinar sobre:

Interação: a computação é feita por processos; eles interagem passando mensagens, resultando na comunicação (fluxo de informações) e na coordenação (sincronização e ordenação das atividades) entre eles. Na análise e no projeto de sistemas distribuídos, preocupamo-nos especialmente com essas interações. O modelo de interação deve refletir o fato de que a comunicação ocorre com atrasos que, frequentemente, têm duração considerável. A precisão com a qual processos independentes podem ser coordenados é limitada pelos atrasos de comunicação e pela dificuldade de se manter a mesma noção de tempo entre todos os computadores de um sistema distribuído.

Falha: a operação correta de um sistema distribuído é ameaçada quando ocorre uma falha em qualquer um dos computadores em que ele é executado (incluindo falhas de *software*) ou na rede que os interliga. O modelo de falhas define e classifica as falhas. Isso fornece uma base para a análise de seus efeitos em potencial e para projetar sistemas capazes de tolerar certos tipos de falhas e de continuar funcionando corretamente.

Segurança: a natureza modular dos sistemas distribuídos, aliada ao fato de ser desejável que sigam uma filosofia de sistemas abertos, expõem-nos a ataques de agentes externos e internos. O modelo de segurança define e classifica as formas que tais ataques podem assumir, dando uma base para a análise das possíveis ameaças a um sistema e, assim, guiando seu desenvolvimento de forma a ser capaz de resistir a eles.

Para facilitar a discussão e o raciocínio, os modelos apresentados neste capítulo são simplificados, omitindo-se grande parte dos detalhes existentes em sistemas reais. A relação desses modelos com sistemas reais, assim como os problemas e as soluções que eles apontam, são o objetivo principal deste livro.

2.4.1 Modelo de interação

A discussão sobre arquiteturas de sistema da Seção 2.3 indica que, fundamentalmente, os sistemas distribuídos são compostos por muitos processos, interagindo de maneiras complexas. Por exemplo:

- Vários processos servidores podem cooperar entre si para fornecer um serviço; os exemplos mencionados anteriormente foram o Domain Name Service, que divide e replica seus dados em diferentes servidores na Internet, e o Network Information Service, da Sun, que mantém cópias replicadas de arquivos de senha em vários servidores de uma rede local.
- Um conjunto de processos *peer-to-peer* pode cooperar entre si para atingir um objetivo comum: por exemplo, um sistema de teleconferência que distribui fluxos de dados de áudio de maneira similar, mas com restrições rigorosas de tempo real.

A maioria dos programadores está familiarizada com o conceito de *algoritmo* – uma sequência de passos a serem executados para realizar um cálculo desejado. Os programas simples são controlados por algoritmos em que os passos são rigorosamente sequenciais. O comportamento do programa e o estado das variáveis do programa são determinados por eles. Tal programa é executado por um único processo. Já os sistemas distribuídos são compostos de vários processos, como aqueles delineados anteriormente, o que os torna mais complexos. Seu comportamento e estado podem ser descritos por um *algoritmo distribuído* – uma definição dos passos a serem executados por cada um dos processos que compõem o sistema, *incluindo a transmissão de mensagens entre eles*. As mensagens são enviadas para transferir informações entre processos e para coordenar suas atividades.

Em geral, não é possível prever a velocidade com que cada processo é executado e a sincronização da troca das mensagens entre eles. Também é difícil descrever todos os estados de um algoritmo distribuído, pois é necessário considerar falhas que podem ocorrer em um ou mais dos processos envolvidos ou na própria troca de mensagens.

Em um sistema distribuído, as atividades são realizadas por processos que interagem entre si, porém cada processo tem seu próprio estado, que consiste no conjunto de dados que ele pode acessar e atualizar, incluindo suas variáveis de programa. O estado pertencente a cada processo é privativo, isto é, ele não pode ser acessado, nem atualizado, por nenhum outro processo.

Nesta seção, discutiremos dois fatores que afetam significativamente a interação de processos em um sistema distribuído:

- o desempenho da comunicação, que é, frequentemente, um fator limitante;
- a impossibilidade de manter uma noção global de tempo única.

Desempenho da comunicação • Os canais de comunicação são modelados de diversas maneiras nos sistemas distribuídos; como, por exemplo, por uma implementação de fluxos ou pela simples troca de mensagens em uma rede de computadores. A comunicação em uma rede de computadores tem as seguintes características de desempenho relacionadas à latência, largura de banda e *jitter**:

* N. de R.T.: *Jitter* é a variação estatística do retardo (atraso) na entrega de dados em uma rede, a qual produz uma recepção não regular dos pacotes. Por não haver uma tradução consagrada para esse termo, preferimos mantê-lo em inglês.

- A *latência* é o atraso decorrido entre o início da transmissão de uma mensagem em um processo remetente e o início da recepção pelo processo destinatário. A latência inclui:
 - O tempo que o primeiro bit de um conjunto de bits transmitido em uma rede leva para chegar ao seu destino. Por exemplo, a latência da transmissão de uma mensagem por meio de um enlace de satélite é o tempo necessário para que um sinal de rádio vá até o satélite e retorne à Terra para seu destinatário.
 - O atraso no acesso à rede, que aumenta significativamente quando a rede está muito carregada. Por exemplo, para uma transmissão em uma rede Ethernet, a estação remetente espera que a rede esteja livre de tráfego para poder enviar sua mensagem.
 - O tempo de processamento gasto pelos serviços de comunicação do sistema operacional nos processos de envio e recepção, que varia de acordo com a carga momentânea dos computadores.
- A *largura de banda* de uma rede de computadores é o volume total de informações que pode ser transmitido em determinado momento. Quando um grande número de comunicações usa a mesma rede, elas compartilham a largura de banda disponível.
- *Jitter* é a variação no tempo exigida para distribuir uma série de mensagens. O *jitter* é crucial para dados multimídia. Por exemplo, se amostras consecutivas de dados de áudio são reproduzidas com diferentes intervalos de tempo, o som resultante será bastante distorcido.

Relógios de computador e eventos de temporização • Cada computador possui seu próprio relógio interno, o qual pode ser usado pelos processos locais para obter o valor atual da hora. Portanto, dois processos sendo executados em diferentes computadores podem associar carimbos de tempo (*time stamps*) aos seus eventos. Entretanto, mesmo que dois processos leiam seus relógios locais ao mesmo tempo, esses podem fornecer valores diferentes. Isso porque os relógios de computador se desviam de uma base de tempo e, mais importante, suas taxas de desvio diferem entre si. O termo *taxa de desvio do relógio* (*drift*) se refere à quantidade relativa pela qual um relógio de computador difere de um relógio de referência perfeito. Mesmo que os relógios de todos os computadores de um sistema distribuído fossem inicialmente ajustados com o mesmo horário, com o passar do tempo eles variariam entre si significativamente, a menos que fossem reajustados.

Existem várias estratégias para corrigir os tempos em relógios de computador. Por exemplo, os computadores podem usar receptores de rádio para obter leituras de tempo GPS (Global Positioning System), que oferece uma precisão de cerca de 1 microsegundo. Entretanto, os receptores GPS não funcionam dentro de prédios, nem o seu custo é justificado para cada computador. Em vez disso, um computador que tenha uma fonte de tempo precisa, como o GPS, pode enviar mensagens de sincronização para os outros computadores da rede. É claro que o ajuste resultante entre os tempos nos relógios locais é afetado pelos atrasos variáveis das mensagens. Para ver uma discussão mais detalhada sobre o desvio e a sincronização de relógio, consulte o Capítulo 14.

Duas variantes do modelo de interação • Em um sistema distribuído é muito difícil estabelecer limites para o tempo que leva a execução dos processos, para a troca de mensagens ou para o desvio do relógio. Dois pontos de vistas opostos fornecem modelos simples: o primeiro é fortemente baseado na ideia de tempo, o segundo não.

Sistemas distribuídos síncronos: Hadzilacos e Toueg [1994] definem um sistema distribuído síncrono como aquele no qual são definidos os seguintes pontos:

- o tempo para executar cada etapa de um processo tem limites inferior e superior conhecidos;
- cada mensagem transmitida em um canal é recebida dentro de um tempo limitado, conhecido;
- cada processo tem um relógio local cuja taxa de desvio do tempo real tem um valor máximo conhecido.

Dessa forma, é possível estimar prováveis limites superior e inferior para o tempo de execução de um processo, para o atraso das mensagens e para as taxas de desvio do relógio em um sistema distribuído. No entanto, é difícil chegar a valores realistas e dar garantias dos valores escolhidos. A menos que os valores dos limites possam ser garantidos, qualquer projeto baseado nos valores escolhidos não será confiável. Entretanto, modelar um algoritmo como um sistema síncrono pode ser útil para dar uma ideia sobre como ele se comportará em um sistema distribuído real. Em um sistema síncrono, é possível usar tempos limites para, por exemplo, detectar a falha de um processo, como mostrado na seção sobre o modelo de falha.

Os sistemas distribuídos síncronos podem ser construídos, desde que se garanta que os processos sejam executados de forma a respeitar as restrições temporais impostas. Para isso, é preciso alocar os recursos necessários, como tempo de processamento e capacidade de rede, e limitar o desvio do relógio.

Sistemas distribuídos assíncronos: muitos sistemas distribuídos, como a Internet, são bastante úteis sem apresentarem características síncronas, portanto, precisamos

Consenso em Pepperland* • Duas divisões do exército de Pepperland, Apple e Orange, estão acampadas no topo de duas colinas próximas. Mais adiante, no vale, estão os invasores Blue Meanies. As divisões de Pepperland estão seguras, desde que permaneçam em seus acampamentos, e elas podem, para se comunicar, enviar mensageiros com toda segurança pelo vale. As divisões de Pepperland precisam concordar sobre qual delas liderará o ataque contra os Blue Meanies e sobre quando o ataque ocorrerá. Mesmo em uma Pepperland assíncrona, é possível concordar sobre quem liderará o ataque. Por exemplo, cada divisão envia o número de seus membros restantes e aquela que tiver mais liderará (se houver empate, a divisão Apple terá prioridade sobre a divisão Orange). Porém, quando elas devem atacar? Infelizmente, na Pepperland assíncrona, os mensageiros têm velocidade muito variável. Se a divisão Apple enviar um mensageiro com a mensagem “Atacar!”, a divisão Orange poderá não recebê-la dentro de, digamos, três horas; ou então, poderá ter recebido em cinco minutos. O problema de coordenação de ataque ainda existe se considerarmos uma Pepperland síncrona, porém as divisões conhecerão algumas restrições úteis: toda mensagem leva pelo menos *min* minutos e no máximo *max* minutos para chegar. Se a divisão que liderará o ataque enviar a mensagem “Atacar!”, ela esperará por *min* minutos e depois atacará. A outra divisão, após receber a mensagem, esperará por 1 minuto e depois atacará. É garantido que seu ataque ocorrerá após a da divisão líder, mas não mais do que (*max* – *min* + 1) minutos depois dela.

* N. de R.T.: O consenso entre partes é um problema clássico em sistemas distribuídos. Os nomes foram mantidos em inglês para honrar sua origem. Pepperland é uma cidade imaginária do desenho animado intitulado *Yellow Submarine*, protagonizado, em clima psicodélico, pelos Beatles. Em Pepperland, seus pacíficos habitantes se divertem escutando a música da banda Sgt. Peppers Lonely Hearts Club. Entretanto, lá também habitam os Blue Meanies que encolhem ao escutar o som da música e, assim, atacam Pepperland, acabando com a música e transformando todos em estátuas de pedra. É quando Lord Mayor consegue escapar e buscar ajuda, embarcando no submarino amarelo. Os Beatles entram em ação para enfrentar os Blue Meanies.

de um modelo alternativo. Um sistema distribuído assíncrono é aquele em que não existem considerações sobre:

- As velocidades de execução de processos – por exemplo, uma etapa do processo pode levar apenas um picosegundo e outra, um século; tudo que pode ser dito é que cada etapa pode demorar um tempo arbitrariamente longo.
- Os atrasos na transmissão das mensagens – por exemplo, uma mensagem do processo A para o processo B pode ser enviada em um tempo insignificante e outra pode demorar vários anos. Em outras palavras, uma mensagem pode ser recebida após um tempo arbitrariamente longo.
- As taxas de desvio do relógio – novamente, a taxa de desvio de um relógio é arbitrária.

O modelo assíncrono não faz nenhuma consideração sobre os intervalos de tempo envolvidos em qualquer tipo de execução. A Internet é perfeitamente representada por esse modelo, pois não há nenhum limite intrínseco sobre a carga no servidor ou na rede e, consequentemente, sobre quanto tempo demora, por exemplo, para transferir um arquivo usando FTP. Às vezes, uma mensagem de *e-mail* pode demorar vários dias para chegar. O quadro a seguir ilustra a dificuldade de se chegar a um acordo em um sistema distribuído assíncrono.

Porém, mesmo desconsiderando as restrições de tempo, às vezes é necessário realizar algum tipo de tratamento para o problema de demoras e atrasos de execução. Por exemplo, embora a Web nem sempre possa fornecer uma resposta específica dentro de um limite de tempo razoável, os navegadores são projetados de forma a permitir que os usuários façam outras coisas enquanto esperam. Qualquer solução válida para um sistema distribuído assíncrono também é válida para um sistema síncrono.

Muito frequentemente, os sistemas distribuídos reais são assíncronos devido à necessidade dos processos de compartilhar tempo de processamento, canais de comunicação e acesso à rede. Por exemplo, se vários processos, de características desconhecidas, compartilharem um processador, então o desempenho resultante de qualquer um deles não poderá ser garantido. Contudo, existem problemas que não podem ser resolvidos para um sistema assíncrono, mas que podem ser tratados quando alguns aspectos de tempo são usados. Um desses problemas é a necessidade de fazer com que cada elemento de um fluxo de dados multimídia seja emitido dentro de um prazo final. Para problemas como esses, exige-se um modelo síncrono.

Ordenação de eventos • Em muitos casos, estamos interessados em saber se um evento (envio ou recepção de uma mensagem) ocorreu em um processo antes, depois ou simultaneamente com outro evento em outro processo. Mesmo na ausência da noção de relógio, a execução de um sistema pode ser descrita em termos da ocorrência de eventos e de sua ordem.

Por exemplo, considere o seguinte conjunto de trocas de mensagens, entre um grupo de usuários de *e-mail*, X, Y, Z e A, em uma lista de distribuição:

1. o usuário X envia uma mensagem com o assunto *Reunião*;
2. os usuários Y e Z respondem, enviando uma mensagem com o assunto *Re: Reunião*.

Seguindo uma linha de tempo, a mensagem de X foi enviada primeiro, Y a lê e responde; Z lê a mensagem de X e a resposta de Y e envia outra resposta fazendo referência às men-

sagens de X e de Y. Contudo, devido aos diferentes atrasos envolvidos na distribuição das mensagens, elas podem ser entregues como ilustrado na Figura 2.13, e alguns usuários poderão ver essas duas mensagens na ordem errada; por exemplo, o usuário A poderia ver:

Caixa de entrada:		
Item	De	Assunto
23	Z	Re: Reunião
24	X	Reunião
25	Y	Re: Reunião

Se os relógios nos computadores de X, de Y e de Z pudessem ser sincronizados, então cada mensagem, ao ser enviada, poderia transportar a hora do relógio de seu computador local. Por exemplo, as mensagens m_1 , m_2 e m_3 transportariam os tempos t_1 , t_2 e t_3 , onde $t_1 < t_2 < t_3$. As mensagens recebidas seriam exibidas para os usuários de acordo com sua ordem temporal de emissão. Se os relógios estiverem aproximadamente sincronizados, então esses carimbos de tempo frequentemente estarão na ordem correta.

Como em um sistema distribuído os relógios não podem ser perfeitamente sincronizados, Lamport [1978] propôs um modelo de *relógio lógico*, que pode ser usado para proporcionar uma ordenação de eventos ocorridos em processos executados em diferentes computadores. O relógio lógico permite deduzir a ordem em que as mensagens devem ser apresentadas, sem apelar para os relógios físicos de cada máquina. O modelo de relógio lógico será apresentado com detalhes no Capítulo 14, mas comentaremos, aqui, como alguns aspectos da ordenação lógica podem ser aplicados ao nosso problema de ordenação de *e-mail*.

Logicamente, sabemos que uma mensagem é recebida após ser enviada; portanto, podemos expressar a ordenação lógica de pares de eventos mostrada na Figura 2.13, por exemplo, considerando apenas os eventos relativos a X e Y:

X envia m_1 antes que Y receba m_1 ; Y envia m_2 antes que X receba m_2 .

Também sabemos que as respostas são enviadas após o recebimento das mensagens; portanto, temos a seguinte ordenação lógica para Y:

Y recebe m_1 antes de enviar m_2 .

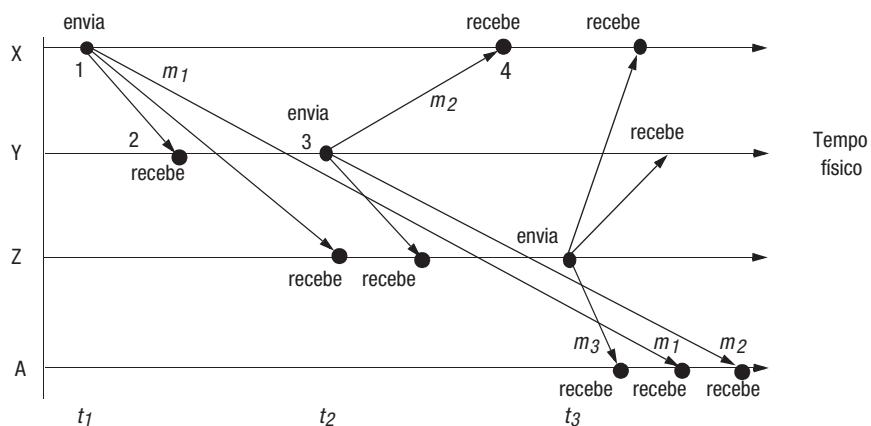


Figura 2.13 Ordenação de eventos no tempo físico.

O relógio lógico leva essa ideia mais adiante, atribuindo a cada evento um número correspondente à sua ordem lógica, de modo que os eventos posteriores tenham números mais altos do que os anteriores. Por exemplo, a Figura 2.13 mostra os números 1 a 4 para os eventos de X e Y.

2.4.2 Modelo de falhas

Em um sistema distribuído, tanto os processos como os canais de comunicação podem falhar – isto é, podem divergir do que é considerado um comportamento correto ou desejável. O modelo de falhas define como uma falha pode se manifestar em um sistema, de forma a proporcionar um entendimento dos seus efeitos e consequências. Hadzilacos e Toueg [1994] fornecem uma taxonomia que distingue as falhas de processos e as falhas de canais de comunicação. Isso é apresentado sob os títulos falhas por omissão, falhas arbitrárias e falhas de sincronização.

O modelo de falhas será usado ao longo de todo o livro. Por exemplo:

- No Capítulo 4, apresentaremos as interfaces Java para comunicações baseadas em datagrama e por fluxo (*stream*), que proporcionam diferentes graus de confiabilidade.
- O Capítulo 5 apresentará o protocolo requisição-resposta (*request-reply*), que suporta RMI. Suas características de falhas dependem tanto dos processos como dos canais de comunicação. O protocolo requisição-resposta pode ser construído sobre datagramas ou *streams*. A decisão é feita considerando aspectos de simplicidade de implementação, desempenho e confiabilidade.
- O Capítulo 17 apresentará o protocolo de confirmação (*commit*) de duas fases para transações. Ele é projetado de forma a ser concluído na presença de falhas bem-definidas de processos e canais de comunicação.

Falhas por omissão • As falhas classificadas como *falhas por omissão* se referem aos casos em que um processo ou canal de comunicação deixa de executar as ações que deveria.

Falhas por omissão de processo: a principal falha por omissão de um processo é quando ele entra em colapso, parando e não executando outro passo de seu programa. Popularmente, isso é conhecido como “dar pau” ou “pendurar”. O projeto de serviços que podem sobreviver na presença de falhas pode ser simplificado, caso se possa supor que os serviços dos quais dependem colapsam de modo limpo, isto é, os processos funcionam corretamente ou param. Outros processos podem detectar essa falha pelo fato de o processo deixar repetidamente de responder às mensagens de invocação. Entretanto, esse método de detecção de falhas é baseado no uso de *timeouts* – ou seja, considera a existência de um tempo limite para que uma determinada ação ocorra. Em um sistema assíncrono, a ocorrência de um *timeout* indica apenas que um processo não está respondendo – porém, ele pode ter entrado em colapso, estar lento ou, ainda, as mensagens podem não ter chegado.

O colapso de um processo é chamado de *parada por falha* se outros processos puderem detectar, com certeza, a ocorrência dessa situação. Em um sistema síncrono, uma parada por falha ocorre quando *timeouts* são usados para determinar que certos processos deixaram de responder a mensagens sabidamente entregues. Por exemplo, se os processos *p* e *q* estiverem programados para *q* responder a uma mensagem de *p* e, se o processo *p* não receber nenhuma resposta do processo *q* dentro de um tempo máximo (*timeout*), medido no relógio local de *p*, então o processo *p* poderá concluir que o processo *q* falhou. O quadro a seguir ilustra a dificuldade para se detectar falhas em um sistema assíncrono ou de se chegar a um acordo na presença de falhas.

Detecção de falha • No caso das divisões de Pepperland acampadas no topo das colinas (veja a página 65), suponha que os Blue Meanies tenham, afinal, força suficiente para atacar e vencer uma das divisões, enquanto estiverem acampadas – ou seja, que uma das duas divisões possa falhar. Suponha também que, enquanto não são derrotadas, as divisões regularmente enviam mensageiros para relatar seus *status*. Em um sistema assíncrono, nenhuma das duas divisões pode distinguir se a outra foi derrotada ou se o tempo para que os mensageiros cruzem o vale entre elas é simplesmente muito longo. Em uma Pepperland síncrona, uma divisão pode saber com certeza se a outra foi derrotada, pela ausência de um mensageiro regular. Entretanto, a outra divisão pode ter sido derrotada imediatamente após ter enviado o último mensageiro.

Impossibilidade de chegar a um acordo em tempo hábil na presença de falhas de comunicação • Até agora, foi suposto que os mensageiros de Pepperland sempre conseguem cruzar o vale; agora, considere que os Blue Meanies podem capturar qualquer mensageiro e impedir que ele chegue a seu destino. (Devemos supor que é impossível para os Blue Meanies fazer lavagem cerebral nos mensageiros para transmitirem a mensagem errada – os Meanies desconhecem seus traiçoeiros precursores: os generais bizantinos*.) As divisões Apple e Orange podem enviar mensagens para que ambas decidam atacar os Meanies ou que decidam se render? Infelizmente, conforme provou o teórico de Pepperland, Ringo, o Grande, nessas circunstâncias, as divisões não podem garantir a decisão correta do que fazer. Para entender como isso acontece, suponha o contrário, que as divisões executem um protocolo Pepperland de consenso: cada divisão propõe “Atacar!” ou “Render-se!” e, através de mensagens, finaliza com as divisões concordando com uma ou outra ação. Agora, considere que o mensageiro que transporta a última mensagem foi capturado pelos Blue Meanies, mas que isso, de alguma forma, não afeta a decisão final de atacar ou se render. Nesse momento, a penúltima mensagem se tornou a última. Se, sucessivamente, aplicarmos o argumento de que o último mensageiro foi capturado, chegaremos à situação em que nenhuma mensagem foi entregue. Isso mostra que não pode existir nenhum protocolo que garanta o acordo entre as divisões de Pepperland, caso os mensageiros possam ser capturados.

Falhas por omissão na comunicação: considere as primitivas de comunicação *send* e *receive*. Um processo *p* realiza um *send* inserindo a mensagem *m* em seu *buffer* de envio. O canal de comunicação transporta *m* para o *buffer* de recepção *q*. O processo *q* realiza uma operação *receive* recuperando *m* de seu *buffer* de recepção (veja a Figura 2.14). Normalmente, os *buffers* de envio e de recepção são fornecidos pelo sistema operacional.

O canal de comunicação produz uma falha por omissão quando não concretiza a transferência de uma mensagem *m* do *buffer* de envio de *p* para o *buffer* de recepção de



Figura 2.14 Processos e canais.

* N. de R.T.: Referência ao problema dos generais bizantinos, no qual as divisões devem chegar a um consenso sobre atacar ou recuar, mas há generais que são traidores.

q. Isso é conhecido como “perda de mensagens” e geralmente é causado pela falta de espaço no *buffer* de recepção, ou pelo fato de a mensagem ser descartada ao ser detectado que houve um erro durante sua transmissão (isso é feito por meio de soma de verificação sobre os dados que compõem a mensagem como, por exemplo, cálculo de CRC). Hadzilacos e Toueg [1994] se referem à perda de mensagens entre o processo remetente e o *buffer* de envio como *falhas por omissão de envio*; à perda de mensagens entre o *buffer* de recepção e o processo destino como *falhas por omissão de recepção*; e à perda de mensagens no meio de comunicação como *falhas por omissão de canal*. Na Figura 2.15, as falhas por omissão estão classificadas junto às falhas arbitrárias.

As falhas podem ser classificadas de acordo com sua gravidade. Por enquanto, todas as falhas descritas até aqui são consideradas *benignas*. A maioria das falhas nos sistemas distribuídos é benigna, as quais incluem as falhas por omissão, as de sincronização e as de desempenho.

Falhas arbitrárias • O termo falha *arbitrária*, ou *bizantina*, é usado para descrever a pior semântica de falha possível na qual qualquer tipo de erro pode ocorrer. Por exemplo, um processo pode atribuir valores incorretos a seus dados ou retornar um valor errado em resposta a uma invocação.

Uma falha arbitrária de um processo é aquela em que ele omite arbitrariamente passos desejados do processamento ou efetua processamento indesejado. Portanto, as falhas arbitrárias não podem ser detectadas verificando-se se o processo responde às invocações, pois ele poderia omitir arbitrariamente a resposta.

Os canais de comunicação podem sofrer falhas arbitrárias; por exemplo, o conteúdo da mensagem pode ser corrompido, mensagens inexistentes podem ser enviadas ou mensagens reais podem ser entregues mais de uma vez. As falhas arbitrárias dos canais de comunicação são raras, pois o *software* de comunicação é capaz de reconhecê-las e rejeitar as mensagens com problemas. Por exemplo, somas de verificação são usadas para detectar mensagens corrompidas e números de sequência de mensagem podem ser usados para detectar mensagens inexistentes ou duplicadas.

Classe da falha	Afeta	Descrição
Parada por falha	Processo	O processo pára e permanece parado. Outros processos podem detectar esse estado.
Colapso	Processo	O processo pára e permanece parado. Outros processos podem não detectar esse estado.
Omissão	Canal	Uma mensagem inserida em um <i>buffer</i> de envio nunca chega no <i>buffer</i> de recepção do destinatário.
Omissão de envio	Processo	Um processo conclui um envio, mas a mensagem não é colocada em seu <i>buffer</i> de envio.
Omissão de recepção	Processo	Uma mensagem é colocada no <i>buffer</i> de recepção de um processo, mas esse processo não a recebe efetivamente.
Arbitraria (bizantina)	Processo ou canal	O processo/canal exibe comportamento arbitrário: ele pode enviar/transmitir mensagens arbitrárias em qualquer momento, cometer omissões; um processo pode parar ou realizar uma ação incorreta.

Figura 2.15 Falhas por omissão e falhas arbitrárias.

Falhas de temporização • As falhas de temporização são aplicáveis aos sistemas distribuídos síncronos em que limites são estabelecidos para o tempo de execução do processo, para o tempo de entrega de mensagens e para a taxa de desvio do relógio. As falhas de temporização estão listadas na Figura 2.16. Qualquer uma dessas falhas pode resultar em indisponibilidade de respostas para os clientes dentro de um intervalo de tempo pre-determinado.

Em um sistema distribuído assíncrono, um servidor sobre-carregado pode responder muito lentamente, mas não podemos dizer que ele apresenta uma falha de temporização, pois nenhuma garantia foi oferecida.

Os sistemas operacionais de tempo real são projetados visando a garantias de cumprimento de prazos, mas seu projeto é mais complexo e pode exigir *hardware* redundante. A maioria dos sistemas operacionais de propósito geral, como o UNIX, não precisa satisfazer restrições de tempo real.

A temporização é particularmente relevante para aplicações multimídia, com canais de áudio e vídeo. As informações de vídeo podem exigir a transferência de um volume de dados muito grande. Distribuir tais informações sem falhas de temporização pode impor exigências muito especiais sobre o sistema operacional e sobre o sistema de comunicação.

Mascaramento de falhas • Cada componente em um sistema distribuído geralmente é construído a partir de um conjunto de outros componentes. É possível construir serviços confiáveis a partir de componentes que exibem falhas. Por exemplo, vários servidores que contêm réplicas dos dados podem continuar a fornecer um serviço quando um deles apresenta um defeito. O conhecimento das características da falha de um componente pode permitir que um novo serviço seja projetado de forma a mascarar a falha dos componentes dos quais ele depende. Um serviço *mascara* uma falha ocultando-a completamente ou convertendo-a em um tipo de falha mais aceitável. Como um exemplo desta última opção, somas de verificação são usadas para mascarar mensagens corrompidas – convertendo uma falha arbitrária em falha por omissão. Nos Capítulos 3 e 4, veremos que as falhas por omissão podem ser ocultas usando-se um protocolo que retransmite as mensagens que não chegam ao seu destino. O Capítulo 18 apresentará o mascaramento feito por meio da replicação. Até o colapso de um processo pode ser mascarado – criando-se um novo processo e restaurando, a partir de informações armazenadas em disco, o estado da memória de seu predecessor.

Confiabilidade da comunicação de um para um • Embora um canal de comunicação possa exibir as falhas por omissão descritas anteriormente, é possível usá-lo para construir um serviço de comunicação que mascare algumas dessas falhas.

Classe da falha	Afeta	Descrição
Relógio	Processo	O relógio local do processo ultrapassa os limites de sua taxa de desvio em relação ao tempo físico.
Desempenho	Processo	O processo ultrapassa os limites do intervalo de tempo entre duas etapas.
Desempenho	Canal	A transmissão de uma mensagem demora mais do que o limite definido.

Figura 2.16 Falhas de temporização.

O termo *comunicação confiável* é definido em termos de validade e integridade, como segue:

Validade: qualquer mensagem do *buffer* de envio é entregue ao *buffer* de recepção de seu destino, independentemente do tempo necessário para tal;

Integridade: a mensagem recebida é idêntica à enviada e nenhuma mensagem é entregue duas vezes.

As ameaças à integridade vêm de duas fontes independentes:

- Qualquer protocolo que retransmita mensagens, mas não rejeite uma mensagem que foi entregue duas vezes. Os protocolos podem incluir números de sequência nas mensagens para detectar aquelas que são entregues duplicadas.
- Usuários mal-intencionados que podem injetar mensagens espúrias, reproduzir mensagens antigas ou falsificar mensagens. Medidas de segurança podem ser tomadas para manter a propriedade da integridade diante de tais ataques.

2.4.3 Modelo de segurança

No Capítulo 1, identificamos o compartilhamento de recursos como um fator motivador para os sistemas distribuídos e, então, na Seção 2.3, descrevemos sua arquitetura de sistema em termos de processos, possivelmente encapsulando abstrações de nível mais alto, como objetos, componentes ou serviços, e fornecendo acesso a eles por meio de interações com outros processos. Esse princípio de funcionamento fornece a base de nosso modelo de segurança:

a segurança de um sistema distribuído pode ser obtida tornando seguros os processos e os canais usados por suas interações e protegendo contra acesso não autorizado os objetos que encapsulam.

A proteção é descrita em termos de objetos, embora os conceitos se apliquem igualmente bem a qualquer tipo de recursos.

Proteção de objetos • A Figura 2.17 mostra um servidor que gerencia um conjunto de objetos para alguns usuários. Os usuários podem executar programas clientes que enviam invocações para o servidor a fim de realizar operações sobre os objetos. O servidor executa a operação especificada em cada invocação e envia o resultado para o cliente.

Os objetos são usados de diversas formas, por diferentes usuários. Por exemplo, alguns objetos podem conter dados privativos de um usuário, como sua caixa de correio, e outros podem conter dados compartilhados, como suas páginas Web. Para dar suporte

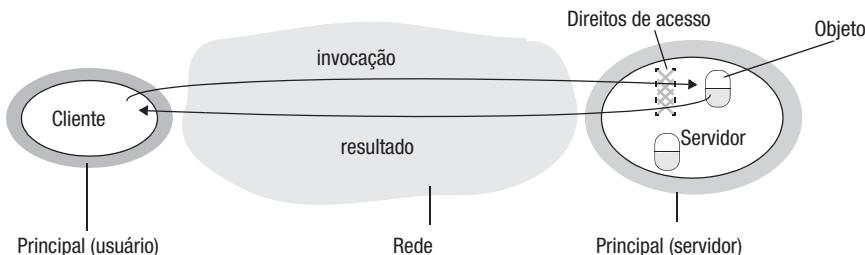


Figura 2.17 Objetos e principais.

a isso, *direitos de acesso* especificam quem pode executar determinadas operações sobre um objeto – por exemplo, quem pode ler ou escrever seu estado.

Dessa forma, os usuários devem ser incluídos em nosso modelo de segurança como os beneficiários dos direitos de acesso. Fazemos isso associando a cada invocação, e a cada resultado, a entidade que a executa. Tal entidade é chamada de *principal*. Um principal pode ser um usuário ou um processo. Em nossa ilustração, a invocação vem de um usuário e o resultado, de um servidor.

O servidor é responsável por verificar a identidade do principal que está por trás de cada invocação e conferir se ele tem direitos de acesso suficientes para efetuar a operação solicitada em determinado objeto, rejeitando as que ele não pode efetuar. O cliente pode verificar a identidade do principal que está por trás do servidor, para garantir que o resultado seja realmente enviado por esse servidor.

Tornando processos e suas interações seguros • Os processos interagem enviando mensagens. As mensagens ficam expostas a ataques, porque o acesso à rede e ao serviço de comunicação é livre para permitir que quaisquer dois processos interajam. Servidores e processos *peer-to-peer* publicam suas interfaces, permitindo que invocações sejam enviadas a eles por qualquer outro processo.

Frequentemente, os sistemas distribuídos são implantados e usados em tarefas que provavelmente estarão sujeitas a ataques externos realizados por usuários mal-intencionados. Isso é especialmente verdade para aplicativos que manipulam transações financeiras, informações confidenciais ou secretas, ou qualquer outro tipo de informação cujo segredo ou integridade seja crucial. A integridade é ameaçada por violações de segurança, assim como por falhas na comunicação. Portanto, sabemos que existem prováveis ameaças aos processos que compõem os aplicativos e as mensagens que trafegam entre eles. No entanto, como podemos analisar essas ameaças para identificá-las e anulá-las? A discussão a seguir apresenta um modelo para a análise de ameaças à segurança.

O invasor • Para modelar as ameaças à segurança, postulamos um invasor (também conhecido como atacante) capaz de enviar qualquer mensagem para qualquer processo e ler ou copiar qualquer mensagem entre dois processos, como se vê na Figura 2.18. Tais ataques podem ser realizados usando-se simplesmente um computador conectado a uma rede para executar um programa que lê as mensagens endereçadas para outros computadores da rede, ou por um programa que gera mensagens que façam falsos pedidos para serviços e deem a entender que sejam provenientes de usuários autorizados. O ataque pode vir de um computador legitimamente conectado à rede ou de um que esteja conectado de maneira não autorizada.

As ameaças de um atacante em potencial são discutidas sob os títulos *ameaças aos processos*, *ameaças aos canais de comunicação* e *negação de serviço*.

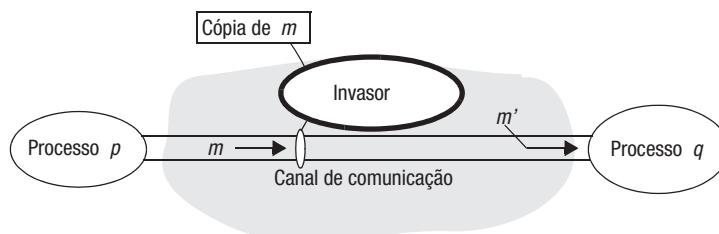


Figura 2.18 O invasor (atacante).

Ameaças aos processos: um processo projetado para tratar pedidos pode receber uma mensagem de qualquer outro processo no sistema distribuído e não ser capaz de determinar com certeza a identidade do remetente. Os protocolos de comunicação, como o IP, incluem o endereço do computador de origem em cada mensagem, mas não é difícil para um atacante gerar uma mensagem com um endereço de origem falsificado. Essa falta de reconhecimento confiável da origem de uma mensagem é, conforme explicado a seguir, uma ameaça ao funcionamento correto tanto de servidores como de clientes:

Servidores: como um servidor pode receber pedidos de muitos clientes diferentes, ele não pode necessariamente determinar a identidade do principal que está por trás de uma invocação em particular. Mesmo que um servidor exija a inclusão da identidade do principal em cada invocação, um atacante poderia gerá-la com uma identidade falsa. Sem o reconhecimento garantido da identidade do remetente, um servidor não pode saber se deve executar a operação ou rejeitá-la. Por exemplo, um servidor de correio eletrônico que recebe de um usuário uma solicitação de leitura de mensagens de uma caixa de correio eletrônico em particular, pode não saber se esse usuário está autorizado a fazer isso ou se é uma solicitação indevida.

Clientes: quando um cliente recebe o resultado de uma invocação feita a um servidor, ele não consegue identificar se a origem da mensagem com o resultado é proveniente do servidor desejado ou de um invasor, talvez fazendo *spoofing* desse servidor. O *spoofing* é, na prática, o roubo de identidade. Assim, um cliente poderia receber um resultado não relacionado à invocação original como, por exemplo, uma mensagem de correio eletrônico falsa (que não está na caixa de correio do usuário).

Ameaças aos canais de comunicação: um invasor pode copiar, alterar ou injetar mensagens quando elas trafegam pela rede e em seus sistemas intermediários (roteadores, por exemplo). Tais ataques representam uma ameaça à privacidade e à integridade das informações quando elas trafegam pela rede e à própria integridade do sistema. Por exemplo, uma mensagem com resultado contendo um correio eletrônico de um usuário poderia ser revelada a outro, ou ser alterada para dizer algo totalmente diferente.

Outra forma de ataque é a tentativa de salvar cópias de mensagens e reproduzi-las posteriormente, tornando possível reutilizar a mesma mensagem repetidamente. Por exemplo, alguém poderia tirar proveito, reenviando uma mensagem de invocação, solicitando uma transferência de um valor em dinheiro de uma conta bancária para outra.

Todas essas ameaças podem ser anuladas com o uso de *canais de comunicação seguros*, que estão descritos a seguir e são baseados em criptografia e autenticação.

Anulando ameaças à segurança • Apresentamos aqui as principais técnicas nas quais os sistemas seguros são baseados. O Capítulo 11 discutirá com mais detalhes o projeto e a implementação de sistemas distribuídos seguros.

Criptografia e segredos compartilhados: suponha que dois processos (por exemplo, um cliente e um servidor) compartilhem um segredo; isto é, ambos conhecem o segredo, mas nenhum outro processo no sistema distribuído sabe dele. Então, se uma mensagem trocada por esses dois processos incluir informações que provem o conhecimento do segredo compartilhado por parte do remetente, o destinatário saberá com certeza que o remetente foi o outro processo do par. É claro que se deve tomar os cuidados necessários para garantir que o segredo compartilhado não seja revelado a um invasor.

Criptografia é a ciência de manter as mensagens seguras, e *cifragem* é o processo de embaralhar uma mensagem de maneira a ocultar seu conteúdo. A criptografia moder-

na é baseada em algoritmos que utilizam chaves secretas – números grandes e difíceis de adivinhar – para transformar os dados de uma maneira que só possam ser revertidos com o conhecimento da chave de *decifração* correspondente.

Autenticação: o uso de segredos compartilhados e da criptografia fornece a base para a *autenticação* de mensagens – provar as identidades de seus remetentes. A técnica de autenticação básica é incluir em uma mensagem uma parte cifrada que possua conteúdo suficiente para garantir sua autenticidade. A autenticação de um pedido de leitura de um trecho de um arquivo enviado a um servidor de arquivos poderia, por exemplo, incluir uma representação da identidade do principal que está fazendo a solicitação, a identificação do arquivo e a data e hora do pedido, tudo cifrado com uma chave secreta compartilhada entre o servidor de arquivos e o processo solicitante. O servidor decifraria o pedido e verificaria se as informações correspondem realmente ao pedido.

Canais seguros: criptografia e autenticação são usadas para construir canais seguros como uma camada de serviço a mais sobre os serviços de comunicação já existentes. Um canal seguro é um canal de comunicação conectando dois processos, cada um atuando em nome de um principal, como se vê na Figura 2.19. Um canal seguro tem as seguintes propriedades:

- Cada um dos processos conhece com certeza a identidade do principal em nome de quem o outro processo está executando. Portanto, se um cliente e um servidor se comunicam por meio de um canal seguro, o servidor conhece a identidade do principal que está por trás das invocações e pode verificar seus direitos de acesso, antes de executar uma operação. Isso permite que o servidor proteja corretamente seus objetos e que o cliente tenha certeza de que está recebendo resultados de um servidor *fidedigno*.
- Um canal seguro garante a privacidade e a integridade (proteção contra falsificação) dos dados transmitidos por ele.
- Cada mensagem inclui uma indicação de relógio lógico ou físico para impedir que as mensagens sejam reproduzidas ou reordenadas.

A construção de canais seguros será discutida em detalhes no Capítulo 11. Os canais seguros têm se tornado uma importante ferramenta prática para proteger o comércio eletrônico e para a proteção de comunicações em geral. As redes virtuais privativas (VPNs, Virtual Private Networks, discutidas no Capítulo 3) e o protocolo SSL (Secure Sockets Layer) (discutido no Capítulo 11) são exemplos.

Outras ameaças possíveis • A Seção 1.5.3 apresentou, muito sucintamente, duas ameaças à segurança – ataques de negação de serviço e utilização de código móvel. Reiteraremos essas ameaças como possíveis oportunidades para o invasor romper as atividades dos processos:

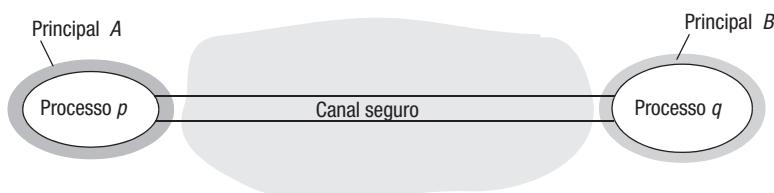


Figura 2.19 Canais seguros.

Negação de serviço: esta é uma forma de ataque na qual o atacante interfere nas atividades dos usuários autorizados, fazendo inúmeras invocações sem sentido em serviços, ou transmitindo mensagens incessantemente em uma rede para gerar uma sobrecarga dos recursos físicos (capacidade de processamento do servidor, largura de banda da rede, etc.). Tais ataques normalmente são feitos com a intenção de retardar ou impedir as invocações válidas de outros usuários. Por exemplo, a operação de trancas eletrônicas de portas em um prédio poderia ser desativada por um ataque que saturasse o computador que controla as trancas com pedidos inválidos.

Código móvel: o código móvel levanta novos e interessantes problemas de segurança para qualquer processo que receba e execute código proveniente de outro lugar, como o anexo de correio eletrônico mencionado na Seção 1.5.3. Esse código pode desempenhar facilmente o papel de cavalo de Troia, dando a entender que vai cumprir um propósito inocente, mas que na verdade inclui código que acessa ou modifica recursos legitimamente disponíveis para o usuário que o executa. Os métodos pelos quais tais ataques podem ser realizados são muitos e variados e, para evitá-los, o ambiente que recebe tais códigos deve ser construído com muito cuidado. Muitos desses problemas foram resolvidos com a utilização de Java e em outros sistemas de código móvel, mas a história recente desse assunto inclui algumas vulnerabilidades embarracosas. Isso ilustra bem a necessidade de uma análise rigorosa no projeto de todos os sistemas seguros.

O uso dos modelos de segurança • Pode-se pensar que a obtenção de segurança em sistemas distribuídos seria uma questão simples, envolvendo o controle do acesso a objetos de acordo com direitos de acesso predefinidos e com o uso de canais seguros para comunicação. Infelizmente, muitas vezes esse não é o caso. O uso de técnicas de segurança como a criptografia e o controle de acesso acarreta custos de processamento e de gerenciamento significativos. O modelo de segurança delineado anteriormente fornece a base para a análise e o projeto de sistemas seguros, em que esses custos são mantidos em um mínimo. Entretanto, as ameaças a um sistema distribuído surgem em muitos pontos e é necessária uma análise cuidadosa das ameaças que podem surgir de todas as fontes possíveis no ambiente de rede, no ambiente físico e no ambiente humano do sistema. Essa análise envolve a construção de um *modelo de ameaças*, listando todas as formas de ataque a que o sistema está exposto e uma avaliação dos riscos e consequências de cada um. A eficácia e o custo das técnicas de segurança necessárias podem, então, ser ponderadas em relação às ameaças.

2.5 Resumo

Conforme ilustrado na Seção 2.2, os sistemas distribuídos estão cada vez mais complexos em termos de suas características físicas subjacentes; por exemplo, em termos da escala dos sistemas, do nível de heterogeneidade inerente a tais sistemas e das demandas reais em fornecer soluções de ponta a ponta em termos de propriedades, como a segurança. Isso aumenta cada vez mais a importância de se entender e considerar os sistemas distribuídos em termos de modelos. Este capítulo seguiu a consideração sobre os modelos físicos subjacentes com um exame aprofundado dos modelos arquitetônicos e fundamentais que formam a base dos sistemas distribuídos.

O capítulo apresentou uma estratégia para se descrever os sistemas distribuídos em termos de um modelo arquitetônico abrangente que dá sentido a esse espaço de projeto, examinando as principais questões sobre o que é a comunicação e como esses siste-

mas se comunicam, complementada com a consideração das funções desempenhadas pelos elemento, junto às estratégias de posicionamento apropriadas, dada a infraestrutura distribuída física. Também apresentou a principal função dos padrões arquitetônicos para permitir a construção de projetos mais complexos a partir dos elementos básicos subjacentes (como o modelo cliente-servidor examinado anteriormente) e destacou os principais estilos de soluções de *middleware* auxiliares, incluindo soluções baseadas em objetos distribuídos, componentes, serviços Web e eventos distribuídos.

Em termos de modelos arquitetônicos, o modelo cliente-servidor predomina – a Web e outros serviços de Internet, como FTP, news e correio eletrônico, assim como serviços Web e o DNS, são baseados nesse modelo, sem mencionar outros serviços locais. Serviços como o DNS, que têm grande número de usuários e gerenciam muitas informações, são baseados em múltiplos servidores e utilizam o particionamento de dados e a replicação para melhorar a disponibilidade e a tolerância a falhas. O uso de cache por clientes e servidores *proxies* é amplamente empregado para melhorar o desempenho de um serviço.

Contudo, atualmente há uma ampla variedade de estratégias para modelar sistemas distribuídos, incluindo filosofias alternativas, como a computação *peer-to-peer* e o suporte para abstrações mais voltadas para o problema, como objetos, componentes ou serviços.

O modelo arquitetônico é complementado por modelos fundamentais, os quais ajudam a refletir a respeito das propriedades do sistema distribuído, em termos, por exemplo, de desempenho, confiabilidade e segurança. Em particular, a presentamos os modelos de interação, falha e segurança. Eles identificam as características comuns dos componentes básicos a partir dos quais os sistemas distribuídos são construídos. O modelo de interação se preocupa com o desempenho dos processos dos canais de comunicação e com a ausência de um relógio global. Ele identifica um sistema síncrono como aquele em que podem ser impostos limites conhecidos para o tempo de execução de um processo, para o tempo de entrega de mensagens e para o desvio do relógio. Ele identifica um sistema assíncrono como aquele em que nenhum limite pode ser imposto para o tempo de execução de um processo, para o tempo de entrega de mensagens e para o desvio do relógio. O comportamento da Internet segue esse modelo.

O modelo de falha classifica as falhas de processos e dos canais de comunicação básicos em um sistema distribuído. O mascaramento é uma técnica por meio da qual um serviço mais confiável é construído a partir de outro menos confiável, escondendo algumas das falhas que ele exibe. Em particular, um serviço de comunicação confiável pode ser construído a partir de um canal de comunicação básico por meio do mascaramento de suas falhas. Por exemplo, suas falhas por omissão podem ser mascaradas pela retransmissão das mensagens perdidas. A integridade é uma propriedade da comunicação confiável – ela exige que uma mensagem recebida seja idêntica àquela que foi enviada e que nenhuma mensagem seja enviada duas vezes. A validade é outra propriedade – ela exige que toda mensagem colocada em um *buffer* de envio seja entregue no *buffer* de recepção de um destinatário.

O modelo de segurança identifica as possíveis ameaças aos processos e canais de comunicação em um sistema distribuído aberto. Algumas dessas ameaças se relacionam com a integridade: usuários mal-intencionados podem falsificar mensagens ou reproduzi-las. Outras ameaçam sua privacidade. Outro problema de segurança é a autenticação do principal (usuário ou servidor) em nome de quem uma mensagem foi enviada. Os canais seguros usam técnicas de criptografia para garantir a integridade, a privacidade das mensagens e para autenticar mutuamente os pares de principais que estejam se comunicando.

Exercícios

- 2.1 Dê três exemplos específicos e contrastantes dos níveis de heterogeneidade cada vez maiores experimentados nos sistemas distribuídos atuais, conforme definido na Seção 2.2. *página 39*
- 2.2 Quais problemas você antevê no acoplamento direto entre entidades que se comunicam, que está implícito nas estratégias de invocação remota? Consequentemente, quais vantagens você prevê a partir de um nível de desacoplamento, conforme o oferecido pelo não acoplamento espacial e temporal? Nota: talvez você queira rever sua resposta depois de ler os Capítulos 5 e 6. *página 43*
- 2.3 Descreva e ilustre a arquitetura cliente-servidor de um ou mais aplicativos de Internet importantes (por exemplo, Web, correio eletrônico ou news). *página 46*
- 2.4 Para os aplicativos discutidos no Exercício 2.1, quais estratégias de posicionamento são empregadas na implementação dos serviços associados? *página 48*
- 2.5 Um mecanismo de busca é um servidor Web que responde aos pedidos do cliente para pesquisar em seus índices armazenados e (concomitantemente) executa várias tarefas de *Web crawling* para construir e atualizar esses índices. Quais são os requisitos de sincronização entre essas atividades concomitantes? *página 46*
- 2.6 Frequentemente, os computadores usados nos sistemas *peer-to-peer* são computadores *desktop* dos escritórios ou das casas dos usuários. Quais são as implicações disso na disponibilidade e na segurança dos objetos de dados compartilhados que eles contêm e até que ponto qualquer vulnerabilidade pode ser superada por meio da replicação? *páginas 47, 48*
- 2.7 Liste os tipos de recurso local vulneráveis a um ataque de um programa não confiável, cujo *download* é feito de um *site* remoto e que é executado em um computador local. *página 49*
- 2.8 Dê exemplos de aplicações em que o uso de código móvel seja vantajoso. *página 49*
- 2.9 Considere uma empresa de aluguel de carros hipotética e esboce uma solução de três camadas físicas para seu serviço distribuído de aluguel de carros. Use sua resposta para ilustrar vantagens e desvantagens de uma solução de três camadas físicas, considerando problemas como desempenho, mudança de escala, tratamento de falhas e manutenção do *software* com o passar do tempo. *página 53*
- 2.10 Dê um exemplo concreto do dilema apresentado pelo princípio fim-a-fim de Saltzer, no contexto do fornecimento de suporte de *middleware* para aplicativos distribuídos (talvez você queira enfocar um aspecto do fornecimento de sistemas distribuídos confiáveis, por exemplo, relacionado à tolerância a falhas ou à segurança). *página 60*
- 2.11 Considere um servidor simples que executa pedidos do cliente sem acessar outros servidores. Explique por que geralmente não é possível estabelecer um limite para o tempo gasto por tal servidor para responder ao pedido de um cliente. O que precisaria ser feito para tornar o servidor capaz de executar pedidos dentro de um tempo limitado? Essa é uma opção prática? *página 62*
- 2.12 Para cada um dos fatores que contribuem para o tempo gasto na transmissão de uma mensagem entre dois processos por um canal de comunicação, cite medidas necessárias para estabelecer um limite para sua contribuição no tempo total. Por que essas medidas não são tomadas nos sistemas distribuídos de propósito geral atuais? *página 63*
- 2.13 O serviço Network Time Protocol pode ser usado para sincronizar relógios de computador. Explique por que, mesmo com esse serviço, nenhum limite garantido é dado para a diferença entre dois relógios. *página 64*

-
- 2.14 Considere dois serviços de comunicação para uso em sistemas distribuídos assíncronos. No serviço A, as mensagens podem ser perdidas, duplicadas ou retardadas, e somas de verificação se aplicam apenas aos cabeçalhos. No serviço B, as mensagens podem ser perdidas, retardadas ou entregues rápido demais para o destinatário manipulá-las, mas sempre chegam com o conteúdo correto. Descreva as classes de falha exibidas para cada serviço. Classifique suas falhas de acordo com seu efeito sobre as propriedades de validade e integridade. O serviço B pode ser descrito como um serviço de comunicação confiável? *páginas 67, 71*
- 2.15 Considere dois processos, X e Y, que utilizam o serviço de comunicação B do Exercício 2.14 para se comunicar entre si. Suponha que X seja um cliente e que Y seja um servidor e que uma *invocação* consiste em uma mensagem de requisição de X para Y, seguida de Y executando a requisição, seguida de uma mensagem de resposta de Y para X. Descreva as classes de falha que podem ser exibidas por uma invocação. *página 67*
- 2.16 Suponha que uma leitura de disco possa, às vezes, ler valores diferentes dos gravados. Cite os tipos de falha exibidos por uma leitura de disco. Sugira como essa falha pode ser mascarada para produzir uma forma de falha benigna diferente. Agora, sugira como se faz para mascarar a falha benigna. *página 71*
- 2.17 Defina a propriedade de integridade da comunicação confiável e liste todas as possíveis ameaças à integridade de usuários e de componentes do sistema. Quais medidas podem ser tomadas para garantir a propriedade de integridade diante de cada uma dessas fontes de ameaças? *páginas 71, 74*
- 2.18 Descreva as possíveis ocorrências de cada um dos principais tipos de ameaça à segurança (ameaças aos processos, ameaças aos canais de comunicação, negação de serviço) que poderiam ocorrer na Internet. *página 73*

3

Redes de Computadores e Interligação em Rede

- 3.1 Introdução
- 3.2 Tipos de rede
- 3.3 Conceitos básicos de redes
- 3.4 Protocolos Internet
- 3.5 Estudos de caso: Ethernet, WiFi e Bluetooth
- 3.6 Resumo

Os sistemas distribuídos utilizam redes locais, redes de longa distância e redes interligadas para comunicação. O desempenho, a confiabilidade, a escalabilidade, a mobilidade e a qualidade das características do serviço das redes subjacentes afetam o comportamento dos sistemas distribuídos e, assim, têm impacto sobre seu projeto. Mudanças nos requisitos do usuário têm resultado no surgimento de redes sem fio e de redes de alto desempenho com garantia da qualidade do serviço.

Os princípios nos quais as redes de computadores são baseados incluem organização de protocolos em camadas, comutação de pacotes, roteamento e fluxo de dados (*streaming*). As técnicas de interligação em rede possibilitam a integração de redes heterogêneas. A Internet é o maior exemplo; seus protocolos são utilizados quase que universalmente em sistemas distribuídos. Os esquemas de endereçamento e de roteamento usados na Internet têm suportado o impacto de seu enorme crescimento. Atualmente, eles estão sendo revisados para acomodar um crescimento futuro e atender aos novos requisitos de aplicativos quanto a mobilidade, segurança e qualidade do serviço.

O projeto de tecnologias de rede é ilustrado em três estudos de caso: Ethernet, IEEE 802.11 (WiFi) e Bluetooth.

3.1 Introdução

As redes de comunicação usadas em sistemas distribuídos são construídas a partir de uma variedade de *mídias de transmissão*, que incluem cabos de cobre, fibras ópticas e comunicação sem fio; *dispositivos de hardware*, que englobam roteadores, comutadores (*switches*), pontes, hubs, repetidores e interfaces de rede; e *componentes de software*, como pilhas de protocolo, rotinas de tratamento de comunicação e *drivers* de dispositivos. A funcionalidade resultante e o desempenho disponível para o sistema distribuído e para os programas aplicativos são afetados por tudo isso. Vamos nos referir ao conjunto de componentes de *hardware* e *software* que fornecem os recursos de comunicação para um sistema distribuído como *subsistema de comunicação*. Os computadores e outros dispositivos que utilizam a rede para propósitos de comunicação são referidos como *hosts*. O termo *nó* é usado para se referir a qualquer computador ou dispositivo de comunicação ligado à rede.

A Internet é um subsistema de comunicação único que fornece comunicação entre todos os *hosts* conectados a ela. Ela é construída a partir de muitas *sub-redes*. Uma sub-rede é uma unidade de roteamento (mecanismo de encaminhamento de dados de uma parte da Internet a outra); ela é composta por um conjunto de nós de uma mesma rede física. A infraestrutura da Internet inclui uma arquitetura e componentes de *hardware* e *software* que integram diversas sub-redes em um único serviço de comunicação de dados.

O projeto de um subsistema de comunicação é fortemente influenciado pelas características dos sistemas operacionais empregados pelos computadores que compõem o sistema distribuído, assim como pelas redes que os interligam. Neste capítulo, consideraremos o impacto das tecnologias de rede sobre o subsistema de comunicação; os problemas relacionados aos sistemas operacionais serão discutidos no Capítulo 7.

Este capítulo se destina a fornecer um panorama introdutório sobre redes de computadores, com referência aos requisitos de comunicação dos sistemas distribuídos. Os leitores que não estiverem familiarizados com redes de computadores devem considerá-lo como uma base para o restante do livro, enquanto aqueles que já estão acostumados, verão que este capítulo oferece um resumo ampliado dos principais aspectos de redes de computadores particularmente relevantes para os sistemas distribuídos.

As redes de computadores foram concebidas logo após a invenção dos computadores. A base teórica da comutação de pacotes foi apresentada em um artigo de Leonard Kleinrock [1961]. Em 1962, J.C.R. Licklider e W. Clark, que participaram do desenvolvimento do primeiro sistema de tempo compartilhado, no MIT, no início dos anos 60, publicaram um artigo discutindo o potencial da computação interativa e das redes de longa distância que previram a Internet em vários aspectos [DEC 1990]. Em 1964, Paul Baran produziu o esboço de um projeto prático para redes de longa distância confiáveis e eficientes [Baran 1964]. Mais material e *links* sobre a história das redes de computadores e da Internet podem ser encontrados nas seguintes fontes: [www.isoc.org, Comer 2007, Kurose e Ross 2007].

No restante desta seção, discutiremos os requisitos de comunicação dos sistemas distribuídos. Forneceremos um panorama dos tipos de rede na Seção 3.2 e uma introdução aos princípios de interligação em rede na Seção 3.3. A Seção 3.4 trata especificamente da Internet. O capítulo termina com estudos de caso sobre as tecnologias de interligação em rede Ethernet, IEEE 802.11 (WiFi) e Bluetooth, na Seção 3.5.

3.1.1 Problemas de interligação em rede para sistemas distribuídos

As primeiras redes de computadores foram projetadas para atender a alguns requisitos de aplicações em rede relativamente simples, como a transferência de arquivos, *login* remoto, correio eletrônico e *newsgroups*. O desenvolvimento subsequente de sistemas distribuídos, com suporte para programas aplicativos distribuídos, acesso a arquivos compartilhados e outros recursos, estabeleceu um padrão de desempenho mais alto para atender às necessidades das aplicações interativas.

Mais recentemente, acompanhando o crescimento, a comercialização e o emprego de novos usos da Internet, surgiram requisitos de confiabilidade, escalabilidade, mobilidade, segurança e qualidade de serviço mais rigorosos. Nesta seção, definimos e descrevemos a natureza de cada um desses requisitos.

Desempenho • Os parâmetros de desempenho de rede que têm principal interesse para nossos propósitos são aqueles que afetam a velocidade com que as mensagens podem ser transferidas entre dois computadores interligados, são eles: latência e taxa de transferência de dados ponto a ponto.

Latência é o tempo decorrido após uma operação de envio ser executada e antes que os dados comecem a chegar em seu destino. Ela pode ser medida como o tempo necessário para transferir uma mensagem vazia. Aqui, estamos considerando apenas a latência da rede, que forma uma parte da latência de processo a processo, definida na Seção 2.4.1.

A *taxa de transferência de dados* é a velocidade com que os dados podem ser transferidos entre dois computadores em uma rede, uma vez que a transmissão tenha começado. É normalmente medida em bits por segundo (bps).

Como resultado dessas definições, o tempo exigido para que uma rede transfira uma mensagem contendo uma certa *largura* de bits entre dois computadores é:

$$\text{Tempo de transmissão da mensagem} = \text{latência} + \frac{\text{largura}}{\text{taxa de transferência}}$$

A equação acima é válida para mensagens cuja largura não ultrapasse o limite máximo determinado pela tecnologia de rede empregada para seu envio. Mensagens maiores precisam ser segmentadas (ou fragmentadas), e o tempo de transmissão é a soma dos tempos dos segmentos (fragmentos).

A taxa de transferência de uma rede é determinada principalmente por suas características físicas, enquanto a latência é determinada principalmente pelas sobrecargas do *software*, pelos atrasos de roteamento e por um fator estatístico dependente da carga da rede, proveniente do conflito de acesso aos canais de transmissão. Em sistemas distribuídos, é comum a transferência de muitas mensagens de pequeno tamanho entre processos; portanto, na determinação do desempenho, a latência frequentemente assume uma importância igual ou maior que a taxa de transferência.

A *largura de banda* de uma rede é uma medida de seu rendimento, isto é, o volume total de tráfego que pode ser transferido na rede em um determinado período de tempo. Em muitas tecnologias de rede local, como a Ethernet, a capacidade de transmissão máxima da rede é disponível para cada transmissão e a largura de banda do sistema é igual à taxa de transferência de dados. Contudo, na maioria das redes de longa distância, as mensagens podem ser transferidas simultaneamente em diferentes canais, e a largura de banda total do sistema não apresenta relacionamento direto com a taxa de transferência. O desempenho das redes se deteriora em condições de sobrecarga – quando existem muitas mensagens

ao mesmo tempo na rede. O efeito preciso da sobrecarga sobre a latência, sobre a taxa de transferência de dados e sobre a largura de banda depende muito da tecnologia da rede.

Agora, considere o desempenho de uma comunicação cliente-servidor. O tempo necessário para a transmissão de uma mensagem de requisição (curta) e para o recebimento de sua resposta, também curta, em uma rede local pouco carregada (incluindo as cargas de processamento) é de cerca de meio milissegundo. Isso deve ser comparado com o tempo inferior a um microsegundo exigido para uma aplicação invocar uma operação sobre um objeto armazenado na memória local. Assim, a despeito dos avanços no desempenho das redes, o tempo exigido para acessar recursos compartilhados em uma rede local permanece cerca de mil vezes maior do que o tempo exigido para acessar recursos armazenados em memória local. Porém, a latência e a largura de banda da rede frequentemente superam o desempenho do disco rígido; o acesso por rede a um servidor Web, ou a um servidor de arquivos local que mantém em cache uma grande quantidade de arquivos frequentemente utilizados, pode equivaler, ou superar, o acesso aos arquivos armazenados em um disco rígido local.

Na Internet, as latências entre a ida e a volta de mensagens estão na faixa dos 5 a 500 ms, com uma média de 20 a 200 ms, dependendo da distância [www.globalcrossing.net]; portanto, os pedidos transmitidos pela Internet são de 10 a 100 vezes mais lentos do que nas redes locais mais rápidas. A maior parte dessa diferença de tempo é consequência dos atrasos em roteadores e da disputa por circuitos de comunicação da rede.

A Seção 7.5.1 irá discutir e comparar o desempenho de operações locais e remotas com mais detalhes.

Escalabilidade • As redes de computadores são uma parte indispensável da infraestrutura das sociedades modernas. Na Figura 1.6, mostramos o crescimento do número de computadores e servidores Web conectados na Internet em um período de 12 anos, terminando em 2005. Desde então, o crescimento tem sido tão rápido e diversificado que fica difícil encontrar estatísticas atualizadas e confiáveis. O potencial futuro tamanho da Internet é proporcional à população do planeta. É realista esperar que ela vá incluir vários bilhões de nós e centenas de milhões de computadores.

Esses números indicam as mudanças futuras no tamanho e na carga que a Internet deverá manipular. As tecnologias de rede em que ela é baseada não foram projetadas para suportar a atual escala da Internet, mas têm funcionado notavelmente bem. Para tratar a próxima fase de crescimento da Internet, algumas mudanças substanciais nos mecanismos de endereçamento e roteamento estão em andamento; elas serão descritas na Seção 3.4. Para aplicações cliente-servidor simples, como a Web, podemos esperar que o tráfego futuro cresça pelo menos na proporção do número de usuários ativos. A capacidade da infraestrutura da Internet de suportar esse crescimento dependerá de fatores econômicos para seu uso, em especial das cobranças impostas aos usuários e dos padrões de demanda de comunicação que realmente ocorrerem – por exemplo, seu grau de localidade.

Confiabilidade • Nossa discussão sobre os modelos de falhas, na Seção 2.4.2, descreveu o impacto dos erros de comunicação. Muitos aplicativos são capazes de se recuperar de falhas de comunicação e, assim, não exigem a garantia da comunicação isenta de erros. O “argumento do princípio fim-a-fim” (Seção 2.3.3) corrobora a visão de que o subsistema de comunicação não precisa oferecer comunicação totalmente isenta de erros; frequentemente, a detecção de erros de comunicação e sua correção é melhor realizada por *software* aplicativo. A confiabilidade da maior parte da mídia de transmissão física é muito alta. Quando ocorrem erros, é normalmente devido a falhas no *software* presente no remetente ou no destinatário (por exemplo, o computador destino deixa de aceitar um pacote) ou devido a estouros de *buffer*, em vez de erros na rede.

Segurança • O Capítulo 11 estabelecerá os requisitos e técnicas para se obter segurança em sistemas distribuídos. O primeiro nível de defesa adotado pela maioria das organizações é proteger suas redes e os computadores ligados a elas com um *firewall*. Um *firewall* cria uma barreira de proteção entre a intranet da organização e o restante da Internet. Seu propósito é proteger os recursos presentes nos computadores de uma organização contra o acesso de usuários ou processos externos e, ao mesmo tempo, controlar o uso dos recursos externos por usuários da própria organização.

Um *firewall* funciona em um *gateway* – um computador posicionado no ponto de entrada da rede intranet de uma organização. O *firewall* recebe e filtra todas as mensagens que entram e saem de uma organização. Ele é configurado, de acordo com a política de segurança da organização, para permitir a passagem de certas mensagens recebidas e enviadas e para rejeitar as demais. Voltaremos a esse assunto na Seção 3.4.8.

Para permitir que os aplicativos distribuídos superem as restrições impostas pelos *firewalls*, há necessidade de produzir um ambiente de rede seguro, no qual uma ampla variedade de aplicativos distribuídos possa ser implantada, com autenticação fim-a-fim, privacidade e segurança. Essa forma mais refinada e flexível de segurança pode ser obtida com o uso de técnicas de criptografia. Normalmente, ela é aplicada em um nível acima do subsistema de comunicação e, assim, não será tratada aqui, mas sim no Capítulo 11. As exceções incluem a necessidade de proteger componentes de rede, como os roteadores, contra interferência não autorizada em sua operação e a necessidade de se ter enlaces seguros para dispositivos móveis e nós externos, para permitir sua participação em uma intranet segura – o conceito de *rede privada virtual* (VPN, Virtual Private Network), discutido na Seção 3.4.8.

Mobilidade • Os dispositivos móveis, como *notebooks* e telefones móveis com acesso à Internet, mudam frequentemente de lugar e são reconectados em diferentes pontos da rede, ou mesmo usados enquanto estão em movimento. As redes sem fio (*wireless*) fornecem conectividade para tais dispositivos, mas os esquemas de endereçamento e de roteamento da Internet foram desenvolvidos antes do surgimento desses dispositivos móveis e não são adaptados às suas características de conexão intermitente em diversas sub-redes diferentes. Os mecanismos da Internet foram adaptados e ampliados para suportar mobilidade, mas o crescimento esperado para o uso de dispositivos móveis exigirá ainda mais desenvolvimentos.

Qualidade do serviço • No Capítulo 1, definimos qualidade do serviço como a capacidade de atender a prazos finais ao transmitir e processar fluxos de dados multimídia em tempo real. Isso impõe às redes de computadores novos requisitos importantes. Os aplicativos que transmitem dados multimídia exigem largura de banda garantida e latências limitadas para os canais de comunicação que utilizam. Alguns aplicativos variam sua demanda dinamicamente e especificam uma qualidade de serviço mínima aceitável e uma ótima desejada. O fornecimento de tais garantias e de sua manutenção será assunto do Capítulo 20.

Difusão seletiva* (multicasting) • A maior parte da comunicação em sistemas distribuídos se dá entre pares de processos, mas frequentemente também há necessidade de uma comunicação de um para muitos. Embora isso possa ser simulado por múltiplos envios para vários destinos, é mais dispendioso do que o necessário e pode não exibir as características de tolerância a falhas exigida pelos aplicativos. Por esses motivos, muitas tecnologias de rede suportam a transmissão de uma mensagem para vários destinatários.

* N. de R.T.: Embora “difusão seletiva” seja empregada como tradução para *multicast*, por questões de clareza, optamos por manter o termo em inglês.

3.2 Tipos de redes

Apresentamos aqui os principais tipos de redes usados pelos sistemas distribuídos: *redes pessoais*, *redes locais*, *redes de longa distância*, *redes metropolitanas* e suas variantes sem fio. As *redes interligadas*, como a Internet, são construídas a partir de redes de todos os tipos. A Figura 3.1 mostra as características de desempenho dos vários tipos de redes discutidos a seguir.

Alguns dos nomes usados para definir os tipos de redes são confusos, pois parecem se referir ao alcance físico (local, longa distância), porém também identificam tecnologias de transmissão físicas e protocolos de baixo nível, os quais diferem entre redes locais e de longa distância. Entretanto, algumas tecnologias de rede, como o ATM (Asynchronous Transfer Mode), são convenientes tanto para aplicativos locais como remotos e algumas redes sem fio também suportam transmissão local e metropolitana.

Denominamos redes interligadas as redes compostas por várias redes integradas de forma a fornecer um único meio de comunicação de dados. A Internet é o exemplo clássico de redes interligadas; ela é composta de milhões de redes locais, metropolitanas e de longa distância. Vamos descrever sua implementação com mais detalhes na Seção 3.4.

Redes pessoais (PAN – Personal Area Networks) • As PANs representam uma subcategoria das redes locais em que vários dispositivos eletrônicos-digitais transportados pelo usuário são conectados por uma rede de baixo custo e baixa energia. As PANs cabeadas não são de muito interesse, pois poucos usuários desejam ser incomodados com a presença ou necessidade de fios, mas as redes pessoais sem fio (WPANs, Wireless Personal Area Networks) têm cada vez mais importância, devido ao número de dispositivos pessoais disponíveis, como telefones móveis, PDAs, câmeras digitais, equipamentos sonoros, etc. Descreveremos a WPAN Bluetooth na Seção 3.5.3.

Redes locais (LANs – Local Area Networks) • As LANs transportam mensagens em velocidades relativamente altas entre computadores conectados em um único meio de comunicação, como um fio de par trançado, um cabo coaxial ou fibra óptica. Um *segmento* é uma seção de cabo que atende a um departamento, ou a um piso de um prédio, e que pode ter muitos computadores ligados. Nenhum roteamento de mensagens é necessário dentro de um segmento, pois o meio permite uma comunicação direta entre todos os computa-

	Exemplo	Alcance	Largura de banda (Mbps)	Latência (ms)
Redes cabeadas:				
LAN	Ethernet	1–2 km	10–10.000	1–10
WAN	Roteamento IP	mundial	0,010–600	100–500
MAN	ATM	2–50 km	1–600	10
Interligação em rede	Internet	mundial	0,5–600	100–500
Redes sem fio:				
WPAN	Bluetooth (IEEE 802.15.1)	10–30 m	0,5–2	5–20
WLAN	WiFi (IEEE 802.11)	0,15–1,5 km	11–108	5–20
WMAN	WiMAX (IEEE 802.16)	5–50 km	1,5–20	5–20
WWAN	Redes telefônicas 3G	celular: 1–5 km	348–14,4	100–500

Figura 3.1 Desempenho de rede.

dores conectados a ele. A largura de banda total é compartilhada entre os computadores de um segmento. Redes locais maiores, como aquelas que atendem a um campus ou a um prédio de escritórios, são compostas de muitos segmentos interconectados por hubs ou *switches* (veja a Seção 3.3.7). Nas redes locais, a largura de banda total é alta e a latência é baixa, exceto quando o tráfego de mensagens é muito alto.

Várias tecnologias para redes locais foram desenvolvidas nos anos 70 – *Ethernet*, *token rings* e *slotted rings*. Cada uma delas oferece uma solução eficiente e de alto desempenho, porém a Ethernet se tornou a tecnologia dominante para redes locais cabeadas. Ela foi originalmente introduzida no início dos anos 70, com uma largura de banda de 10 Mbps (milhões de bits por segundo) e, mais recentemente, ampliada para versões de 100 Mbps, 1000 Mbps (1 gigabit por segundo) e 10 Gbps. Descreveremos os princípios de operação das redes Ethernet na Seção 3.5.1.

Existe uma base instalada muito grande de redes locais, as quais possuem um ou mais computadores pessoais ou estações de trabalho. Seu desempenho geralmente é adequado para a implementação de sistemas e aplicativos distribuídos. A tecnologia Ethernet não possui as garantias de latência e de largura de banda necessárias para muitos aplicativos multimídia. As redes ATM foram desenvolvidas para preencher essa lacuna, mas seu custo inibe sua adoção em redes locais. Para superar esses inconvenientes, foram implantadas redes Ethernet comutadas, de altas velocidades, embora elas não tenham tanta eficiência como as ATM.

Redes de longa distância (WANs – Wide Area Networks) • As WANs transportam mensagens em velocidades mais lentas, frequentemente entre nós que estão em organizações diferentes e talvez separadas por grandes distâncias. As WANs podem estar localizadas em diferentes cidades, países ou continentes. O meio de transmissão empregado é formado por um conjunto de circuitos de comunicação interligando um grupo de equipamentos dedicados, chamados *roteadores*. Eles gerenciam a rede de comunicação e direcionam as mensagens, ou pacotes, para seus destinos. Na maioria das redes, as operações de roteamento introduzem um atraso em cada um dos pontos de uma rota; portanto, a latência total da transmissão de uma mensagem depende da rota que ela segue e das cargas de tráfego nos diversos segmentos de rede pelos quais ela passa. Nas redes atuais, essas latências podem ser de até 0,1 a 0,5 segundos. A velocidade de propagação dos sinais eletromagnéticos na maioria das mídias de comunicação é próxima à velocidade da luz e isso estabelece um limite inferior para a latência da transmissão para redes de longa distância. Por exemplo, o atraso da propagação de um sinal para ir da Europa para a Austrália por meio de um enlace terrestre é de aproximadamente 0,13 segundos, e os sinais via satélite geoestacionário entre quaisquer dois pontos na superfície da Terra estão sujeitos a um atraso de aproximadamente 0,20 segundos.

As larguras de banda disponíveis pela Internet também variam bastante. Velocidades de até 600 Mbps são comuns, mas a maior parte das transferências de dados é feita em velocidades de 1 a 10 Mbps.

Redes metropolitanas (MANs – Metropolitan Area Networks) • Este tipo de rede é baseada em uma infraestrutura de cabeamento de fibra óptica e cabos de cobre de alta largura de banda, instalados em algumas cidades para transmissão de vídeo, voz e outros dados em distâncias de até 50 quilômetros. Tem sido usada uma variedade de tecnologias para implementar o roteamento de dados nas MANs, variando de Ethernet a ATM.

As conexões DSL (Digital Subscriber Line – linha digital de assinante) e de modem a cabo, agora disponíveis em muitos países, são um exemplo. Normalmente, a DSL utiliza comutadores ATM, localizados nas estações telefônicas, para redirecionar os dados digitais que trafegam nos pares de fios de cobre trançados (usando sinalização de alta fre-

quência na fiação existente das próprias conexões telefônicas), para a casa ou escritório do assinante, em velocidades na faixa de 1 a 10 Mbps. O uso de pares de cobre trançados para as conexões de assinantes DSL limitam sua abrangência a cerca de 5,5 km a partir do comutador. As conexões de modem a cabo usam sinalização analógica nas redes de televisão a cabo para atingir velocidades de 15 Mbps por meio de cabo coaxial, com uma abrangência maior que a DSL. O termo DSL representa, na verdade, uma família de tecnologias, às vezes referida como xDSL, incluindo, por exemplo, a ADSL (ou Asymmetric Digital Subscriber Line – linha digital de assinante assimétrica). Os desenvolvimentos mais recentes incluem a VDSL e a VDSL2 (Very High Bit Rate DSL – DSL com taxa de bits muito alta), capazes de atingir velocidades de até 100 Mbps e projetadas para suportar tráfego multimídia variado, incluindo a TV de alta definição (HDTV).

Redes locais sem fio (WLANs – Wireless Local Area Networks) • As WLANs são projetadas para substituir as LANs cabeadas. Seu objetivo é fornecer conectividade para dispositivos móveis, ou simplesmente eliminar a necessidade de uma infraestrutura com fios e cabos para interconectar computadores dentro de casas e prédios de escritório entre si e a Internet. Seu uso é difundido em diversas variantes do padrão IEEE 802.11 (WiFi), oferecendo larguras de banda entre 10 e 100 Mbps com abrangência de até 1,5 quilômetros. A Seção 3.5.2 fornece mais informações sobre seu método de operação.

Redes metropolitanas sem fio (WMANs – Wireless Metropolitan Area Network) • O padrão IEEE 802.16 WiMAX é destinado a essa classe de rede. Seu propósito é fornecer uma alternativa às conexões cabeadas para casas ou prédios de escritório e substituir as redes 802.11 WiFi em algumas aplicações.

Redes de longa distância sem fio (WWANs – Wireless Wide Area Networks) • A maioria das redes de telefonia móvel é baseada em tecnologias de rede digital sem fio, como o padrão GSM (Global System for Mobile), usado na maior parte dos países do mundo. As redes de telefonia móvel são projetadas para operar em áreas amplas (normalmente, países ou continentes inteiros) por meio de conexões de rádio. O sinal de rádio é confinado em regiões denominadas células, e o alcance da comunicação celular, ou seja, sua cobertura, é garantida pela interligação e superposição dessas células. Com a infraestrutura da telefonia celular é possível oferecer conexão à Internet para dispositivos móveis. As redes celulares oferecem taxas de transmissão de dados relativamente baixas – 9,6 a 33 Kbps – mas agora está disponível a “terceira geração” (3G) de redes de telefonia móvel, com taxas de transmissão de dados na faixa de 2 a 14,4 Mbps quando o equipamento está fixo e 348 Kbps quando está em movimento (em um carro, por exemplo). A tecnologia subjacente é conhecida como UMTS (Universal Mobile Telecommunications System – sistema de telecomunicações móveis universais). Também foi definido um caminho para que o UMTS evolua para 4G, com taxa de dados de até 100 Mbps. Os leitores que estiverem interessados em estudar mais a fundo (do que podemos aqui) as tecnologias de redes móveis e sem fio, podem consultar o excelente livro de Stojmenovic [2002].

Inter-redes • Uma rede inter-rede, ou redes interligadas, é um subsistema de comunicação em que várias redes são unidas para fornecer recursos de comunicação de dados comuns, abstraindo as tecnologias e os protocolos das redes componentes individuais e os métodos usados para sua interconexão.

As redes interligadas são necessárias para o desenvolvimento de sistemas distribuídos abertos e extensíveis. A característica aberta dos sistemas distribuídos sugere que as redes neles usadas devem ser extensíveis para um número muito grande de computadores, enquanto as redes individuais têm espaços de endereço restritos e algumas possuem

limitações no desempenho, sendo incompatíveis para uso em larga escala. Diversas tecnologias de rede local e de longa distância podem ser integradas para fornecer a capacidade de interligação em rede necessária para um grupo de usuários. Assim, as redes interligadas trazem muitas das vantagens dos sistemas abertos para o aprovisionamento de comunicação em sistemas distribuídos.

As redes interligadas são construídas a partir de uma variedade de redes individuais. Elas são interconectadas por equipamentos dedicados, os *roteadores*, e por computadores de propósito geral, os *gateways*. O subsistema de comunicação resultante é dado por uma camada de *software* que suporta o endereçamento e a transmissão de dados para todos os computadores das redes interligadas. O resultado pode ser considerado como uma “rede virtual” construída pela sobreposição de uma camada de integração de redes através de um meio de comunicação composto por redes individuais, roteadores e *gateways* subjacentes. A Internet é o maior exemplo de interligação em rede e seus protocolos TCP/IP são um exemplo da camada de integração mencionada anteriormente.

Erros em comunicação em rede • Um ponto adicional de comparação, não mencionado na Figura 3.1, é a frequência e os tipos de falha que podem ser esperados nos diferentes tipos de rede. A confiabilidade da mídia de transmissão de dados subjacente é muito alta em todos os tipos, exceto nas redes sem fio, nas quais pacotes são frequentemente perdidos devido à interferência externa. No entanto, os pacotes podem ser perdidos em todos os tipos de rede, devido aos atrasos de processamento e ao estouro de *buffer* nos comutadores e nos nós de destino, e essa é a causa mais comum da perda de pacotes.

Os pacotes também podem ser entregues em uma ordem diferente daquela em que foram emitidos. Isso acontece apenas em redes em que os pacotes são direcionados individualmente – principalmente em redes de longa distância. Outro erro comum é a entrega de cópias duplicadas dos pacotes; normalmente, isso é uma consequência da suposição, pelo remetente, de que um pacote foi perdido. Neste caso, o pacote é retransmitido e, então, o original e a cópia retransmitida aparecem no destino.

3.3 Conceitos básicos de redes

A base de todas as redes de computadores é a técnica de comutação de pacotes, desenvolvida pela primeira vez nos anos 60. Essa técnica permite que pacotes de dados endereçados a diferentes destinos compartilhem um único enlace de comunicação, ao contrário do que ocorre com a tecnologia de comutação de circuitos, que fundamenta a telefonia convencional. Os pacotes são enfileirados em um *buffer* e transmitidos quando o enlace está disponível. A comunicação é assíncrona – as mensagens chegam ao seu destino após um atraso que varia de acordo com o tempo que os pacotes levam para trafegar pela rede.

3.3.1 Transmissão de pacotes

Na maioria das aplicações de redes de computadores, o requisito fundamental é a transmissão de unidades lógicas de informação ou *mensagens* – sequências de dados de comprimento arbitrário. Porém, antes que uma mensagem seja transmitida, ela é subdividida em *pacotes**. A forma mais simples de pacote é uma sequência de dados binários (uma

* N. de R.T.: “Pacote” é um termo genérico para referenciar uma sequência de dados binários com tamanho limitado usado como unidade de transmissão. Entretanto, conceitualmente, cada camada de um protocolo de rede define sua própria unidade de transmissão, denominada de PDUs (Protocol Data Unit), as quais possuem nomes específicos no TCP/IP: datagrama para o IPv4, pacote para o IPv6 e datagrama e segmento para o UDP e o TCP, respectivamente.

sequência de bits ou bytes) de comprimento limitado, junto a informações de endereçamento suficientes para identificar os computadores de origem e de destino. Pacotes de comprimento limitado são usados:

- para que cada computador da rede possa alocar armazenamento em *buffer* suficiente para armazenar o maior pacote possível a ser recebido;
- para eliminar os atrasos excessivos que ocorreriam na espera da liberação de canais de comunicação, caso as mensagens longas fossem transmitidas sem nenhum tipo de subdivisão.

3.3.2 Fluxo de dados

A transmissão e exibição de áudio e vídeo em tempo real é referida como *fluxo* (ou *stream*). Isso exige larguras de banda muito mais altas do que a maioria das outras formas de comunicação em sistemas distribuídos. Já mencionamos, no Capítulo 2, que os aplicativos multimídia dependem da transmissão de fluxos de dados de áudio e de vídeo em altas velocidades garantidas e com latências limitadas.

Um fluxo de vídeo exige uma largura de banda de cerca de 1,5 Mbps, se os dados forem compactados, ou 120 Mbps, caso contrário. Os pacotes do protocolo Internet UDP (datagramas) são geralmente usados para conter os quadros de vídeo, mas como o fluxo de dados é contínuo, em oposição ao tráfego intermitente gerado pelas interações cliente-servidor típicas, os pacotes são manipulados de forma um pouco diferente. O *tempo de reprodução* de um elemento multimídia, como um quadro de vídeo, é o tempo no qual ele deve ser exibido (para um elemento de vídeo) ou convertido em som (para uma amostra de som). Por exemplo, em um fluxo de quadros de vídeo com uma velocidade de projeção de 24 quadros por segundo, o quadro N tem um tempo de reprodução de $N/24$ segundos, após o tempo de início do fluxo. Os elementos que chegam ao seu destino depois de seu tempo de reprodução não são mais úteis e serão eliminados pelo processo receptor.

A distribuição adequada de fluxos de áudio e vídeo depende da disponibilidade de conexões com qualidade de serviço – largura de banda, latência e confiabilidade, tudo deve ser considerado. De preferência, uma qualidade de serviço apropriada deve ser garantida. Em geral, a Internet não oferece essa capacidade, e a qualidade dos fluxos de vídeo em tempo real às vezes reflete isso; mas, em intranets particulares, como as operadas por empresas de mídia, às vezes essas garantias são alcançadas. Isso pode ser obtido com o estabelecimento de um canal de comunicação de um fluxo multimídia da origem para o destino, com uma rota predefinida pela rede, com um conjunto de recursos reservados em cada nó pelo qual ele passará e, onde for apropriado, com o uso de *buffer*, para atenuar as irregularidades no fluxo de dados através do canal de comunicação. Os dados podem, então, passar pelo canal, do remetente para o destinatário, na velocidade exigida.

As redes ATM são especificamente projetadas para fornecer alta largura de banda com baixa latência e para suportar qualidade de serviço por meio da reserva de recursos de rede. O IPv6, o novo protocolo de rede da Internet, descrito em linhas gerais na Seção 3.4.4, inclui recursos que permitem que cada um dos pacotes IP (datagrama) de um fluxo em tempo real sejam identificados e tratados separadamente dos demais.

Os subsistemas de comunicação que fornecem garantia de qualidade do serviço exigem meios para a alocação prévia de recursos de rede e o cumprimento dessas alocações. O protocolo RSVP (Resource Reservation Protocol) [Zhang *et al.* 1993] permite que os aplicativos negociem a alocação prévia de largura de banda para fluxos de

dados em tempo real. O protocolo RTP (Real Time Transport Protocol) [Schulzrinne *et al.* 1996] é um protocolo de transferência de dados, em nível de aplicação, que inclui em cada pacote os detalhes do tempo de reprodução e outros requisitos de sincronismo. A disponibilidade de implementações eficazes desses protocolos na Internet, em geral, dependerá de alterações substanciais nas camadas de transporte e de rede. O Capítulo 20 discutirá em mais detalhes as necessidades dos aplicativos multimídia distribuídos.

3.3.3 Esquemas de comutação

Uma rede consiste em um conjunto de nós interconectados por circuitos. Para transmitir informações entre dois nós arbitrários, é necessário um sistema de comutação. Definimos aqui os quatro tipos de comutação usados na interligação de redes de computadores.

Difusão (broadcast) • O *broadcast* é uma técnica de transmissão que não envolve nenhuma comutação. Tudo é transmitido para cada nó e fica por conta dos receptores recuperar as transmissões a eles endereçadas. Algumas tecnologias de rede local, incluindo a Ethernet, são baseadas em *broadcast*. A interligação em rede sem fio é necessariamente baseada em *broadcast*, mas na ausência de circuitos fixos, as difusões são organizadas de modo a atingir nós agrupados em células.

Comutação de circuitos • Houve um tempo em que as redes telefônicas eram as únicas redes de telecomunicações. Sua operação era simples de entender: quando o chamador discava um número, o par de fios de seu telefone, que ia até a estação local, era conectado por um comutador automático dessa central ao par de fios conectados ao telefone da outra pessoa. Para uma chamada interurbana, o processo era semelhante, mas a conexão seria comutada até seu destino por meio de diversas estações intermediárias. Às vezes, esse sistema é referenciado como sistema telefônico antigo (POTS, Plain Old Telephone System). Trata-se de uma *rede de comutação de circuitos* típica.

Comutação de pacotes • O advento dos computadores e da tecnologia digital trouxe muitas possibilidades para as telecomunicações, como as capacidades básicas de processamento e armazenamento. Isso possibilitou a construção de redes de comunicação de uma maneira bastante diferente. Esse novo tipo de rede de comunicação é chamada de *rede de armazenamento e encaminhamento (store-and-forward)*. Em vez de estabelecer e desfazer conexões para construir circuitos, uma rede de armazenamento e encaminhamento apenas encaminha pacotes de sua origem para seu destino. Existe um computador em cada nó de comutação (isto é, onde vários circuitos precisam ser interconectados). Cada pacote que chega a um nó é primeiramente armazenado em sua memória e, depois, processado por um programa que o transmite a um circuito de saída, que transferirá o pacote para outro nó mais próximo de seu destino final.

Não há nada de realmente novo nessa ideia: o sistema postal é uma rede de armazenamento e encaminhamento de cartas, com o processamento feito por seres humanos ou automaticamente nos centros de triagem. No entanto, em uma rede de computadores, os pacotes podem ser armazenados e processados de forma rápida o suficiente para dar a ilusão de uma transmissão instantânea, mesmo que o pacote tenha de ser direcionado por meio de muitos nós.

Frame relay • Na realidade, leva desde algumas dezenas de microsegundos a alguns milissegundos para, em uma rede de armazenamento e encaminhamento, comutar um pacote em cada nó. Esse atraso de comutação depende do tamanho do pacote, da velocidade do hardware e da quantidade do tráfego, mas seu limite inferior é determinado pela largura

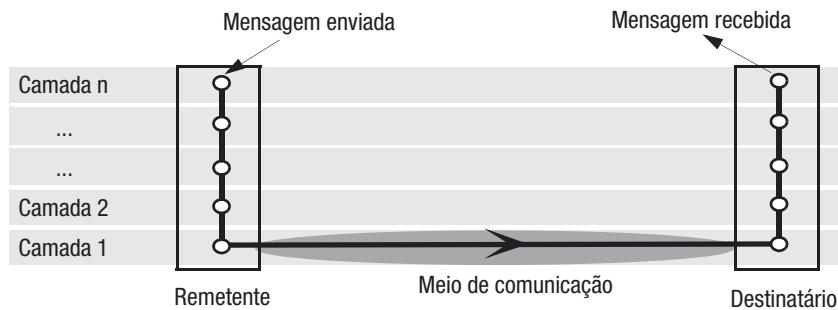


Figura 3.2 Organização conceitual de protocolos em camadas.

de banda da rede, pois o pacote inteiro deve ser recebido antes que possa ser encaminhado para outro nó. Grande parte da Internet é baseada em comutação de pacotes e, conforme já vimos, na Internet, mesmo os pacotes pequenos normalmente demoram até 200 milisegundos para chegar aos seus destinos. Atrasos dessa magnitude são longos demais para aplicações em tempo real, como telefonia e videoconferência, em que atrasos de menos de 50 milissegundos são necessários para manter uma conversação em alta qualidade.

O método de comutação *frame relay* traz algumas das vantagens da comutação de circuitos para as redes de comutação de pacotes. O problema de atraso é superado por meio do emprego de pequenos pacotes, denominados quadros (*frames*), que são comutados dinamicamente. Os nós de comutação (que normalmente são processadores de propósito específico) redirecionam os quadros baseados apenas no exame de seus primeiros bits; sem armazená-los como um todo. As redes ATM são o melhor exemplo do emprego dessa técnica – redes ATM de alta velocidade podem transmitir pacotes por redes compostas de muitos nós em poucas dezenas de microssegundos.

3.3.4 Protocolos

O termo *protocolo* é usado para designar um conjunto bem conhecido de regras e formatos a serem usados na comunicação entre processos a fim de realizar uma determinada tarefa. A definição de protocolo tem duas partes importantes:

- uma especificação da sequência de mensagens que devem ser trocadas;
- uma especificação do formato dos dados nas mensagens.

A existência de protocolos bem-conhecidos permite que os vários componentes de *software* de um sistema distribuído sejam desenvolvidos e implementados de forma independente, usando diferentes linguagens de programação, em computadores que podem usar distintas representações internas para códigos e dados.

Um protocolo é implementado por meio de dois módulos de *software* localizados nos computadores origem e destino. Por exemplo, um *protocolo de transporte* transmite mensagens de qualquer comprimento entre um processo remetente e um processo destino. Um processo que queira transmitir uma mensagem para outro emite uma chamada para o módulo de protocolo de transporte, passando a ele uma mensagem em um formato especificado. O *software* de transporte se ocupa, então, com o envio da mensagem para o destino, subdividindo-a em pacotes de tamanho e formato especificados, que podem ser transmitidos para o destino por meio do *protocolo de rede* – outro protocolo de nível

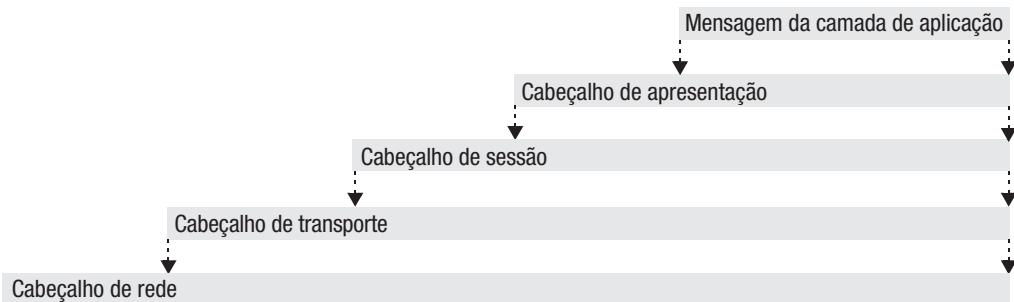


Figura 3.3 Encapsulamento em protocolos dispostos em camadas.

inferior. No computador destino, o módulo de protocolo de transporte correspondente recebe o pacote por meio do módulo de protocolo em nível de rede e realiza transformações inversas para recompor a mensagem, antes de passá-la para um processo destino.

Camadas de protocolo • O *software* de rede é organizado em uma hierarquia de camadas ou níveis. Cada camada apresenta uma interface bem definida para as camadas que estão acima dela e implementa novos serviços sobre as funcionalidades do sistema de comunicação subjacente. Uma camada é representada por um módulo de *software* em cada computador conectado à rede. A Figura 3.2 ilustra a estrutura e o fluxo de dados que ocorre quando uma mensagem é transmitida usando um protocolo em camadas. Logicamente, cada módulo de *software* de um nível de um computador se comunica diretamente com seu correspondente no outro computador, mas, na realidade, os dados não são transmitidos diretamente entre os módulos de protocolo de cada nível. Em vez disso, cada camada de *software* de rede se comunica com as camadas acima e abaixo dela por intermédio de chamadas de procedimentos locais.

No lado remetente, cada camada (exceto a superior, ou camada de aplicação) aceita dados da camada que está acima dela em um formato especificado e aplica transformações para encapsular esses dados em um formato próprio, específico a si mesma, antes de repassá-los para a camada abaixo dela. A Figura 3.3 ilustra esse procedimento aplicado às quatro camadas superiores do conjunto de protocolos OSI (apresentado na próxima seção). A figura mostra a inclusão de cabeçalhos que contêm informações de controle de cada uma das camadas. Conceitualmente, é possível que as camadas também insiram informações de controle na parte posterior de cada pacote, porém, por questões de clareza, isso foi omitido na Figura 3.3; ela também presume que a mensagem da camada de aplicação a ser transmitida é menor do que o tamanho de pacote máximo da rede subjacente. Se não fosse, então ela teria de ser segmentada e encapsulada em vários pacotes. No lado destino, os dados recebidos por uma camada sofrem as transformações inversas antes de serem repassados para a camada superior. O tipo de protocolo da camada superior é incluído no cabeçalho de cada camada para permitir que a pilha de protocolos no destino selecione os componentes de *software* corretos para desempacotar os pacotes.

Cada camada fornece serviços para a camada acima dela. A camada mais inferior é a *camada física*, que oferece o serviço de transmissão física dos dados. Isso é implementado por um meio de comunicação (cabos de cobre ou fibra óptica, canais de comunicação via satélite ou transmissão de rádio) e por circuitos de sinalização analógicos que inserem sinais eletromagnéticos no meio de comunicação no nó de envio e os capta no nó de recepção. Nos nós de recepção, os dados são recebidos e repassados para as camadas superiores

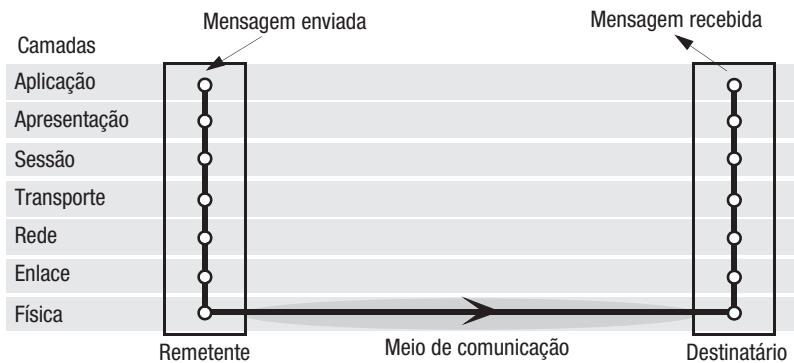


Figura 3.4 Camadas de protocolo no modelo de protocolo Open System Interconnection (OSI) da ISO.

da hierarquia de protocolos. Cada camada realiza transformações nos dados até estarem em uma forma que possa ser entregue para o processo destinatário pretendido.

Conjuntos de protocolo • Um conjunto completo de camadas de protocolo é referido como *conjunto de protocolos* ou *pilha de protocolos*, refletindo a estrutura em camadas. A Figura 3.4 mostra uma pilha de protocolos que obedece ao modelo de referência de sete camadas do padrão *OSI (Open System Interconnection)*, adotado pela ISO (International Organization for Standardization) [ISO 1992]. O modelo de referência OSI foi adotado para estimular o desenvolvimento de padrões de protocolo que atendessem aos requisitos dos sistemas abertos.

O objetivo de cada camada no modelo de referência OSI está resumido na Figura 3.5. Conforme seu nome implica, ele é uma estrutura para a definição de protocolos e não uma definição de um conjunto de protocolos específico. Os conjuntos de protocolo que obedecem ao modelo OSI devem incluir, pelo menos, um protocolo específico em cada um dos sete níveis, ou camadas, definidos pelo modelo.

A disposição de protocolos em camadas apresenta vantagens substanciais na simplificação e na generalização das interfaces de *software* para acesso aos serviços de comunicação das redes, mas também acarreta custos de desempenho significativos. A transmissão de uma mensagem em nível de aplicação, por meio de uma pilha de protocolos com N camadas, envolve N transferências de controle entre as camadas de *software*, sendo uma delas a entrada do sistema operacional e, como parte dos mecanismos de encapsulamento, são necessários N cópias dos dados. Todas essas sobrecargas resultam em taxas de transferência de dados entre processos aplicativos muito mais lentas do que a largura de banda disponível da rede.

A Figura 3.5 inclui exemplos de protocolos usados na Internet, mas sua implementação não segue o modelo OSI em dois aspectos. Primeiro, as camadas de aplicação, apresentação e sessão não são claramente distinguidas na pilha de protocolos Internet. Em vez disso, as camadas de aplicação e apresentação são implementadas como uma única camada de *middleware* ou, separadamente, dentro de cada aplicativo. Assim, o CORBA implementa invocações entre objetos e representação de dados em uma biblioteca de *middleware* que é incluída em cada processo aplicativo (veja o Capítulo 8 para mais detalhes sobre o CORBA). De forma similar, os navegadores Web e outros aplicativos que exigem canais seguros empregam a Secure Sockets Layer (Capítulo 11) como biblioteca de funções.

Segundo, a camada de sessão é integrada à camada de transporte. Os conjuntos de protocolos de redes interligadas incluem uma camada de aplicação, uma camada de

Camada	Descrição	Exemplos
Aplicativo	Protocolos projetados para atender aos requisitos de comunicação de aplicativos específicos, frequentemente definindo uma interface para um serviço.	HTTP, FTP, SMTP, CORBA IIOP
Apresentação	Os protocolos deste nível transmitem dados em uma representação de rede independente das usadas em cada computador, as quais podem diferir. A criptografia, se exigida, também é feita nesta camada.	Segurança TLS, CORBA Data Rep.
Sessão	Neste nível são realizadas a confiabilidade e a adaptação, como a detecção de falhas e a recuperação automática.	SIP
Transporte	Este é o nível mais baixo em que mensagens (em vez de pacotes) são manipuladas. As mensagens são endereçadas para portas de comunicação associadas aos processos. Os protocolos desta camada podem ser orientados a conexão ou não.	TCP, UDP
Rede	Transfere pacotes de dados entre computadores em uma rede específica. Em uma rede de longa distância, ou em redes interligadas, isso envolve a geração de uma rota passando por roteadores. Em uma única rede local, nenhum roteamento é exigido.	IP, circuitos virtuais ATM
Enlace	Responsável pela transmissão de pacotes entre nós que estão diretamente conectados por um enlace físico. Em uma rede de longa distância, a transmissão é feita entre pares de roteadores ou entre roteadores e <i>hosts</i> . Em uma rede local, ela é feita entre qualquer par de <i>hosts</i> .	MAC Ethernet, transferência de célula ATM, PPP
Física	São os circuitos e o <i>hardware</i> que materializam a rede. Essa camada transmite sequências de dados binários através de sinalização analógica, usando modulação em amplitude ou em frequência de sinais elétricos (em circuitos a cabo), sinais luminosos (em circuitos de fibra óptica) ou outros sinais eletromagnéticos (em circuitos de rádio e micro-ondas).	Sinalização de banda-base Ethernet, ISDN

Figura 3.5 Exemplos de protocolos OSI.

transporte e uma *camada inter-rede*. A camada inter-rede implementa um rede virtual, responsável por transmitir pacotes das redes interligadas para um computador de destino. Um *pacote inter-rede* é a unidade de dados transmitidos entre as redes interligadas.

Os protocolos de redes interligadas são sobrepostos em redes subjacentes, conforme ilustrado na Figura 3.6. A *camada de enlace* aceita pacotes inter-rede e os converte adequadamente em pacotes de transmissão de cada rede subjacente.

Montagem de pacotes • A tarefa de dividir mensagens em pacotes antes da transmissão e sua remontagem no computador destino normalmente é executada na camada de transporte.

Os pacotes do protocolo da camada de rede consistem em um *cabeçalho* e em um . Na maioria das tecnologias de rede, o campo de dados tem comprimento variável, com o comprimento máximo chamado de *unidade de transferência máxima (MTU, Maximum Transfer Unit)*. Se o comprimento de uma mensagem ultrapassar o MTU da camada de rede subjacente, ela deve ser fragmentada em porções de tamanho apropriado, identificada com números de sequência para ser remontada e transmitida em vários pacotes. Por exemplo, o MTU da Ethernet é de 1.500 bytes – não mais do que esse volume de dados pode ser transmitido em um único pacote Ethernet.

Embora o protocolo IP (Internet Protocol) seja um protocolo de camada de rede, seu MTU é extraordinariamente grande, 64 Kbytes, (na prática, são usados frequentemente 8 Kbytes, pois alguns nós não conseguem manipular pacotes grandes assim). Qualquer que seja o valor do MTU adotado pelo IP, pacotes maiores que o MTU Ethernet

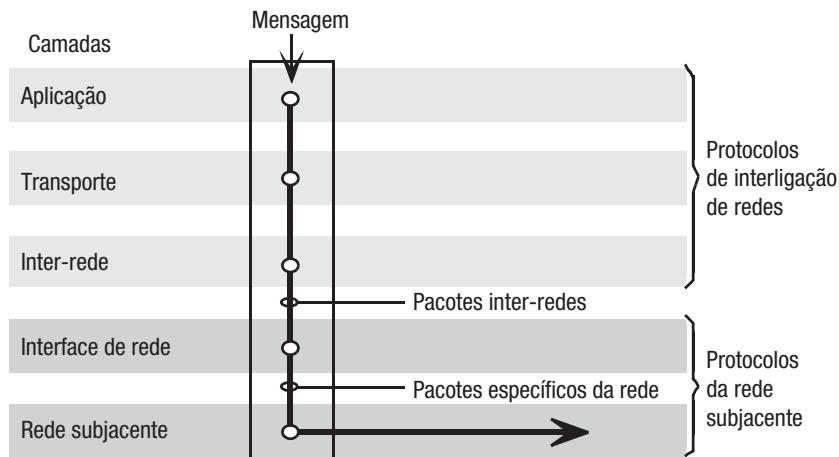


Figura 3.6 Camadas de interligação de redes.

devem ser fragmentados para transmissão em redes Ethernet. Os pacotes IP são denominados datagramas.

Portas • A tarefa da camada de transporte é fornecer um serviço de transporte de mensagens entre pares de *portas* independentemente do tipo de rede realmente empregado. As portas são pontos de destino definidos pelo *software* em um computador *host*. Elas são ligadas a processos, permitindo que a transmissão de dados seja endereçada para um processo específico em um nó de destino. Aqui, discutiremos o endereçamento via portas de acordo com sua implementação na Internet e na maioria das redes. O Capítulo 4 descreverá sua programação.

Endereçamento • A camada de transporte é responsável pela entrega de mensagens para seus destinos baseados em *endereços de transporte* que são compostos pelo *endereço de rede* de um computador *host* e um *número de porta*. Um endereço de rede é um identificador numérico que identifica exclusivamente um computador *host* e permite sua localização pelos nós responsáveis pelo roteamento. Na Internet, cada computador *host* recebe um número IP, que identifica o computador e a sub-rede à qual ele está conectado, permitindo que os dados sejam direcionados a ele a partir de qualquer outro nó, conforme descrito nas seções a seguir. Em uma rede local Ethernet não existem nós de roteamento; cada *host* é responsável por reconhecer e selecionar os pacotes endereçados a ele.

Os serviços Internet conhecidos, como HTTP ou FTP, receberam *números de porta de contato* que são registrados junto a uma autoridade central, a IANA (Internet Assigned Numbers Authority) [www.iana.org]. Para acessar um serviço em um determinado *host*, o pedido é enviado para a porta associada a esse serviço, nesse *host*. Alguns serviços, como o FTP (porta 21), em sua execução, alocam uma nova porta (com um número privado) e enviam o número dessa nova porta para o cliente. O cliente usa essa nova porta para efetuar o restante de uma transação ou de uma sessão. Outros serviços, como o HTTP (porta 80), fazem todas as suas transações diretamente pela porta de contato.

Os números de porta abaixo de 1023 são definidos como *portas bem conhecidas*, cujo uso é restrito a processos privilegiados na maioria dos sistemas operacionais. As portas entre 1024 e 49151 são *portas registradas*, para as quais a IANA mantém descri-

ções de serviço, e as portas restantes, até 65535, estão disponíveis para propósitos gerais. Na prática, todas as portas acima de 1023 podem ser usadas para propósitos gerais, mas os computadores que as utilizam para isso não podem acessar simultaneamente os serviços registrados correspondentes.

A alocação de um número de porta fixo não é apropriada para o desenvolvimento de sistemas distribuídos, os quais frequentemente abrangem uma multiplicidade de servidores, incluindo aqueles alocados dinamicamente. As soluções para esse problema envolvem a alocação dinâmica de portas para serviços e o aprovisionamento de mecanismos de vinculação para permitir que os clientes localizem serviços e suas portas usando nomes simbólicos. Algumas dessas técnicas serão discutidas mais a fundo no Capítulo 5.

Entrega de pacotes • Existem duas estratégias para a entrega de pacotes pela camada de rede:

Redes baseadas em datagramas: o termo datagrama se refere à semelhança desse modo de entrega de dados com a maneira pela qual cartas e telegramas são distribuídos. A característica essencial das redes de datagramas é que a distribuição de cada pacote é um procedimento independente; nenhuma configuração é exigida e, uma vez que o pacote é entregue, a rede não mantém mais nenhuma informação sobre ele. Em uma rede de datagramas, uma sequência de pacotes transmitida por um *host* para um único destino pode seguir rotas diferentes (se, por exemplo, a rede for capaz de se adaptar às falhas de manipulação ou de reduzir os efeitos de congestionamentos localizados) e, quando isso ocorre, eles podem chegar fora da sequência em que foram emitidos.

Todo datagrama contém o endereço de rede dos *hosts* de origem e de destino; este último é um parâmetro essencial para o processo de roteamento, conforme descreveremos na próxima seção. As redes baseadas em datagramas são o conceito sobre o qual as redes de pacotes foram originalmente concebidas e são encontradas na maioria das redes de computadores atuais. A camada de rede da Internet – IP –, a Ethernet e um grande número das tecnologias de rede local com e sem fio são baseadas em datagramas.

Redes baseadas em circuito virtual: alguns serviços da camada de rede implementam o envio de pacotes de maneira análoga a uma rede telefônica. Um circuito virtual deve ser configurado antes que os pacotes possam passar de um *host* de origem A para um *host* de destino B. O estabelecimento de um circuito virtual envolve a definição de um caminho entre a origem e o destino, possivelmente passando por vários nós intermediários. Em cada nó ao longo do caminho, é criada uma entrada em uma tabela, indicando qual enlace deve ser usado para atingir a próxima etapa do caminho.

Uma vez configurado o circuito virtual, ele pode ser usado para transmitir qualquer quantidade de pacotes. Cada pacote da camada de rede contém apenas o número de circuito virtual no lugar dos endereços de origem e de destino. Os endereços não são necessários, pois os pacotes são direcionados nos nós intermediários pela referência ao número do circuito virtual. Quando um pacote chega ao seu destino, a origem pode ser determinada a partir do número do circuito virtual.

A analogia com as redes telefônicas não deve ser tomada literalmente. No sistema telefônico antigo, uma chamada telefônica resultava no estabelecimento de um circuito físico entre os correspondentes, e os enlaces de voz a partir dos quais ele era construído ficavam reservados para seu uso exclusivo. Na distribuição de pacotes por circuito virtual, os circuitos são representados apenas por entradas de tabela nos nós

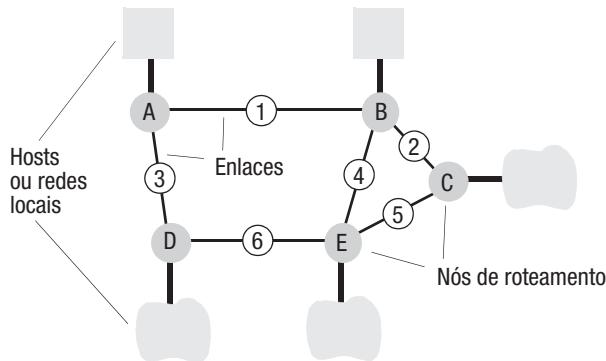


Figura 3.7 Roteamento em redes de longa distância.

de encaminhamento, e os enlaces em que os pacotes são redirecionados são ocupados apenas pelo tempo de transmissão do pacote; no restante do tempo, eles estão liberados. Portanto, um único enlace pode ser empregado em muitos circuitos virtuais separados. A tecnologia de rede de circuito virtual mais importante em uso atualmente é a ATM; já mencionamos (na Seção 3.3.3) que ela tira proveito de latências menores para a transmissão de pacotes individuais; isso é um resultado direto do uso de circuitos virtuais. Contudo, a necessidade de uma fase de configuração resulta em um pequeno atraso antes que os pacotes possam ser enviados para um novo destino.

A distinção entre a distribuição de pacotes baseados em redes de datagramas e de circuito virtual na camada de rede não deve ser confundida com dois mecanismos de nome semelhante na camada de transporte – transmissão orientada à conexão e não orientada à conexão. Vamos descrevê-los na Seção 3.4.6, no contexto dos protocolos de transporte da Internet, UDP (não orientado à conexão) e TCP (orientado à conexão). Aqui, simplesmente mencionamos que cada um desses modos da camada de transporte pode ser implementado sobre qualquer tipo de rede.

3.3.5 Roteamento

O roteamento é uma função necessária sempre que se têm a interligação de redes permitindo a comunicação entre seus *hosts*. Note que em redes locais, como a Ethernet, o roteamento é desnecessário já que os *hosts* têm a capacidade de se comunicar diretamente. Em redes de grande dimensão é empregado o *roteamento adaptativo*: a melhor rota de comunicação entre dois pontos é periodicamente reavaliada, levando em conta o tráfego corrente na rede e as falhas, como conexões desfeitas ou nós de roteamento danificados.

A entrega de pacotes aos seus destinos, em uma inter-rede como a ilustrada na Figura 3.7, é de responsabilidade coletiva dos nós de roteamento localizados nos pontos de interconexão. A não ser que os *hosts* de origem e destino estejam na mesma rede local, o pacote precisa ser transmitido em uma série de etapas ou passos (*hops*), passando por vários nós de roteamento intermediários. A determinação das rotas para a transmissão de pacotes para seus destinos é de responsabilidade de um *algoritmo de roteamento* – implementado na camada de rede em cada nó.

Um algoritmo de roteamento tem duas partes:

1. Tomar decisões para determinar a rota que cada pacote deve seguir ao passar pela rede. Em camadas de rede com comutação de circuitos, como o X.25, e em redes

Nó A			Nó B			Nó C		
Para	Saída	Custo	Para	Saída	Custo	Para	Saída	Custo
A	local	0	A	1	1	A	2	2
B	1	1	B	local	0	B	2	1
C	1	2	C	2	1	C	local	0
D	3	1	D	1	2	D	5	2
E	1	2	E	4	1	E	5	1

Nó D			Nó E		
Para	Saída	Custo	Para	Saída	Custo
A	3	1	A	4	2
B	3	2	B	4	1
C	6	2	C	5	1
D	local	0	D	6	1
E	6	1	E	local	0

Figura 3.8 Tabelas de roteamento para a rede da Figura 3.7.

frame relay, como ATM, a rota é determinada quando o circuito virtual ou uma conexão é estabelecida. Nas redes baseadas em comutação de pacotes, como o IP, a rota é determinada individualmente para cada pacote, e o algoritmo deve ser particularmente simples e eficiente para não degradar o desempenho da rede.

2. Atualizar dinamicamente o conhecimento da topologia da rede com base no monitoramento do tráfego e na detecção de alterações de configuração ou falhas. Essa atividade é menos crítica quanto ao tempo, podendo ser usadas técnicas mais lentas e que utilizam mais poder de computação.

Essas duas atividades são distribuídas em toda a rede. As decisões de roteamento são tomadas em cada nó de roteamento, usando informações mantidas localmente, para determinar o próximo *hop* que um pacote deve ser reencaminhado. As informações de roteamento mantidas localmente são atualizadas periodicamente por um algoritmo que distribui informações sobre os estados dos enlaces (suas cargas e status de falha).

Um algoritmo simples de roteamento • O algoritmo que descrevemos aqui é o algoritmo de vetor de distância. Isso fornecerá a base para a discussão da Seção 3.4.3, sobre o algoritmo de *estado de enlace*, que é usado desde 1979 como o principal algoritmo de roteamento na Internet. O roteamento é um exemplo do problema de determinação de caminhos a serem percorridos em grafos. O algoritmo do menor caminho de Bellman, publicado bem antes que as redes de computadores fossem desenvolvidas [Bellman 1957], fornece a base do método do vetor de distância. O método de Bellman foi transformado por Ford e Fulkerson [1962] em um algoritmo distribuído, conveniente para implementação em redes de grande dimensão, e os protocolos baseados no trabalho deles são frequentemente referidos como protocolos Bellman-Ford.

A Figura 3.8 mostra as tabelas de roteamento que seriam mantidas em cada um dos nós de roteamento da rede da Figura 3.7, pressupondo que não exista nenhum enlace ou nó defeituoso. Cada linha da tabela fornece as informações de roteamento dos pacotes endere-

çados a um certo destino. O campo *Saída* especifica o enlace a ser utilizado para os pacotes alcançarem um determinado destino. O campo *Custo* é uma métrica a ser usada no cálculo do vetor de distância, nesse caso, corresponde ao número de nós intermediários (*hops*) até um dado destino. Essa métrica é comum em redes que possuem enlaces de largura de banda semelhante. As informações de custo armazenadas nas tabelas de roteamento não são usadas durante o roteamento de pacotes executadas pela parte 1 do algoritmo de roteamento mas são exigidas para as ações de construção e manutenção da tabela de roteamento pela parte 2.

As tabelas de roteamento contêm uma única entrada para cada destino possível, mostrando o próximo *passo* (*hop*) que um pacote deve dar em direção ao seu destino. Quando um pacote chega a um nó de roteamento, o endereço de destino é extraído e pesquisado na tabela de roteamento local. A entrada na tabela correspondente ao endereço de destino identifica o enlace de saída que deve ser usado para encaminhar o pacote para diante, em direção ao seu destino. O enlace de saída corresponde a uma interface de rede do nó.

Por exemplo, quando um pacote endereçado para C é recebido pelo nó A, o roteamento feito por este nó examina a entrada correspondente a C em sua tabela de roteamento. Ela mostra que o pacote deve ser direcionado para fora de A pela interface de saída (enlace) rotulada como 1. O pacote chega ao nó B e o mesmo procedimento é realizado. A tabela de roteamento do nó B mostra que a rota na direção de C é por meio da interface de saída rotulada como 2. Finalmente, quando o pacote chega ao nó C, sua entrada da tabela de roteamento mostra “local”, em vez de uma interface de saída. Isso indica que o pacote deve ser entregue para um *host* local a rede associada ao nó C.

Agora, vamos considerar como as tabelas de roteamento são construídas e mantidas quando ocorrem falhas na rede, ou seja, como é executada a parte 2 do algoritmo de roteamento descrito anteriormente. Como cada tabela de roteamento especifica apenas um único *hop* para cada rota, a construção ou atualização das informações de roteamento pode ocorrer de maneira distribuída. Um nó de roteamento troca informações sobre a rede com os nós de roteamento vizinhos enviando um resumo de sua tabela de roteamento, usando um *protocolo de informações de roteamento* (*RIP*, *Routing Information Protocol*). As ações do RIP feitas por um nó de roteamento são descritas, informalmente, a seguir:

1. *Periodicamente e quando a tabela de roteamento local muda*, um nó de roteamento envia sua tabela (em forma resumida) para todos os seus vizinhos imediatos que estão acessíveis. Isto é, envia através de cada interface de saída (enlace) não defeituosa um pacote RIP contendo uma cópia de sua tabela de roteamento.
2. *Quando uma tabela é recebida de um vizinho imediato*, o nó verifica se a tabela recebida possui uma rota para um novo destino ou uma rota melhor (de custo mais baixo) para um destino já existente. Em caso afirmativo, a tabela de roteamento local é atualizada com essa nova rota. Se a tabela recebida em um enlace *n* possuir um custo diferente para as rotas que iniciam em *n*, o custo armazenado na tabela é substituído por esse novo custo. Esse procedimento é feito porque a tabela recebida foi originada por um nó de roteamento mais próximo de um destino em questão e, portanto, possui informações mais exatas sobre uma determinada rota. Nesse caso, diz-se que este nó de roteamento tem *autoridade* sobre a rota.

O comportamento básico do algoritmo de vetor de distância é descrito pelo programa em pseudo-código mostrado na Figura 3.9, onde *Tr* é uma tabela recebida de um nó vizinho e *Tl* é a tabela local. Ford e Fulkerson [1962] mostraram que os passos descritos anteriormente são suficientes para garantir que as tabelas de roteamento converjam para obter as melhores rotas para cada destino, quando houver uma alteração na rede. A frequência *t* com que as tabelas de roteamento são propagadas, mesmo quando nenhuma alteração

Envia: a cada t segundos ou quando Tl mudar, envia Tl para todos enlaces não defeituosos.

Recebe: quando uma tabela de roteamento Tr é recebida no enlace n :

```

Para todas as linhas  $Rr$  em  $Tr$ {
    if ( $Rr.enlace \neq n$ ) {
         $Rr.custo = Rr.custo + 1;$ 
         $Rr.enlace = n;$ 
        if ( $Rr.destino$  não está em  $Tl$ ) adiciona  $Rr$  em  $Tl$ ; // adiciona novo destino em  $Tl$ 
        else para todas as linhas  $Rl$  em  $Tl$ {
            if ( $Rr.destino = Rl.destino$  e
                ( $Rr.custo < Rl.custo$  ou  $Rl.enlace = n$ ))  $Rl = Rr$ ;
                //  $Rr.custo < Rl.custo$ : o nó remoto tem uma rota melhor
                //  $Rl.link = n$ : o nó remoto é autoridade.
            }
        }
    }
}

```

Figura 3.9 Pseudo-código do algoritmo de roteamento RIP.

tiver ocorrido, é projetada de modo a garantir a estabilidade do algoritmo de roteamento; para, por exemplo, o caso de alguns pacotes RIP serem perdidos. O valor de t adotado pela Internet é de 30 segundos.

Para tratar das falhas, cada roteador monitora seus enlaces e atua como segue:

Quando um enlace defeituoso n *é detectado*, configura o custo como ∞ para todas as entradas da tabela local que se referem a esse enlace defeituoso e executa a ação *Envia*.

Assim, a informação de que o enlace está danificado é representada por um valor infinito do custo para os destinos que são alcançáveis por ele. Quando essa informação for propagada para os nós vizinhos, ela será processada de acordo com a ação *Recebe* (note que $\infty + 1 = \infty$) e, então, propagada novamente, até que seja atingido um nó que tenha uma rota que funcione para os destinos relevantes, caso haja uma. O nó que ainda tem a rota que funciona finalmente propagará sua tabela, e a rota que funciona substituirá a defeituosa em todos os nós.

O algoritmo do vetor de distância pode ser melhorado de várias maneiras: os custos (também conhecidos como *métricas*) podem ser baseados nas larguras de banda dos enlaces; o algoritmo pode ser modificado de forma a aumentar sua convergência e evitar que alguns estados intermediários indesejados, como laços de encaminhamento, ocorram antes que a convergência seja obtida. Um protocolo de informações de roteamento com esses aprimoramentos foi o primeiro protocolo de roteamento empregado na Internet, conhecido como RIP-1 e descrito na RFC 1058 [Hedrick 1988]. No entanto, as soluções para os problemas causados pela convergência lenta não são totalmente eficazes e isso leva a um roteamento ineficiente e à perda de pacotes enquanto a rede está em estados intermediários.

Existe uma série de desenvolvimentos subsequentes de algoritmos de roteamento, no sentido de aumentar o conhecimento da rede mantido em cada nó. A família mais importante dos algoritmos desse tipo são os *algoritmos de estado de enlace*. Eles são baseados na distribuição e na atualização de um banco de dados em cada nó, representando toda a rede ou uma parte substancial dela. Cada nó é, então, responsável por calcular as

rotas ótimas para os destinos mostrados em seu banco de dados. Esse cálculo pode ser feito por uma variedade de algoritmos, alguns dos quais evitam conhecidos problemas no algoritmo de Bellman-Ford, como a convergência lenta e os estados intermediários indesejáveis. O projeto de algoritmos de roteamento é um assunto muito importante e nossa discussão sobre eles aqui, é necessariamente limitada. Voltaremos a esse ponto na Seção 3.4.3, com uma descrição do funcionamento do algoritmo RIP-1, um dos primeiros utilizados para roteamento de IP e ainda em uso em muitas partes da Internet. Para uma abordagem ampla sobre roteamento na Internet, consulte Huitema [2000] e, para mais material sobre algoritmos de roteamento em geral, consulte Tanenbaum [2003].

3.3.6 Controle de congestionamento

A capacidade de uma rede é limitada pelo desempenho de seus enlaces de comunicação e nós de comutação. Quando a carga em um enlace, ou em um nó específico, aproximar-se de sua capacidade máxima, serão criadas filas de espera nos *hosts* que estão tentando enviar pacotes e nos nós intermediários que contêm pacotes cuja transmissão para diante está bloqueada por outro tráfego. Se a carga continuar no mesmo nível alto, as filas continuarão a crescer até atingirem o limite do espaço de *buffer* disponível.

Quando esse estado é atingido em um nó, o nó não tem outra opção a não ser descartar os novos pacotes recebidos. Conforme já mencionamos, a perda ocasional de pacotes na camada de rede é aceitável e pode ser corrigida por retransmissões iniciadas pelas camadas superiores. Contudo, se a taxa de perda e a retransmissão de pacotes atingirem um nível significativo, o efeito sobre o desempenho da rede poderá ser devastador. É fácil ver por que isso acontece: se pacotes são eliminados em nós intermediários, os recursos da rede que eles já consumiram são desperdiçados e as retransmissões resultantes exigirão uma quantidade de recursos semelhante para atingir o mesmo ponto na rede. Como regra geral, quando a carga em uma rede ultrapassa 80% de sua capacidade nominal, o desempenho de saída total tende a cair como resultado das perdas de pacote, a não ser que a utilização de enlaces muito sobrecarregados seja controlada.

Em vez de permitir que os pacotes trafeguem pela rede até atingirem nós congestionados, onde serão eliminados, seria melhor mantê-los em nós anteriores, até que o nível de congestionamento seja reduzido. Isso resultará em atrasos maiores para os pacotes, mas não degradará significativamente o desempenho total da rede. *Controle de congestionamento* é o nome dado às técnicas empregadas para se obter isso.

Em geral, o controle de congestionamento é feito informando-se os nós ao longo de uma rota de que ocorreu o congestionamento e que, portanto, sua taxa de transmissão de pacotes deve ser reduzida. Nos nós intermediários, isso implica armazenar os pacotes recebidos por um período maior. Para *hosts* que são origens de pacotes, o resultado pode ser o enfileiramento dos pacotes antes da transmissão ou o bloqueio do processo aplicativo que os está gerando, até que a rede possa manipulá-los.

Todas as camadas de rede baseadas em datagrama, incluindo IP e Ethernet, contam apenas com um controle de tráfego fim-a-fim, isto é, o nó emissor deve reduzir a taxa com que transmite pacotes com base somente na informação que recebe do destinatário. A informação de congestionamento é fornecida para o nó que está enviando os pacotes por meio da transmissão explícita de mensagens especiais (chamadas de *pacotes reguladores – choke packets*) solicitando uma redução na taxa de transmissão, ou pela implementação de um protocolo de controle de transmissão específico (do qual o TCP deriva seu nome – a Seção 3.4.6 explica o mecanismo usado no TCP) ou, ainda, pela observação da ocorrência de pacotes eliminados (caso o protocolo seja um no qual cada pacote é confirmado).

Em algumas redes baseadas em circuito virtual, a informação de congestionamento pode ser recebida e tratada em cada nó. Embora o ATM utilize circuito virtual, ele conta com gerenciamento de qualidade do serviço (veja o Capítulo 20) para garantir que cada circuito possa transmitir o tráfego exigido.

3.3.7 Interligação de redes

Existem muitas tecnologias de rede com diferentes protocolos de camadas de rede, enlace e física. As redes locais são construídas com tecnologias Ethernet, enquanto as redes de longa distância são construídas sobre redes telefônicas digitais e analógicas de vários tipos, enlaces de satélite e redes ATM. Os computadores individuais e as redes locais são ligadas à Internet ou às intranets por conexões de modems, sem fio e DSL.

Para construir uma rede integrada, ou seja, para interligar redes, devemos associar muitas sub-redes, cada uma baseada em uma dessas tecnologias de rede. Para tornar isso possível, é necessário:

1. Um esquema de endereçamento unificado de rede que permita aos pacotes serem endereçados para qualquer *host* conectado a qualquer sub-rede.
2. Um protocolo definindo o formato dos pacotes de rede e fornecendo regras de acordo com as quais eles são manipulados.
3. A interconexão componentes que direcionam pacotes para seus destinos, em termos de endereços unificados de rede, transmitindo os pacotes usando sub-redes com uma variedade de tecnologias de rede.

Para a Internet, (1) é fornecido por endereços IP, (2) é o protocolo IP e (3) é realizado por componentes chamados *roteadores Internet*. O protocolo IP e o endereçamento IP serão descritos com detalhes na Seção 3.4. Aqui, vamos descrever as funções dos roteadores Internet e outros componentes usados para integrar as redes.

A Figura 3.10 mostra uma pequena parte da intranet do campus de uma universidade britânica. Muitos dos detalhes mostrados serão explicados em seções posteriores. Aqui, notemos que a parte mostrada na figura compreende várias sub-redes interconectadas por roteadores. Existem cinco sub-redes; três das quais compartilham a rede IP 138.37.95.0 (usando o esquema de roteamento interdomínios sem classe – Classless InterDomain Routing –, descrito na Seção 3.4.3). Os números no diagrama são endereços IP; sua estrutura será explicada na Seção 3.4.1. Os roteadores no diagrama são membros de várias sub-redes e têm um endereço IP para cada sub-rede, mostrado nos enlaces de conexão.

Os roteadores (nomes de *host hammer* e *sickle*) são, na verdade, computadores de propósito geral que também cumprem outros objetivos. Um desses objetivos é servir como *firewalls*; a função de um *firewall* está intimamente ligada à função de roteamento, conforme vamos descrever a seguir. A sub-rede 138.37.95.232/29 não está conectada ao restante da rede em nível IP. Apenas o servidor de arquivos *custard* pode acessá-la para fornecer um serviço de impressão por intermédio de um processo que monitora e controla o uso das impressoras.

Todos os enlaces da Figura 3.10 são Ethernet. A largura de banda da maioria deles é de 100 Mbps, mas um é de 1.000 Mbps, pois transporta um alto volume de tráfego entre um grande número de computadores usados pelos alunos e o servidor de arquivos *custard*, que contém todos os arquivos deles.

Existem dois *switches* e vários hubs Ethernet na parte da rede ilustrada. Os dois tipos de componentes são transparentes para os datagramas IP. Um hub Ethernet é apenas um meio de conectar vários segmentos de cabo Ethernet, todos formando uma única rede

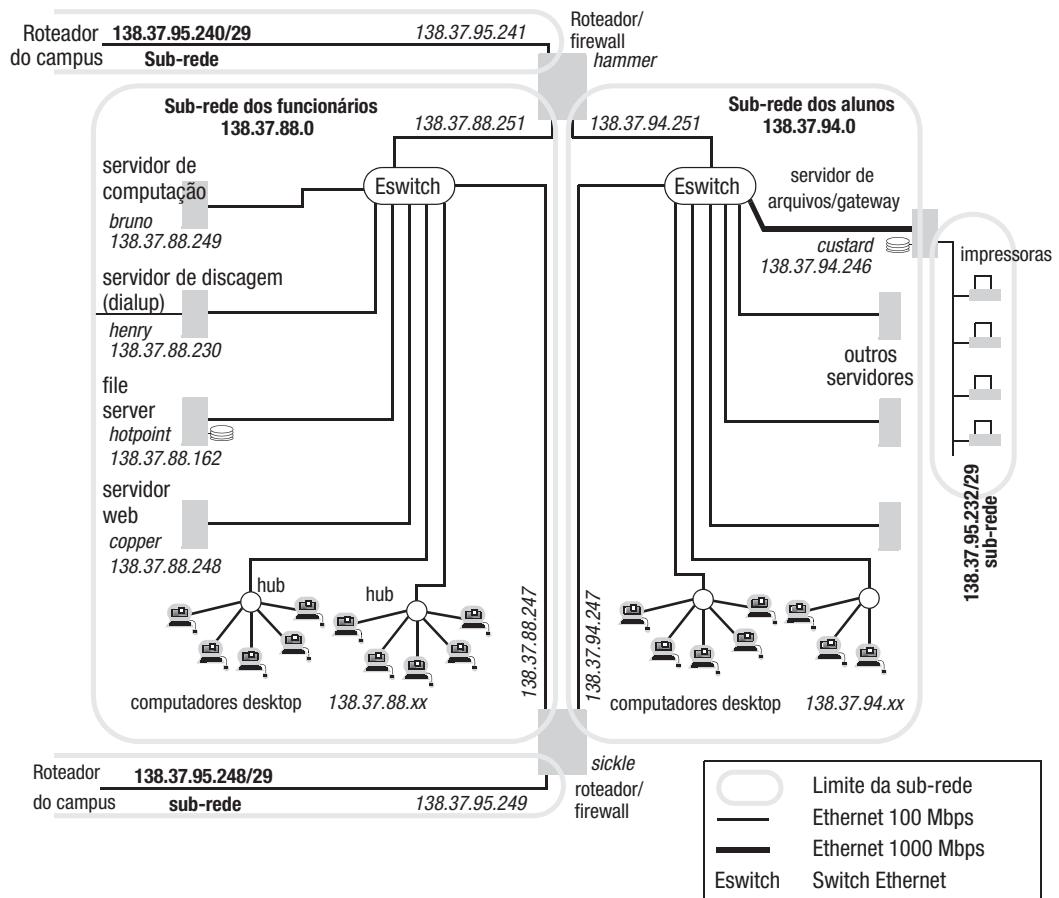


Figura 3.10 Visão simplificada de parte da rede de um campus universitário.

física Ethernet. Todos os pacotes Ethernet enviados por um *host* são retransmitidos para todos os segmentos. Um *switch* Ethernet conecta várias redes Ethernet, mas direciona os pacotes apenas para a rede Ethernet na qual o *host* de destino está conectado.

Roteadores • Mencionamos que o roteamento é exigido em todas as redes, exceto aquelas nas quais todos os *hosts* são conectados por um único meio de transmissão, como as Ethernet e as redes sem fio. A Figura 3.7 mostra uma rede com cinco roteadores conectados por seis enlaces. Na interligação de várias redes, os roteadores podem ser conectados diretamente entre si, como mostra a Figura 3.7, ou por meio de sub-redes, como mostrado para *custard* na Figura 3.10. Nos dois casos, os roteadores são responsáveis por encaminhar os pacotes de rede que chegam em um enlace qualquer para a saída apropriada. Conforme explicado anteriormente, eles mantêm tabelas de roteamento para esse propósito.

Pontes (bridges) • As pontes (*bridges*) ligam redes de diferentes tipos. Algumas pontes ligam várias redes e são denominadas pontes/roteadores, pois também executam funções de roteamento. Por exemplo, a rede mais ampla do campus inclui um *backbone* FDDI

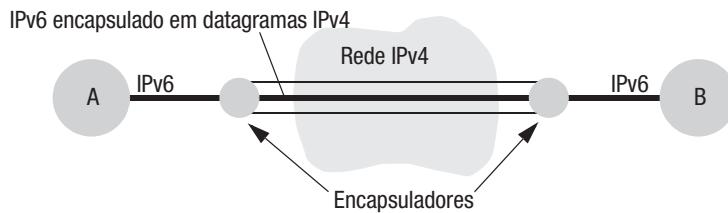


Figura 3.11 Utilização de túnel para migração para o IPv6.

(Fibre Distributed Data Interface) (não mostrado na Figura 3.10), e esse *backbone* está ligado às sub-redes da figura por meio de pontes/roteadores.

Hubs • *Hubs* são simplesmente uma maneira conveniente de conectar *hosts* e estender segmentos Ethernet e outras tecnologias de rede local baseadas em transmissão *broadcast*. Eles têm várias (comumente de 4 a 64) tomadas, ou portas, nas quais um computador *host* pode ser conectado. Eles também podem ser usados para estender o limite de distância de um segmento e fornecer um modo de acrescentar mais *hosts* na rede.

Switches • Os *switches* executam uma função semelhante aos roteadores, mas apenas para redes locais (normalmente, Ethernet). Isto é, eles interligam várias redes Ethernet separadas, direcionando os pacotes recebidos para a rede de saída apropriada. Eles executam sua tarefa no nível do protocolo de enlace Ethernet. Os *switches*, quando iniciam sua operação, não possuem conhecimento sobre as redes que interligam e começam a construir suas tabelas de roteamento observando o tráfego. Na falta da informação para onde direcionar um pacote, eles direcionam para todas as suas portas de saída. Isso é denominado *broadcast*.

A vantagem dos *switches* em relação aos hubs é que eles retransmitem o tráfego recebido apenas para a rede de saída relevante, reduzindo o congestionamento nas outras redes em que estão conectados.

Tunelamento (tunneling) • As pontes e os roteadores transmitem pacotes em uma variedade de redes subjacentes, adequando os pacotes ao tipo de tecnologia empregada por seus enlaces. No entanto, há uma situação em que, sem o uso de um protocolo de rede, a tecnologia de rede subjacente fica oculta das camadas que estão acima dele. Dois nós conectados em redes separadas, de um mesmo tipo, podem se comunicar por meio de uma rede de tipo diferente. Isso é possível por meio da construção de um “túnel” de protocolo. Um túnel de protocolo é uma camada de *software* que transmite pacotes em um ambiente de rede diferente daquele em que nativamente eles existem.

A analogia a seguir explica o motivo da escolha da terminologia e fornece outra maneira de pensar sobre o uso de túneis. Um túnel através de uma montanha permite a existência de uma estrada para transportar carros onde, de outro modo, isso seria impossível. A estrada é contínua – o túnel é transparente para a aplicação (carros). A estrada é o mecanismo de transporte, e o túnel permite que ela funcione em um ambiente estranho.

A Figura 3.11 ilustra o uso de túneis na migração da Internet para o protocolo IPv6. O protocolo IPv6 se destina a substituir a versão de IP ainda amplamente em uso, o IPv4, e ambos são incompatíveis entre si. (Tanto o protocolo IPv4 como o IPv6 serão descritos na Seção 3.4.) Durante o período de transição para o IPv6, haverá “ilhas” de interligação em rede IPv6 no mar do IPv4. Em nossa ilustração, A e B são essas ilhas. Nos limites das

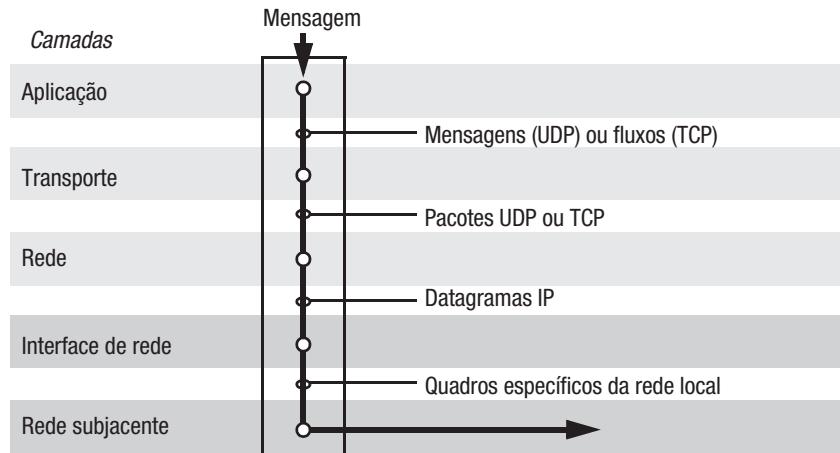


Figura 3.12 Camadas TCP/IP.

ilhas, os pacotes IPv6 são encapsulados em IPv4 e, dessa maneira, transportados pelas redes IPv4 intervenientes.

Como outro exemplo, o MobileIP (descrito na Seção 3.4.5) transmite datagramas IP para *hosts* móveis em qualquer parte da Internet, construindo um túnel para eles a partir de sua base de origem. Os nós das redes intervenientes não precisam ser modificados para manipular o protocolo MobileIP. O protocolo *multicast* IP opera de maneira semelhante. Para os roteadores que oferecem suporte a *multicast*, o protocolo explora essa capacidade e, para os demais, envia os pacotes de *multicast* encapsulando-os sobre o IP padrão. O protocolo PPP para a transmissão de datagramas IP sobre enlaces seriais é outro exemplo.

3.4 Protocolos Internet

Aqui, descreveremos os principais recursos da pilha de protocolos TCP/IP e discutiremos suas vantagens e limitações quando usados em sistemas distribuídos.

A Internet surgiu a partir de duas décadas de pesquisa e desenvolvimento sobre redes de longa distância nos Estados Unidos, começando no início dos anos 70 com a ARPANET – a primeira rede de computadores em larga escala desenvolvida [Leiner *et al.* 1997]. Uma parte importante dessa pesquisa foi o desenvolvimento da pilha de protocolos TCP/IP. TCP significa *Transmission Control Protocol* (protocolo de controle de transmissão) e IP é *Internet Protocol* (protocolo Internet). A ampla adoção do protocolo TCP/IP e dos protocolos de aplicação da Internet em redes de pesquisa em nível nacional e, mais recentemente, em redes comerciais de muitos países, permitiu a integração dessas em uma única rede que cresceu de forma extremamente rápida, até seu tamanho atual, com mais de 60 milhões de *hosts*. Atualmente, existem muitos serviços e protocolos em nível de aplicação baseados em TCP/IP, incluindo, entre outros, a Web (HTTP), *e-mail* (SMTP, POP), *netnews* (NNTP), transferência de arquivos (FTP) e Telnet (telnet). O TCP é um protocolo de transporte; ele pode ser usado para suportar aplicativos diretamente, ou protocolos adicionais podem ser dispostos em camadas sobre ele para fornecer recursos adicionais. Por exemplo, o protocolo HTTP é normalmente transportado diretamente

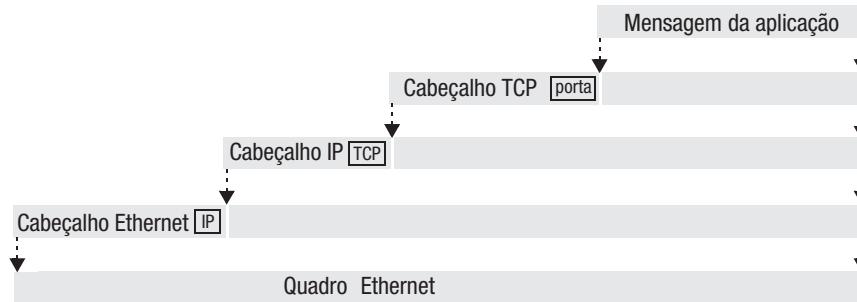


Figura 3.13 Encapsulamento que ocorre quando uma mensagem é transmitida via TCP sobre uma rede local Ethernet.

sobre o TCP, mas quando é exigida segurança fim-a-fim, o protocolo TLS (Transport Layer Security), descrito na Seção 11.6.3, é disposto em uma camada sobre o TCP para produzir canais seguros para transmitir as mensagens HTTP.

Originalmente, os protocolos Internet foram desenvolvidos principalmente para suportar aplicativos remotos simples, como transferências de arquivos e correio eletrônico, envolvendo comunicação com latências relativamente altas entre computadores geograficamente dispersos. No entanto, eles se mostraram eficientes o bastante em redes locais e de longa distância para suportar os requisitos de muitos aplicativos distribuídos, e agora são quase universalmente usados nos sistemas distribuídos. A padronização resultante dos protocolos de comunicação tem trazido imensos benefícios.

A ilustração geral das camadas de protocolos da Figura 3.6 é transformada, no caso específico da Internet, na da Figura 3.12. Existem dois protocolos de transporte – TCP (Transmission Control Protocol) e UDP (User Datagram Protocol). TCP é um protocolo confiável, orientado à conexão, e UDP é um protocolo baseado em datagrama que não garante uma transmissão confiável. O Internet Protocol (IP) é o protocolo da “rede virtual” Internet – isto é, datagramas IP fornecem o mecanismo de transmissão básico da Internet e de outras redes TCP/IP. Colocamos a palavra “rede” entre aspas na frase anterior porque ela não é a única camada envolvida na implementação da comunicação na Internet. Isso porque os protocolos Internet normalmente são dispostos em camadas sobre outras tecnologias de rede locais, como a Ethernet, a qual permite aos computadores ligados em uma mesma rede local trocarem pacotes de dados (quadros Ethernet). A Figura 3.13 ilustra o encapsulamento de dados que ocorre na transmissão de uma mensagem via TCP sobre uma rede Ethernet. Os cabeçalhos de uma camada possuem rótulos que indicam o tipo de protocolo da camada que está acima. Isso é necessário para que a pilha de protocolos do lado destino possa desempacotar os pacotes corretamente. Na camada TCP, o número da porta destino tem um propósito semelhante: permitir que o módulo de *software* TCP no *host* de destino repasse a mensagem para um processo específico da camada de aplicação.

As especificações do protocolo TCP/IP [Postel 1981a; 1981b] não definem as camadas inferiores à camada do datagrama Internet. Para sua transmissão, os datagramas IP são apropriadamente formatados nos quadros de dados das tecnologias de redes locais ou enlaces subjacentes.

Por exemplo, a princípio, o protocolo IP era executado na ARPANET, que consistia em *hosts* e uma versão inicial de roteadores (chamados PSEs), conectados por enlaces de



Figura 3.14 Visão conceitual de um programador de uma rede TCP/IP.

dados de longa distância. Atualmente, o IP é usado em praticamente todas as tecnologias de rede conhecidas, incluindo ATM, redes locais, como Ethernet, e redes *token ring*. O protocolo IP é implementado sobre linhas seriais e circuitos telefônicos por intermédio do protocolo PPP [Parker 1992], permitindo ser utilizado em comunicações via modem e outros canais seriais.

O sucesso do protocolo TCP/IP deve-se à sua independência em relação à tecnologia de transmissão subjacente, o que permite a interligação de muitas redes e enlaces de dados heterogêneos. Os usuários e os programas aplicativos percebem uma única rede virtual suportando TCP e UDP, e os desenvolvedores de programas baseados em TCP e UDP veem uma única rede IP virtual, ocultando a diversidade da mídia de transmissão subjacente. A Figura 3.14 ilustra essa visão.

Nas próximas duas seções, descreveremos o esquema de endereçamento e o protocolo IP. O Domain Name System, que converte em endereços IP os nomes de domínio como *www.amazon.com*, *hpl.hp.com*, *stanford.edu* e *qmw.ac.uk*, com os quais os usuários da Internet estão tão familiarizados, será apresentado na Seção 3.4.7 e descrito de forma mais completa no Capítulo 13.

A versão predominante por toda a Internet é o IPv4 (desde janeiro de 1984) e essa é a versão que vamos descrever nas duas próximas seções. Porém, o rápido crescimento da Internet levou à publicação da especificação de uma nova versão, o IPv6, para superar as limitações de endereçamento do IPv4 e acrescentar recursos para suportar alguns novos requisitos. Descreveremos a versão IPv6 na Seção 3.4.4. Devido à enorme quantidade de *software* que será afetada, uma migração gradual para a versão IPv6 está sendo planejada para um período de dez anos ou mais.

3.4.1 Endereçamento IP

Talvez o aspecto mais desafiador do projeto dos protocolos Internet tenha sido a construção de esquemas para a atribuição de endereços a *hosts* e de roteamento dos datagramas IP para seus destinos. O esquema usado para atribuição de endereços para redes e para os *hosts* nelas conectados tinha de satisfazer os seguintes requisitos:

- Ser universal – qualquer *host* deveria conseguir enviar pacotes para qualquer outro *host* na Internet.
- Ser eficiente no uso do espaço de endereçamento – é impossível prever o tamanho final da Internet e o número de endereços de rede e de *hosts* que provavelmente será exigido. O espaço de endereçamento deveria ser cuidadosamente particionado para garantir que os endereços não acabassem. Em 1978-1982, quando as especificações dos protocolos TCP/IP estavam sendo desenvolvidas, o aprovisionamento de 2^{32} de endereços, ou seja, aproximadamente 4 bilhões (praticamente

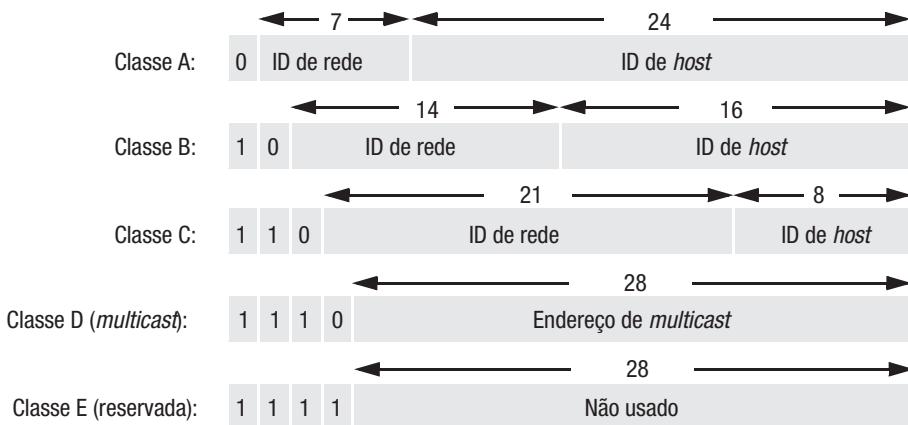


Figura 3.15 Estrutura de endereços Internet, mostrando os tamanhos dos campos em bits.

igual à população mundial da época), era considerado adequado. Esse julgamento se mostrou imprevidente por dois motivos:

- a taxa de crescimento da Internet superou, em muito, todas as previsões;
- o espaço de endereços foi alocado e usado de forma muito menos eficiente do que o esperado.
- Servir para o desenvolvimento de um esquema de roteamento flexível e eficiente, mas sem que os endereços em si contivessem toda a informação necessária para direcionar um datagrama até seu destino.

Atualmente, a maioria esmagadora do tráfego da Internet continua a usar a versão 4 do formato de endereçamento e pacotes IP, definido há quatro décadas. O esquema atribui um endereço IP para cada *host* na Internet – um identificador numérico de 32 bits, contendo um identificador de rede, que identifica exclusivamente uma sub-rede na Internet, e um identificador de *host*, que identifica exclusivamente um *host* nessa rede. São esses os endereços colocados nos datagramas IP e usados para direcioná-los até seus destinos.

O projeto adotado para o espaço de endereçamento da Internet aparece na Figura 3.15. Existem quatro classes alocadas de endereços Internet – A, B, C e D. A classe D é reservada para comunicação *multicast* na Internet. Atualmente, ela é suportada apenas em alguns roteadores e será melhor discutida na Seção 4.4.1. A classe E contém um intervalo de endereços não alocados, os quais estão reservados para requisitos futuros.

Os endereços Internet, que são um número em 32 bits, e que contêm um identificador de rede e um identificador de *host*, são normalmente escritos como uma sequência de quatro números decimais separados por pontos. Cada número decimal representa um dos quatro bytes, ou *octetos*, do endereço IP. Os valores permitidos para cada classe de endereço de rede aparecem na Figura 3.16.

Três classes de endereço foram projetadas para satisfazer os requisitos de diferentes tipos de organização. Os endereços de classe A, com uma capacidade de 2^{24} hosts em cada sub-rede, são reservados para redes muito grandes, como a US NSFNet e outras redes de longa distância em nível nacional. Os endereços de classe B são alocados para organizações que possuem redes que provavelmente conterão mais de 255 computadores, e os endereços da classe C são usados nos demais tipos de redes.

	<i>Octeto 1</i>	<i>Octeto 2</i>	<i>Octeto 3</i>	<i>Intervalo de endereçamento</i>
Classe A:	<i>ID de rede</i> 1 a 127	0 a 255	<i>ID de host</i> 0 a 255	1.0.0.0 a 127.255.255.255
Classe B:	<i>ID de rede</i> 128 a 191	0 a 255	<i>ID de host</i> 0 a 255	128.0.0.0 a 191.255.255.255
Classe C:	<i>ID de rede</i> 192 a 223	0 a 255	<i>ID de host</i> 0 a 255	192.0.0.0 a 223.255.255.255
Classe D (<i>multicast</i>):	<i>Endereço de multicast</i>			224.0.0.0 a 239.255.255.255
Classe E (reservada):	240 a 255	0 a 255	0 a 255	240.0.0.0 a 255.255.255.255

Figura 3.16 Representação decimal dos endereços Internet.

Os endereços Internet com identificadores de *host* com todos os seus bits em 0 ou em 1 (binário), são usados para propósitos especiais. Os endereços com identificador de *host* igual a 0 são usados para se referenciar *esta rede*. Aqueles com o identificador de *host* com todos seus bits igual a 1 são usados para endereçar uma mensagem para todos os *hosts* conectados na rede especificada na parte do identificador de rede do endereço. Esse endereço é denominado endereço de *broadcast*.

Para as organizações com redes conectadas à Internet, os identificadores de rede são alocados pela IANA (Internet Assigned Numbers Authority). Os identificadores de *host* para os computadores de cada rede são atribuídos pelo administrador da rede em questão.

Como os endereços IP incluem um identificador de rede, qualquer computador que esteja conectado em mais de uma rede deve ter endereços separados em cada uma delas, e quando um computador é movido para uma rede diferente, seu endereço IP deve ser alterado. Esses requisitos podem levar a sobrecargas administrativas substanciais; por exemplo, no caso dos computadores portáteis.

Na prática, o esquema de alocação de endereço IP não se mostrou muito eficiente. A principal dificuldade é que os administradores de rede das diversas organizações não podem prever facilmente o crescimento futuro de suas necessidades de endereços IP e tendem a superestimá-las, solicitando endereços de Classe B, quando estão em dúvida. Por volta de 1990, tornou-se evidente que, com base na taxa de alocação da época, os endereços IP provavelmente estariam esgotados em 1996. Três medidas foram tomadas. A primeira foi iniciar o desenvolvimento de um novo esquema de protocolo e endereçamento IP, cujo resultado foi a especificação do IPv6. A segunda medida foi modificar radicalmente a maneira pela qual os endereços IP eram alocados. Foi introduzido um novo esquema de alocação de endereços e roteamento, projetado para fazer uso mais eficiente do espaço de endereços IP, chamado CIDR (Classless Interdomain Routing – roteamento entre domínios sem classes). Descreveremos o CIDR na Seção 3.4.3. A rede local ilustrada na Figura 3.10 inclui várias sub-redes de classe C, no intervalo 138.37.88.0 – 138.37.95.0, ligadas por roteadores. Os roteadores gerenciam a distribuição de datagramas IP para todas as sub-redes e manipulam o tráfego delas para o resto do mundo. A figura também ilustra o uso de CIDR para subdividir um espaço de endereços de classe B para produzir várias sub-redes classe C.

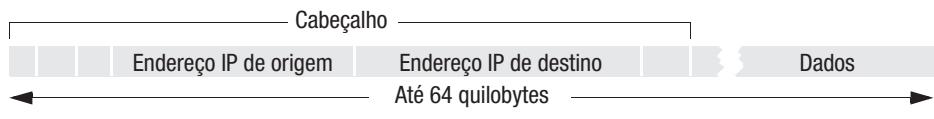


Figura 3.17 Leiaute de um datagrama IP.

A terceira medida foi permitir que computadores não acessassem a Internet diretamente, mas por meio de dispositivos que implementam um esquema NAT (Network Address Translation). Descreveremos esse esquema na Seção 3.4.3.

3.4.2 O protocolo IP

O protocolo IP transmite datagramas de um equipamento para outro, se necessário por meio de roteadores intermediários. O formato completo do datagrama IP é bastante complexo, mas a Figura 3.17 mostra os principais componentes. Existem vários campos de cabeçalho, não mostrados na figura, que são usados pelos algoritmos de transmissão e roteamento.

O protocolo IP fornece um serviço de entrega que oferece uma semântica descrita como *não confiável* ou de *melhor esforço (best effort delivery)*, pois não há garantia de entrega dos datagramas. Os datagramas podem ser perdidos, duplicados, retardados ou entregues fora de ordem, mas esses erros surgem apenas quando as redes subjacentes falham ou os *buffers* do destino estão cheios. O IP possui uma única soma de verificação que é feita sobre os bytes que formam seu cabeçalho, cujo cálculo não é dispendioso e garante que quaisquer modificações nos endereços IP fonte/destino ou nos dados de controle do datagrama sejam detectadas. Não existe nenhuma soma de verificação na área de dados, o que evita sobrecargas ao passar por roteadores, deixando os protocolos de nível mais alto (TCP e UDP) fornecerem as próprias somas de verificação – um exemplo prático do princípio fim-a-fim (Seção 2.3.3).

A camada IP coloca datagramas IP em um formato conveniente para transmissão na rede subjacente (a qual poderia, por exemplo, ser uma Ethernet). Quando um datagrama IP é maior que o MTU da rede subjacente, ele é dividido, na origem, em datagramas menores e novamente montado em seu destino final. Esse procedimento de divisão se denomina segmentação ou fragmentação. Os datagramas resultantes podem ser subdivididos ainda mais, para estarem de acordo com as redes subjacentes encontradas durante a jornada da origem até o destino. (Cada fragmento tem um identificador e um número de sequência para permitir a remontagem correta.)

A camada de rede deve obter um endereço físico do destino da mensagem na rede subjacente. Ela consegue isso por meio do módulo de resolução de endereços que será descrito a seguir.

Resolução de endereços • O módulo de resolução de endereços é responsável por converter endereços Internet em endereços físicos para uma rede subjacente específica. Por exemplo, se a rede subjacente for Ethernet, o módulo de resolução de endereços obtém o endereço Ethernet (48 bits) associado a um dado endereço IP (32 bits).

Essa transformação depende da tecnologia da rede:

- Quando os *hosts* são conectados diretamente, ponto a ponto; os datagramas IP podem ser direcionados para eles sem a transformação de endereço.
- Algumas redes locais permitem que endereços físicos de rede sejam atribuídos dinamicamente aos *hosts*, e esses endereços podem ser convenientemente esco-

lhidos para corresponder à parte do identificador de *host* do endereço de Internet – a resolução é simplesmente uma questão de extrair o identificador de *host* do endereço IP.

- Para redes Ethernet, e algumas outras redes locais, o endereço físico de rede de cada *host* é incorporado no *hardware* de sua interface de rede e não tem nenhuma relação direta com seu endereço IP – a resolução depende do conhecimento da correspondência entre endereços IP e endereços físicos dos *hosts* na rede local e é feita usando um protocolo de resolução de endereços (ARP, Address Resolution Protocol).

Agora, vamos esboçar a implementação do ARP para redes Ethernet. Ele é baseado na troca de mensagens de requisição-resposta, mas explora o uso de cache para minimizar esse tráfego. Considere, primeiramente, o caso em que um computador *host* conectado a uma rede Ethernet utiliza IP para transmitir uma mensagem para outro computador na mesma rede Ethernet. O módulo de *software* IP no computador remetente deve transformar o endereço IP do destino em um endereço Ethernet antes que o pacote possa ser enviado. Para fazer isso, ele ativa o módulo ARP no computador remetente.

O módulo ARP em cada *host* mantém uma cache de pares (*endereço IP*, *endereço Ethernet*) obtidos anteriormente. Se o endereço IP exigido está na cache, então a consulta é respondida imediatamente. Caso contrário, o módulo ARP transmite, em *broadcast* Ethernet, uma mensagem de requisição ARP na rede Ethernet local, contendo o endereço IP desejado. Cada um dos computadores da rede local Ethernet recebe o quadro de requisição ARP e verifica se o endereço IP nele contido corresponde ao seu próprio endereço IP. Se corresponder, uma mensagem de resposta ARP é enviada para a origem da requisição ARP, contendo o endereço Ethernet do remetente; caso contrário, a mensagem de requisição ARP é ignorada. O módulo ARP da origem adiciona o novo mapeamento *endereço IP* → *endereço Ethernet* em sua cache local de pares (*endereço IP*, *endereço Ethernet*), para que seja possível responder a pedidos semelhantes no futuro, sem propagar uma nova requisição ARP. Com o passar do tempo, a cache ARP de cada computador conterá o par (*endereço IP*, *endereço Ethernet*) de todos os computadores para os quais são enviados datagramas IP. Assim, o *broadcast* ARP só será necessário quando um computador tiver sido recentemente conectado na Ethernet local.

Spoofing de IP • Já vimos que os datagramas IP incluem um endereço de origem – o endereço IP do computador remetente – junto a um endereço de porta encapsulado no campo de dados (para mensagens UDP e TCP), os quais são usados pelos servidores para gerar um endereço de retorno. Infelizmente, não é possível garantir que o endereço de origem informado seja de fato o endereço do remetente. Um remetente mal-intencionado pode substituir facilmente seu endereço por um diferente do seu próprio. Essa brecha tem sido a fonte de vários ataques conhecidos, incluindo os ataques de negação de serviço (DoS, Denial of Service), de fevereiro de 2000 [Farrow 2000], mencionado no Capítulo 1, Seção 1.4.3. O método usado foi efetuar muitos pedidos do serviço *ping* para um grande número de computadores em diversos *sites* (o *ping* é um serviço simples, projetado para verificar a disponibilidade de um *host*). Todos esses pedidos mal-intencionados de *ping* continham o endereço IP de um computador alvo em seu campo de endereço do remetente. Portanto, as respostas do *ping* foram todas direcionadas para o alvo, cujos *buffers* de entrada ficaram sobrecarregados, impedindo a passagem de datagramas IP legítimos. Esse ataque será melhor discutido no Capítulo 11.

3.4.3 Roteamento IP

A camada IP direciona os datagramas de sua origem para seu destino. Cada roteador na Internet possui pelo menos um tipo de algoritmo de roteamento implementado.

Backbones • A topologia da Internet é conceitualmente particionada em *sistemas autônomos* (SA), os quais são subdivididos em *áreas*. As intranets da maioria das grandes organizações, como universidades e empresas, são consideradas SAs e, normalmente, incluem várias áreas. Na Figura 3.10, a intranet do campus é um SA e a parte mostrada é uma área. Todo SA tem uma área de *backbone*. O conjunto de roteadores que conectam áreas que não são *backbone* com o *backbone* e os enlaces que interconectam esses roteadores são chamados de *backbone* da rede. Os enlaces do *backbone* normalmente têm largura de banda alta e, por questão de confiabilidade, são replicados. Essa estrutura hierárquica é conceitual e explorada principalmente no gerenciamento de recursos e manutenção dos componentes. Ela não afeta o roteamento de datagramas IP.

Protocolos de roteamento • O RIP-1, o primeiro algoritmo de roteamento usado na Internet, é uma versão do algoritmo de vetor de distância descrito na Seção 3.3.5. O RIP-2 (descrito na RFC 1388 [Malkin 1993]) foi desenvolvido subsequentemente a partir dele para acomodar vários requisitos adicionais, incluindo roteamento CIDR, melhor suporte a roteamento *multicast* e a necessidade de autenticação de mensagens RIP para evitar ataques aos roteadores.

Como a escala da Internet se expandiu e a capacidade de processamento dos roteadores aumentou, houve uma mudança no sentido de adotar algoritmos que não sofram com os problemas de convergência lenta e da instabilidade potencial dos algoritmos de vetor de distância. A direção dessa mudança é a classe de algoritmos de estado de enlace, mencionados na Seção 3.3.5, e o algoritmo chamado de *caminho mais curto primeiro* (*OSPF, Open Shortest Path First*). A letra O é originada do fato de que esse protocolo segue uma filosofia de padrão aberto (*open*). O protocolo OSPF é baseado em um algoritmo de descoberta de caminho de Dijkstra [1959], e tem mostrado convergir mais rapidamente do que o algoritmo RIP.

Devemos notar que a adoção de novos algoritmos de roteamento em roteadores IP pode ocorrer paulatinamente. Uma mudança no algoritmo de roteamento resulta em uma nova versão do protocolo RIP, e o número de versão é identificado em cada mensagem RIP. O protocolo IP não muda quando uma nova versão do protocolo RIP é introduzida. Qualquer roteador IP encaminhará corretamente os datagramas IP recebidos por uma rota razoável, senão ótima, qualquer que seja a versão de RIP utilizada. No entanto, para os roteadores cooperarem na atualização de suas tabelas de roteamento, eles devem compartilhar um algoritmo semelhante. Para esse propósito, são usadas as áreas definidas anteriormente. Dentro de cada área, é aplicado um único algoritmo de roteamento e os roteadores cooperam na manutenção de suas tabelas de roteamento. Os roteadores que suportam apenas RIP-1 ainda são comuns e coexistem com os roteadores que suportam RIP-2 e OSPF, usando recursos de compatibilidade com versões anteriores incorporados nos protocolos mais recentes.

Em 1993, observações empíricas [Floyd e Jacobson 1993] mostraram que a frequência de 30 segundos com que os roteadores RIP trocam informações estava produzindo uma periodicidade no desempenho das redes IP. A latência média para transmissões de datagramas IP mostrava um pico em intervalos de 30 segundos. Isso aconteceu devido ao comportamento dos roteadores que executavam o protocolo RIP – ao receber uma mensagem RIP, os roteadores retardavam a transmissão de todos os datagramas IP que

mantinham, até que o processo de atualização da tabela de roteamento estivesse concluído para todas as mensagens RIP recebidas até o momento. Isso tendia a fazer com que os roteadores executassem as ações do RIP lentamente. A correção recomendada foi fazer com que os roteadores adotassem um valor aleatório no intervalo de 15 a 45 segundos para o período de atualização do RIP.

Rotas padrão (default) • Até agora, nossa discussão sobre algoritmos de roteamento sugeriu que cada roteador mantém uma tabela de roteamento completa, mostrando a rota para cada destino (sub-rede ou *host* diretamente conectado) na Internet. Na escala atual da Internet, isso é claramente impraticável (o número de destinos provavelmente já ultrapassa 1 milhão e está crescendo muito rapidamente).

Duas soluções possíveis para esse problema vêm à mente e ambas foram adotadas em um esforço para atenuar os efeitos do crescimento da Internet. A primeira solução foi adotar alguma forma de agrupamento de endereços IP. Antes de 1993, a partir de um endereço IP, nada podia ser inferido sobre sua localização. Em 1993, como parte da mudança visando à simplificação e à economia na alocação de endereços IP, que será discutida mais adiante, quando estudarmos o CIDR, decidiu-se que para as futuras alocações seriam aplicadas as seguintes regras:

Os endereços de 194.0.0.0 a 195.255.255.255 ficariam na Europa.

Os endereços de 198.0.0.0 a 199.255.255.255 ficariam na América do Norte.

Os endereços de 200.0.0.0 a 201.255.255.255 ficariam na América Central e do Sul.

Os endereços de 202.0.0.0 a 203.255.255.255 ficariam na Ásia e no Pacífico.

Como essas regiões geográficas também correspondem a regiões topológicas bem definidas na Internet, e apenas alguns roteadores *gateway* fornecem acesso a cada região, isso permite uma simplificação substancial das tabelas de roteamento para esses intervalos de endereço. Por exemplo, um roteador fora da Europa pode ter uma única entrada na tabela para o intervalo de endereços 194.0.0.0 a 195.255.255.255. Isso faz com que ele envie todos os datagramas IP com destinos nesse intervalo, em uma mesma rota, para o roteador *gateway* europeu mais próximo. Porém, note que antes da data dessa decisão, os endereços IP eram alocados liberalmente, sem respeitar a topologia ou a geografia. Muitos desses endereços ainda estão em uso e a decisão de 1993 nada fez para reduzir a escala das entradas na tabela de roteamento para esses endereços.

A segunda solução para a explosão no tamanho da tabela de roteamento é mais simples e muito eficiente. Ela é baseada na observação de que a precisão da informação de roteamento pode ser relaxada para a maioria dos roteadores, desde que alguns roteadores-chave, aqueles mais próximos aos enlaces do *backbone*, tenham tabelas de roteamento relativamente completas. Esse relaxamento de precisão assume a forma de uma entrada de destino *padrão (default)* nas tabelas de roteamento. A entrada padrão especifica uma rota a ser usada por todos os datagramas IP cujo destino não está incluído na tabela de roteamento. Para ilustrar isso, considere as Figuras 3.7 e 3.8, e suponha que a tabela de roteamento do nó C seja alterada para mostrar:

Tabela de roteamento nó C		
Para	Enlace	Custo
B	2	1
C	local	0
E	5	1
Padrão	5	-

Assim, o nó C não sabe da existência dos nós A e D. Ele direcionará todos os pacotes endereçados a eles para E, por meio do enlace 5. Como consequência, os pacotes endereçados a D atingirão seu destino sem perda de eficiência no roteamento, mas os pacotes endereçados a A farão um salto extra, passando por E e B no caminho. O uso de rotas padrão compensa a perda de eficiência com a redução do tamanho da tabela de roteamento. No entanto, em alguns casos, especialmente naqueles em que um roteador é ponto de passagem obrigatória, não há perda de eficiência. O esquema de rota padrão é bastante usado na Internet; nela, nenhum roteador contém, sozinho, as rotas para todos os destinos.

Roteamento em uma sub-rede local • Os pacotes endereçados para *hosts* que estão na mesma rede que o remetente são transmitidos para o *host* destino em um único salto, usando a parte do endereço relativa ao identificador de *host* para obter o endereço físico de rede do *host* destino na rede subjacente. A camada IP simplesmente usa o ARP para obter o endereço físico de rede do destino e, depois, utiliza a rede subjacente para transmitir os pacotes.

Se a camada IP no computador remetente descobrir que o destino está em uma rede diferente, ela deve enviar a mensagem para o *gateway* da rede local. Ela usa o ARP para obter o endereço físico de rede do *gateway*, normalmente um roteador, e depois usa a rede subjacente para transmitir o datagrama IP para ele. Os *gateways* são conectados em duas ou mais redes e têm vários endereços IP, um para cada rede em que estão ligados.

Roteamento interdomínios sem classes (CIDR – Classless InterDomain Routing) • O esgotamento de endereços IP, mencionado na Seção 3.4.1, levou à introdução, em 1996, desse novo esquema de alocação de endereços e gerenciamento das entradas nas tabelas de roteamento. O problema principal era a escassez de endereços de classe B – para sub-redes com mais de 254 *hosts* conectados. Muitos endereços de classe C estavam disponíveis. A solução CIDR para esse problema é permitir a alocação de um lote de endereços de classe C adjacentes para uma sub-rede que exija mais de 254 endereços. O esquema CIDR também torna possível subdividir um espaço de endereços de classe B para alocação para várias sub-redes.

Dispor de endereços de classe C em lotes parece uma medida simples, mas apenas se for acompanhada de uma alteração no formato da tabela de roteamento; caso contrário, haverá um impacto substancial no tamanho das tabelas e, portanto, na eficiência dos algoritmos que as gerenciam. A mudança adotada foi adicionar um campo de *máscara* nas tabelas de roteamento. A máscara é um padrão de bits usado para selecionar a parte de um endereço IP que é comparada à entrada da tabela de roteamento. Isso permite, efetivamente, que o endereço de rede, e de sub-rede, ocupe qualquer porção de um endereço IP, proporcionando mais flexibilidade do que as classes A, B e C. Daí o nome roteamento entre domínios *sem classes*. Mais uma vez, essas alterações nos roteadores são feitas paulatinamente, de modo que alguns roteadores executam o esquema CIDR e outros usam os antigos algoritmos baseados em classe.

Isso funciona porque os novos intervalos de endereços são sempre alocados em módulo de 256, portanto cada novo intervalo é sempre um múltiplo inteiro de uma rede classe C. Por outro lado, algumas sub-redes também fazem uso de CIDR para subdividir o intervalo de endereços em uma única rede de classe A, B ou C. Se um conjunto de sub-redes estiver conectado com o resto do mundo inteiramente através de roteadores CIDR, então os intervalos de endereços IP, usados dentro desse conjunto, poderão ser alocados em blocos de diferentes tamanhos. Isso é feito empregando-se máscaras de tamanho variável.

Por exemplo, um espaço de endereços de classe C pode ser subdividido em 32 grupos de 8 endereços. A Figura 3.10 apresenta um exemplo do uso do mecanismo CIDR

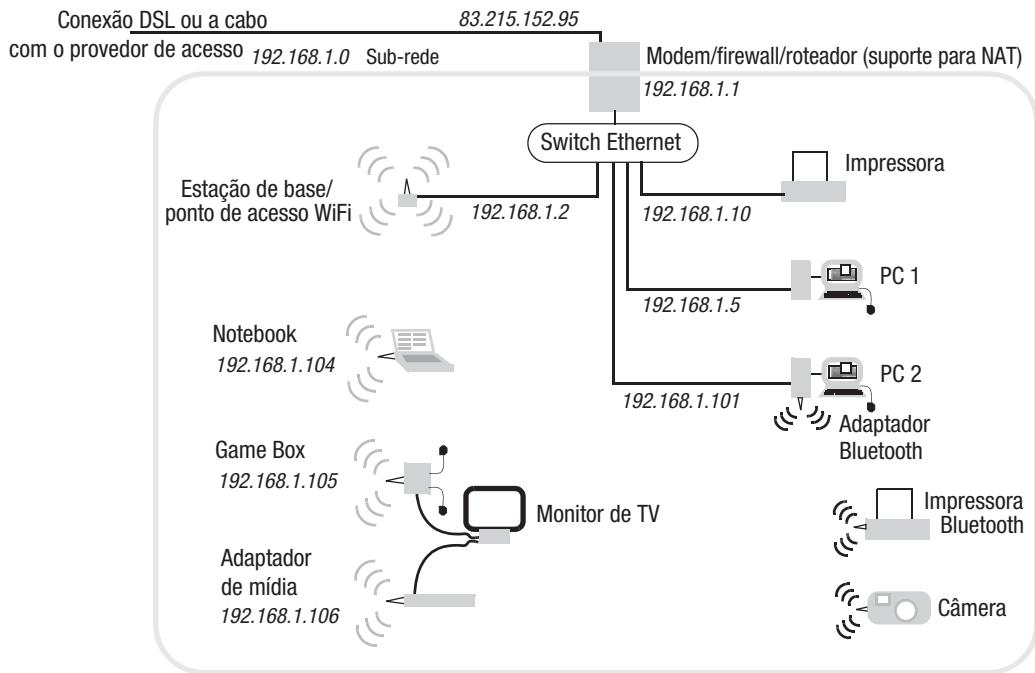


Figura 3.18 Uma rede doméstica típica baseada em NAT.

para dividir a sub-rede 138.37.95.0 (classe C) em vários grupos de oito endereços rotteados separadamente. Os grupos são designados pelas notações 138.37.95.232/29, 138.37.95.248/29, etc. A parte /29 desses endereços indica uma máscara binária de 32 bits composta pelos 29 bits mais significativos em 1 e os três últimos em zero.

Endereços não registrados e NAT (Network Address Translation) • Nem todos os computadores e dispositivos que acessam a Internet precisam receber endereços IP globalmente exclusivos. Os computadores que estão ligados a uma rede local e acessam a Internet por meio de um roteador com suporte a NAT podem usá-lo para enviar e receber mensagens UDP e TCP. A Figura 3.18 ilustra uma rede doméstica típica, com computadores e outros dispositivos de rede ligados à Internet por meio de um roteador capaz de realizar NAT. A rede inclui computadores diretamente conectados no roteador por meio de uma conexão Ethernet cabeadas, assim como outros equipamentos conectados por meio de um ponto de acesso WiFi. Genericamente, são mostrados alguns dispositivos compatíveis com o padrão Bluetooth, mas eles não estão conectados no roteador e, assim, não podem acessar a Internet diretamente. A rede doméstica recebeu um único endereço IP registrado (83.215.152.95) de seu provedor de serviços de Internet. A estratégia descrita aqui é conveniente para qualquer organização que queira conectar à Internet computadores sem endereços IP registrados.

Todos os dispositivos da rede doméstica com conexão à Internet receberam endereços IP não registrados na sub-rede classe C 192.168.1.0. A maior parte dos computadores e dispositivos internos recebem endereços IP individuais, dinamicamente, por um serviço DHCP (Dynamic Host Configuration Protocol) executado no roteador. Em nossa ilustração, os números acima de 192.168.1.100 são fornecidos automaticamente pelo serviço

DHCP, e os endereços menores (como o PC 1) foram alocados manualmente, por um motivo explicado posteriormente nesta subseção. Embora todos esses endereços fiquem completamente ocultos do restante da Internet pelo roteador NAT, é convenção usar um intervalo de endereços de um de três blocos de endereços (10.0.0.0/8, 172.192.0.0/12 e 192.168.0.0/16) reservados pelo IANA para redes privadas, também denominadas não registradas, não roteadas ou, ainda, falsas.

O NAT está descrito na RFC 1631 [Egevang e Francis 1994] e foi ampliado na RFC 2663 [Srisuresh e Holdrege 1999]. Os roteadores com suporte a NAT mantêm uma tabela de mapeamento de endereços e empregam os campos de número de porta de origem e de destino das mensagens UDP e TCP para atribuir, a cada mensagem de resposta recebida, o computador interno que enviou a mensagem de solicitação correspondente. Note que a porta de origem fornecida em uma mensagem de solicitação é sempre usada como porta de destino na mensagem de resposta correspondente.

A variante mais comumente usada em NAT funciona da seguinte forma:

- Quando um computador da rede interna envia uma mensagem UDP ou TCP para um computador fora dela, o roteador recebe a mensagem e salva o endereço IP e o número de porta da origem em uma entrada disponível em sua tabela de mapeamento de endereços.
- O roteador substitui o endereço IP de origem pelo seu endereço IP, e a porta de origem por um número de porta virtual que indexa a entrada da tabela que contém a informação de endereço do computador remetente.
- O pacote com o endereço de origem e o número de porta modificados é, então, encaminhado para seu destino pelo roteador. Agora, a tabela de mapeamento de endereços contém uma associação do número de porta virtual para o endereço IP e número de porta internos reais para todas as mensagens enviadas pelos computadores da rede interna.
- Quando o roteador recebe uma mensagem UDP ou TCP de um computador externo, ele usa o número de porta de destino presente na mensagem para acessar uma entrada na tabela de mapeamento de endereços. Ele substitui o endereço de destino e a porta de destino, presentes na mensagem recebida, pelos armazenados na entrada e encaminha a mensagem modificada para o computador interno identificado pelo endereço de destino.

O roteador mantém o mapeamento de porta enquanto ela estiver em uso. Um temporizador é zerado sempre que o roteador acessa uma entrada na tabela de mapeamento. Se a entrada não for acessada novamente antes de um certo período de tempo, a entrada é removida da tabela.

O esquema descrito anteriormente trata satisfatoriamente dos modos mais comuns de comunicação para computadores que possuem endereços IP não registrados que atuam como clientes para serviços externos, como os servidores Web. No entanto, isso não os permite atuar como servidores atendendo a solicitações provenientes de clientes externos. Para tratar desse caso, os roteadores NAT podem ser manualmente configurados para encaminhar todos os pedidos recebidos em determinada porta para um computador interno específico. Os computadores que atuam como servidores devem manter o mesmo endereço IP interno e isso é obtido por meio da configuração manual de seus endereços (como foi feito para PC 1). Essa solução para o problema do fornecimento de acesso externo a serviços é satisfatória, desde que não haja nenhum requisito para que mais de um computador interno ofereça um serviço em qualquer porta.

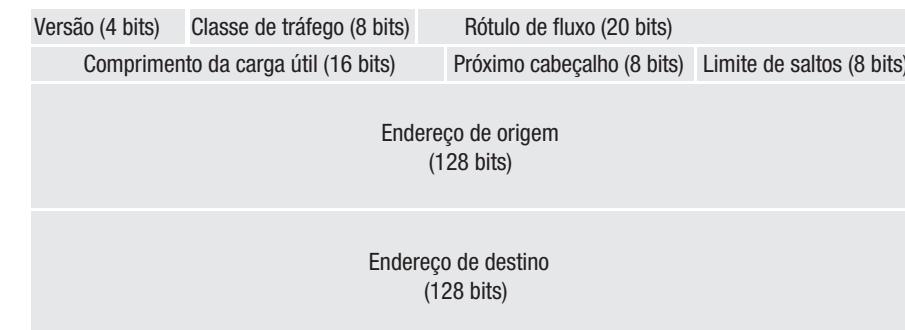


Figura 3.19 Formato do cabeçalho do IPv6.

O NAT foi introduzido como uma solução a curto prazo para o problema da alocação de endereços IP para computadores pessoais e domésticos. Ele tem permitido que a expansão do uso da Internet ocorra de forma muito maior do que foi originalmente previsto, mas também impõe algumas limitações, como a exemplificada anteriormente. O IPv6 deve ser visto como o próximo passo, permitindo participação total na Internet de todos os computadores e dispositivos portáteis.

3.4.4 IPv6

Uma solução definitiva para as limitações do endereçamento do IPv4 foi investigada, o que levou ao desenvolvimento e à adoção de uma nova versão do protocolo IP com endereços substancialmente maiores. Já em 1990, o IETF percebeu os problemas em potencial provenientes dos endereços de 32 bits do IPv4 e iniciou um projeto para desenvolver uma nova versão do protocolo IP. O IPv6 foi adotado pelo IETF em 1994 e foi também recomendada uma estratégia de migração.

A Figura 3.19 mostra o formato do cabeçalho IPv6. Não nos propomos a abordar sua construção em detalhes aqui. Indicamos aos leitores os textos de Tanenbaum [2003], Stallings [2002] e Huitema [1998] para estudar sobre o processo de projeto e os planos de implementação do IPv6. Aqui, destacaremos os principais avanços dados pelo IPv6.

Espaço de endereçamento: os endereços do IPv6 têm 128 bits (16 bytes). Isso proporciona um número astronômico de entidades endereçáveis: 2^{128} ou, aproximadamente, $3 \cdot 10^{38}$. Tanenbaum calcula que isso é suficiente para fornecer 7×10^{23} endereços IP por metro quadrado da superfície inteira da Terra. De forma mais conservadora, Huitema fez um cálculo pressupondo que os endereços IP são alocados de modo tão ineficiente como os números de telefone e obteve o valor de 1.000 endereços IP por metro quadrado da superfície da Terra (terra e água).

O espaço de endereçamento do IPv6 é particionado. Não detalharemos aqui esse particionamento, mas mesmo as menores partições (uma das quais conterá o intervalo de endereços IPv4 inteiro, com mapeamento de um para um) são maiores do que o espaço total do IPv4. Muitas partições (representando 72% do total) são reservadas para propósitos ainda indefinidos. Duas partições grandes (cada uma compreendendo 1/8 do espaço de endereços) são alocadas para propósitos gerais e serão atribuídas aos nós de rede. Uma delas é baseada na localização geográfica dos nós e a outra, de acordo com uma localização organizacional. Isso possibilita duas

estratégias alternativas para agregar endereços com vistas ao roteamento – o futuro dirá qual se mostrará mais eficiente ou popular.

Desempenho no roteamento: o processamento efetuado em cada nó para executar o roteamento IPv6 é reduzido em relação ao IPv4. Nenhuma soma de verificação é aplicada ao conteúdo do pacote (carga útil) e nenhuma fragmentação pode ocorrer, uma vez que um pacote tenha iniciado sua jornada. O primeiro é considerado aceitável, pois os erros podem ser detectados em níveis mais altos (o protocolo TCP inclui uma soma de verificação de conteúdo), e o último é obtido por meio do suporte de um mecanismo para determinação do menor MTU, antes que um pacote seja transmitido.

Suporte a tempo real e outros serviços especiais: os campos *classe de tráfego* e *rótulo de fluxo* estão relacionados a isso. Fluxos multimídia e outras sequências de elementos de dados em tempo real podem ser transmitidos em um fluxo identificado. Os primeiros seis bits do campo *classe de tráfego* podem ser usados em conjunto, ou independentemente, com o *rótulo de fluxo* para permitir que pacotes específicos sejam manipulados mais rapidamente ou com confiabilidade maior do que outros. Os valores de classe de tráfego de 0 a 8 servem para transmissões mais lentas sem causar efeitos não desejados para as aplicações. Os outros valores são reservados para pacotes cuja distribuição é dependente do tempo. Tais pacotes devem ser prontamente encaminhados ou eliminados – sua posterior distribuição não tem significado.

Os rótulos de fluxo permitem que recursos sejam reservados para atender a requisitos de sincronização de fluxos de dados em tempo real específicos, como transmissões de áudio e vídeo ao vivo. O Capítulo 20 discutirá esses requisitos e os métodos para a alocação de recursos a eles. É claro que os roteadores e os enlaces de transmissão na Internet possuem limitações físicas e o conceito de reservá-los para usuários e aplicações específicas não foi considerado anteriormente. O uso dessas facilidades do IPv6 dependerá de maiores aprimoramentos na infraestrutura e no desenvolvimento de métodos convenientes para ordenar e arbitrar a alocação de recursos.

Evolução futura: o segredo para a evolução futura é o campo *próximo cabeçalho*. Se for diferente de zero, ele define o tipo de um cabeçalho de extensão que é incluído no pacote. Atualmente, existem seis tipos de cabeçalho de extensão que fornecem dados adicionais para serviços especiais: informações para roteadores, definição de rota, tratamento de fragmentos, autenticação, criptografia e informações de tratamento no destino. Cada tipo de cabeçalho de extensão tem um tamanho específico e um formato definido. Novos tipos de cabeçalho de extensão poderão ser definidos quando surgirem novos requisitos de serviços. Um cabeçalho de extensão, se presente, vem após o cabeçalho obrigatório (Figura 3.19), precedendo a área de dados, e inclui um campo *próximo cabeçalho*, permitindo que vários cabeçalhos de extensão sejam encadeados.

Difusão seletiva (multicast) e não-seletiva (anycast): tanto o IPv4 como o IPv6 incluem suporte para a transmissão de pacotes IP para vários *hosts* usando um único endereço (que está no intervalo reservado para isso). Os roteadores IP são, então, responsáveis pelo roteamento do pacote para todos os *hosts* que se inscreveram no grupo identificado pelo endereço *multicast*. Mais detalhes sobre comunicação *multicast* IP podem ser encontrados na Seção 4.4.1. Além disso, o IPv6 suporta um novo modo de transmissão, chamado *difusão não-seletiva (anycast)*. Esse serviço distribui um pacote para pelo menos um dos *hosts* que está inscrito no endereço de grupo *anycast*.

Segurança: até agora, os aplicativos Internet que exigem transmissão de dados autenticada, ou com privacidade, contam com o uso de técnicas de criptografia na camada de aplicação. O princípio fim-a-fim apoia a visão de que esse é o lugar certo para isso. Se a segurança fosse implementada em nível IP, os usuários e desenvolvedores de aplicações dependeriam do código implementado em cada roteador percorrido e deveriam confiar neles, e em outros nós intermediários, para a manipulação das chaves criptográficas.

A vantagem de implementar a segurança em nível IP é que ela pode ser aplicada sem a necessidade das aplicações serem implementadas com requisitos de segurança. Por exemplo, os administradores de rede podem configurá-la em um *firewall* e aplicá-la uniformemente a toda comunicação externa, sem incorrer no custo da criptografia para as comunicações realizadas na rede interna. Os roteadores também podem explorar um mecanismo de segurança em nível IP para dar garantias sobre as mensagens de atualização de tabelas de roteamento que trocam entre si.

A segurança no IPv6 é implementada por meio dos tipos de cabeçalho de extensão de *autenticação* e de *criptografia*. Eles implementam recursos equivalentes ao conceito de canal seguro apresentado na Seção 2.4.3. A carga útil (área de dados) é cifrada e/ou assinada digitalmente, conforme for exigido. Recursos de segurança semelhantes também estão disponíveis no IPv4, usando túneis IP entre roteadores e *hosts* que implementam a especificação IPSec (veja a RFC 2411 [Thayer 1998]).

Migração do IPv4 • As consequências para a infraestrutura existente na Internet de uma alteração em seu protocolo básico são profundas. O IP é processado na pilha de protocolos TCP/IP em cada *host* e no *software* de cada roteador. Os endereços IP são manipulados em muitos programas aplicativos e utilitários. Isso exige que todos sofram atualizações para suportar a nova versão de IP. No entanto, a alteração é inevitável em razão do futuro esgotamento do espaço de endereçamento fornecido pelo IPv4, e o grupo de trabalho da IETF, responsável pelo IPv6, definiu uma estratégia de migração – basicamente, ela envolve a implementação de “ilhas” de roteadores e *hosts* IPv6 se comunicando por meio de túneis com outras ilhas IPv6 e sua gradual agregação em ilhas maiores.

Conforme mencionamos, os roteadores e *hosts* IPv6 não devem ter nenhuma dificuldade no tratamento de tráfego misto, pois o espaço de endereçamento do IPv4 é incorporado ao espaço do IPv6. Todos os principais sistemas operacionais (Windows XP, Mac OS X, GNU/Linux e outras variantes do Unix) já incluem implementações de soquetes UDP e TCP (conforme descrito no Capítulo 4) sobre IPv6, permitindo que os aplicativos sejam migrados com uma atualização simples.

A teoria dessa estratégia é tecnicamente sólida, mas o andamento da implementação tem sido muito lento, talvez porque o CIDR e o NAT têm aliviado a pressão mais do que havia sido previsto. Esse quadro começou a mudar com os mercados de telefonia móvel e de equipamentos portáteis. Todos esses dispositivos provavelmente serão habilitados para a Internet em um futuro próximo, e eles não podem ser facilmente ocultados atrás de roteadores com suporte a NAT. Por exemplo, estima-se que mais de um bilhão de dispositivos IP sejam distribuídos na Índia e na China em 2014. Apenas o IPv6 pode tratar de necessidades como essa.

3.4.5 MobileIP

Os computadores móveis, como os *notebooks* e os *palmtops*, são conectados à Internet a partir de diferentes locais, à medida que migram. O dono de um *notebook*, enquanto estiver em seu escritório, pode acessar a Internet através de uma conexão a uma rede local

Ethernet, assim como também pode fazê-lo no trânsito, dentro de seu carro, a partir de um modem para telefone celular. É possível ainda que esse *notebook* seja conectado em uma rede local Ethernet em um outro local qualquer. Esse usuário pode desejar ter acesso a serviços como *e-mail* e Web em cada um desses locais.

O simples acesso aos serviços não exige que um computador móvel mantenha um único endereço, e ele pode adquirir um novo endereço IP em cada local diferente; esse é o objetivo do DHCP, que permite a um computador recentemente conectado adquirir dinamicamente um endereço IP do intervalo de endereços da sub-rede local e descobrir os endereços de recursos locais, como um servidor de DNS. Ele também precisará descobrir quais serviços locais (como impressão, *e-mail*, etc.) estão disponíveis em cada instalação que visitar. Os serviços de descoberta são um tipo de serviço de atribuição de nomes que auxiliam nessa tarefa; eles serão descritos no Capítulo 19 (Seção 19.2).

No *notebook* podem existir arquivos, ou outros tipos de recursos, necessários para outros usuários, ou pode estar sendo executado um aplicativo distribuído, como um serviço que recebe notificações sobre o comportamento de ações da bolsa que atinjam um limite predefinido. Em situações como essa, um computador móvel precisa se manter acessível para clientes e aplicativos, mesmo se locomovendo entre redes locais e redes sem fio. Para isso, ele deve manter um único número IP, porém o roteamento IP é baseado na sub-rede. As sub-redes estão em locais fixos e o roteamento é dependente da localização na rede.

O MobileIP é uma solução para este último problema. A solução é implementada de forma transparente, de modo que a comunicação de IP continue normalmente, mesmo quando um *host* móvel se movimenta entre sub-redes de diferentes locais. Ela é baseada na alocação permanente de um endereço IP “convencional” para cada *host* móvel, em uma sub-rede de seu domínio doméstico.

Quando o *host* móvel está conectado em sua rede de domicílio, os pacotes são direcionados para ele da maneira habitual. Quando ele está conectado na Internet, em qualquer lugar, dois processos assumem a responsabilidade pelo redirecionamento. São eles: um *agente doméstico* (AD) e um *agente estrangeiro* (AE). Esses processos são executados em computadores fixos na rede doméstica e no local corrente do *host* móvel.

O AD é responsável por manter o conhecimento atualizado da localização corrente do *host* (o endereço IP por meio do qual ele pode ser encontrado). Isso é feito com a ajuda do próprio *host* móvel que, quando deixa sua rede doméstica, informa ao AD, que anota a ausência. Durante a ausência, o AD se comporta como um *proxy*; para isso, ele diz aos roteadores locais que cancelam todos os registros colocados em cache relacionados ao endereço IP do *host* móvel. Enquanto estiver atuando como *proxy*, o AD responde às requisições ARP relativas ao endereço IP do *host* móvel, fornecendo seu próprio endereço físico de rede como endereço físico de rede do *host* móvel.

Quando o *host* móvel se instala em um novo local, ele informa ao AE sobre sua chegada. O AE aloca para ele um “endereço aos cuidados de” (COA, care-of-address) – que é um novo endereço IP, temporário, na sub-rede local. Então, o AE entra em contato com o AD e fornece o endereço IP local atribuído ao *host* móvel e o *endereço aos cuidados de* que foi alocado a ele.

A Figura 3.20 ilustra o mecanismo de roteamento do MobileIP. Quando um datagrama IP endereçado ao endereço doméstico do *host* móvel é recebido na rede doméstica, ele é direcionado para o AD. Então, o AD encapsula o datagrama IP em um pacote MobileIP e o envia para o AE. O AE desempacota o datagrama IP original e o distribui para o *host* móvel por meio da rede local na qual está atualmente ligado. Note que o método pelo qual o AD e o AE redirecionam o datagrama original até seu destino é um exemplo da técnica de túneis descrita na Seção 3.3.7.

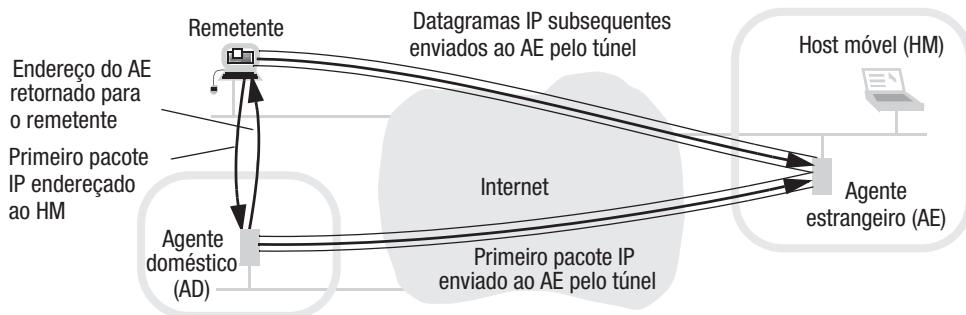


Figura 3.20 O mecanismo de roteamento MobileIP.

O AD também envia o *endereço aos cuidados de* do host móvel para o remetente original. Se o remetente for compatível com o MobileIP, ele notará o novo endereço e o utilizará nas comunicação subsequentes com esse host móvel, evitando as sobrecargas do redirecionamento por meio do AD. Caso não seja compatível, ele ignorará a mudança de endereço e a comunicação continuará a ser redirecionada por meio do AD.

A solução MobileIP é eficaz, mas dificilmente eficiente. Uma solução que tratasse os hosts móveis como cidadão de primeira classe seria preferível, permitindo-os perambular sem aviso e direcionar datagramas a eles sem nenhuma utilização de túnel ou redirecionamento. Devemos notar que esse feito aparentemente difícil é exatamente o que ocorre na telefonia celular – os celulares não mudam de número ao se moverem entre as células ou mesmo entre os países. Em vez disso, eles simplesmente notificam, de tempos em tempos, a estação de base de telefonia celular local sobre sua presença.

3.4.6 TCP e UDP

Os protocolos TCP e UDP disponibilizam os recursos de comunicação via Internet de uma forma bastante prática para programas aplicativos. No entanto, os desenvolvedores de aplicativos talvez queiram outros tipos de serviço de transporte, por exemplo, para fornecer garantias ou segurança em tempo real, mas tais serviços geralmente exigem mais suporte na camada de rede do que o fornecido pelo IPv4. Os protocolos TCP e UDP podem ser considerados um reflexo exato, em nível da programação de aplicativos, dos recursos de comunicação que o IPv4 tem a oferecer. O IPv6 é outra história; ele certamente continuará a suportar TCP e UDP, mas incluirá recursos que não podem ser convenientemente usados por meio de TCP e UDP. As capacidades do IPv6 serão úteis para se introduzir tipos adicionais de serviço de transporte quando a sua utilização for suficientemente ampla para justificar tal desenvolvimento.

O Capítulo 4 descreverá as características do TCP e do UDP, do ponto de vista dos programadores de aplicativos distribuídos. Aqui, precisamos ser muito sucintos, descrevendo apenas a funcionalidade que eles acrescentam ao IP.

Uso de portas • A primeira característica a notar é que, enquanto o IP suporta comunicação entre pares de computadores (identificados pelos seus endereços IP), o TCP e o UDP, como protocolos de transporte, fornecem comunicação de processo para processo. Isso é feito pelo uso de portas. Os *números de porta* são utilizados para endereçar mensagens para processos em um computador em particular e são válidos somente nesse computador. Um número de porta é um valor inteiro de 16 bits. Uma vez que um datagrama IP

tenha sido entregue ao *host* destino, o *software* da camada TCP (ou UDP) o envia para um processo específico associado a uma dada porta nesse *host*.

Características do UDP • O UDP é quase uma réplica, em nível de transporte, do IP. Um datagrama UDP é encapsulado dentro de um datagrama IP. Ele tem um cabeçalho curto que inclui os números de porta de origem e de destino (os endereços de *host* correspondentes estão presentes no cabeçalho IP), um campo de comprimento e uma soma de verificação. O UDP não oferece nenhuma garantia de entrega. Já dissemos que datagramas IP podem ser perdidos por causa de congestionamento ou erro na rede. O UDP não acrescenta nenhum mecanismo de confiabilidade adicional, exceto a soma de verificação, que é opcional. Se o campo de soma de verificação for diferente de zero, o *host* destino calculará um valor de verificação a partir do conteúdo do datagrama e o comparará com a soma de verificação recebida; se eles não corresponderem, o datagrama UDP é eliminado.

Assim, o UDP fornece uma maneira de transmitir mensagens de até 64 Kbytes de tamanho (o pacote máximo permitido pelo IP) entre pares de processos (ou de um processo para vários, no caso de datagramas enviados para endereços IP *multicast*) com custos adicionais ou atrasos mínimos se comparados àqueles esperados para transmissão IP. Ele não acarreta nenhum custo de configuração e não exige nenhuma mensagem de confirmação. Seu uso, no entanto, está restrito a aplicativos e serviços que não exigem entrega confiável de uma ou várias mensagens.

Características do TCP • O TCP fornece um serviço de transporte muito mais sofisticado. Ele oferece entrega confiável de sequências de bytes arbitrariamente longas por meio de uma abstração denominada fluxo de dados (*data stream*). A garantia da confiabilidade implica a recepção, por parte do processo destino, de todos os dados enviados, na mesma ordem. O TCP é orientado à conexão. Antes que qualquer dado seja transferido, o processo remetente e o processo destino devem estabelecer um canal de comunicação bidirecional: a conexão. A conexão é simplesmente um acordo, entre os processos participantes, que viabiliza uma transmissão de dados confiável. Os nós intermediários, como os roteadores, não têm conhecimento algum das conexões TCP, e nem todos os datagramas IP que transferem os dados em uma transmissão TCP seguem necessariamente a mesma rota.

A camada TCP inclui mecanismos adicionais (implementados sobre IP) para satisfazer as garantias de confiabilidade. São eles:

Sequenciamento: um processo remetente divide o fluxo TCP em uma sequência de segmentos de dados e os transmite como datagramas IP. Um número de sequência é anexado a cada segmento TCP, fornecendo a ordem em que o primeiro byte desse segmento aparece no fluxo TCP. O destino usa os números de sequência para ordenar os segmentos recebidos, antes de passá-los para o fluxo de entrada de dados do processo destino. Nenhum segmento pode ser colocado no fluxo de entrada até que todos os segmentos de número menor tenham sido recebidos e inseridos no fluxo; portanto, os segmentos que chegam fora de ordem devem ser mantidos em um *buffer*, até que seus predecessores cheguem.

Controle de fluxo: o remetente toma o cuidado de não sobrecarregar o destino ou os nós intermediários. Isso é obtido por meio de um sistema de confirmação de segmentos. Quando um destino recebe um segmento com êxito, ele grava seu número de sequência. De tempos em tempos, o destino envia uma confirmação para o remetente, fornecendo o número de sequência mais alto em seu fluxo de entrada, junto a um *tamanho de janela*. Se houver um fluxo de dados no sentido inverso, as confirmações são transportadas nos próprios segmentos de dados; caso contrário,

elas são enviadas em segmentos específicos de confirmação. O campo tamanho de janela no segmento de confirmação especifica o volume de dados que o remetente pode enviar antes de receber a próxima confirmação.

Quando uma conexão TCP é usada para comunicação com um programa interativo remoto, os dados podem ser produzidos em pequenos volumes, mas na forma de picos. Por exemplo, a entrada do teclado pode resultar em apenas alguns caracteres por segundo, mas os caracteres devem ser enviados de maneira suficientemente rápida para que o usuário veja os resultados de sua digitação. Isso é tratado pela configuração de um tempo de espera T no *buffer* local – normalmente 0,5 segundos. Com esse esquema simples, um segmento é enviado para o receptor quando os dados estão esperando no *buffer* de saída há T segundos ou quando o conteúdo do *buffer* atinge o limite da MTU. Esse esquema de bufferização não pode acrescentar mais do que T segundos ao atraso interativo. Nagle [1984] descreveu outro algoritmo, usado em muitas implementações de TCP, que produz menos tráfego e é mais eficiente para alguns aplicativos interativos. A maioria das implementações de TCP pode ser configurada, permitindo que os aplicativos alterem o valor de T ou selecionem um de vários algoritmos de bufferização.

Por causa da falta de confiabilidade das redes sem fio, e da resultante perda frequente de pacotes, esses mecanismos de controle de fluxo não são apropriados para a comunicação sem fio. Esse é um dos motivos para a adoção de um mecanismo de transporte diferente na família de protocolos WAP, para comunicação móvel de longa distância. Contudo a implementação de TCP para redes sem fio também é importante, e foram propostas modificações no mecanismo TCP com esse objetivo [Balakrishnan *et al.* 1995, 1996]. A ideia é implementar um módulo de suporte a TCP no ponto de acesso sem fio (o *gateway* entre redes com e sem fio). O módulo de suporte monitora os segmentos TCP na rede sem fio, retransmitindo os segmentos que não são confirmados rapidamente pelo receptor móvel e solicitando retransmissões dos segmentos recebidos quando são notadas lacunas nos números de sequência.

Retransmissão: o remetente registra os números de sequência dos segmentos que envia. Quando recebe uma confirmação, ele nota que os segmentos foram recebidos com sucesso e pode, então, excluí-los de seus *buffers* de saída. Se qualquer segmento não for confirmado dentro de um tempo limite especificado, o remetente o retransmitirá.

Uso de buffers: o *buffer* de entrada do receptor é usado para balancear o fluxo entre o remetente e o destino. Se o processo destino executar operações de recepção mais lentamente do que o remetente faz operações de envio, o volume de dados no *buffer* crescerá. Normalmente, os dados são extraídos do *buffer*, antes que ele se torne cheio, mas o *buffer* pode esgotar e, quando isso acontece, os segmentos recebidos são simplesmente eliminados, sem registro de sua chegada. Portanto, sua chegada não é confirmada e o remetente é obrigado a retransmiti-los.

Soma de verificação: cada segmento mantém uma soma de verificação abrangendo o cabeçalho e os dados do segmento. Se um segmento recebido não corresponde à sua soma de verificação, ele é eliminado.

3.4.7 Sistema de nomes de domínio

O projeto e a implementação do DNS (Domain Name System) serão descritos em detalhes no Capítulo 13; fornecemos, aqui, um breve panorama para completar nossa discussão sobre os protocolos Internet. A Internet suporta um esquema para o uso de nomes

simbólicos para *hosts* e redes, como *binkley.cs.mcgill.ca* ou *essex.ac.uk*. Os nomes são organizados de acordo com uma hierarquia de atribuição na forma de uma árvore. As entidades nomeadas são chamadas de *domínios* e os nomes simbólicos são chamados de *nomes de domínio*. Os domínios são organizados, hierarquicamente, de forma a refletir sua estrutura organizacional. A hierarquia de atribuição de nomes é totalmente independente da topologia física das redes que constituem a Internet. Os nomes de domínio são convenientes para os seres humanos, mas precisam ser transformados em endereços IP antes de serem usados como identificadores de uma comunicação. Isso é responsabilidade de um serviço específico, o DNS. Os aplicativos enviam requisições para o serviço de DNS, para converter os nomes de domínio fornecidos pelos usuários em endereços IP.

O DNS é implementado como um processo servidor que pode ser executado nos computadores *host* de qualquer parte da Internet. Existem pelo menos dois servidores de DNS em cada domínio e, frequentemente, há mais. Os servidores de cada domínio contêm um mapa parcial da hierarquia de nomes que está abaixo de seu domínio. Eles devem conter pelo menos a parte que consiste em todos os nomes de domínio e *host* dentro de seus domínios, mas frequentemente contêm uma parte maior. Os servidores de DNS tratam as requisições de resolução de nomes de domínio que estão fora de sua hierarquia, emitindo requisições para os servidores de DNS nos domínios relevantes, prosseguindo recursivamente, da direita para a esquerda, transformando o nome em segmentos. A transformação resultante é, então, colocada na cache do servidor que está tratando a requisição original, para que futuras requisições de resolução de nomes que se refiram ao mesmo domínio sejam solucionadas sem referência a outros servidores. O DNS não funcionaria sem o uso extensivo da cache, pois os servidores de nomes-raiz seriam consultados em praticamente todos os casos, criando um gargalo no acesso ao serviço.

3.4.8 Firewalls

Quase todas as organizações precisam de conectividade à Internet para fornecer serviços para seus clientes, para usuários externos e para permitir que seus usuários internos acessem informações e serviços. Os computadores da maioria das organizações são bastante diversificados, executando uma variedade de sistemas operacionais e de *softwares* aplicativos. Para piorar a situação, alguns aplicativos podem incluir tecnologias de segurança de ponta, porém, a maior parte deles tem pouca ou nenhuma capacidade para garantir a integridade dos dados recebidos ou, quando necessário, a privacidade no envio dos dados. Em resumo, em uma intranet com muitos computadores e uma ampla variedade de *software*, é inevitável que algumas partes do sistema tenham vulnerabilidades que o exponham a ataques contra sua segurança. As formas de ataque serão melhor detalhadas no Capítulo 11.

O objetivo de um *firewall* é monitorar e controlar toda a comunicação para dentro e para fora de uma intranet. Um *firewall* é implementado por um conjunto de processos que atuam como um *gateway* para uma intranet (Figura 3.21a), aplicando uma política de segurança determinada pela organização.

O objetivo de uma política de segurança com *firewall* pode incluir parte de ou tudo que segue:

Controle de serviço: para determinar quais serviços nos *hosts* internos estão disponíveis para acesso externo e para rejeitar quaisquer pedidos a outros serviços. Os pedidos de serviço enviados e as respostas também podem ser controlados. Essas ações de filtragem podem ser baseadas no conteúdo de datagramas IP e dos cabeçalhos TCP e UDP que eles contêm. Por exemplo, os pedidos de HTTP recebidos podem ser rejeitados, a não ser que sejam direcionados para um servidor Web considerado “oficial”.

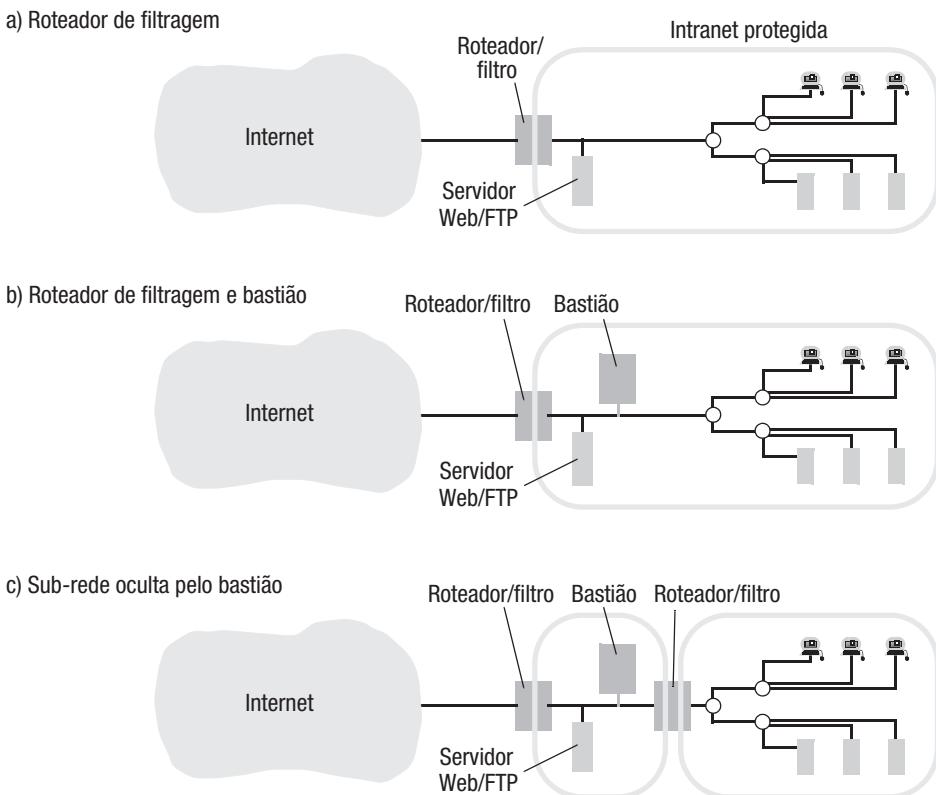


Figura 3.21 Configurações de *firewall*.

Controle de comportamento: para evitar comportamentos que violem as políticas da organização, seja antissocial ou que não tenha nenhum propósito legítimo discernível e, portanto, seja suspeito de fazer parte de um ataque. Algumas dessas ações de filtragem podem ser aplicadas às camadas de rede (IP) ou de transporte (TCP/UDP), mas outras podem exigir interpretação das mensagens da camada de aplicação. Por exemplo, a filtragem de ataques por *spam* pode exigir um exame do endereço de *e-mail* do remetente nos cabeçalhos da mensagem ou mesmo no seu conteúdo.

Controle de usuário: talvez a organização queira fazer discriminação entre seus usuários, permitindo que alguns acessem serviços externos, mas proibindo que outros o façam. Um exemplo de controle de usuário, que talvez seja mais socialmente aceitável, é impedir, exceto para os usuários que sejam membros da equipe de administração da rede, a instalação de *software* para evitar infecção por vírus ou para manter padrões de *software*. Na verdade, esse exemplo em particular seria difícil de implementar sem limitar o uso da Web por usuários normais.

Outro exemplo de controle de usuário é o gerenciamento de conexões *dial-up* e outras fornecidas externamente para os usuários. Se o *firewall* também é o *host* para conexões via modem, ele pode autenticar o usuário no momento da conexão e exigir o uso de um canal seguro para toda comunicação (para evitar intromissão,

mascaramento e outros ataques na conexão externa). Esse é o objetivo da tecnologia de VPN (Virtual Private Network), descrita na próxima sub-seção.

A política precisa ser expressa em termos das operações a serem executadas pelos processos de filtragem nos vários níveis diferentes:

Filtragem de datagramas IP: esse é um processo de filtragem que examina individualmente os datagramas IP. Ele pode tomar decisões com base nos endereços de destino e de origem. O processo também pode examinar o campo *tipo de serviço* dos datagramas IP e interpretar o conteúdo dos pacotes com base nele. Por exemplo, o processo pode filtrar pacotes TCP de acordo com o número da porta para a qual eles foram endereçados, e como os serviços estão geralmente localizados em portas conhecidas, isso permite que os pacotes sejam filtrados com base no serviço solicitado. Por exemplo, muitos *sites* proíbem o uso de servidores de NFS por clientes externos.

Por razões de desempenho, a filtragem de datagramas IP é normalmente realizada por um processo dentro do núcleo do sistema operacional de um roteador. Se forem usados vários *firewalls*, o primeiro pode marcar certos pacotes para um exame mais exaustivo por parte de um *firewall* posterior, permitindo que pacotes “limpos” prossigam. É possível filtrar com base nas sequências de datagramas IP, por exemplo, para impedir o acesso a um servidor FTP antes que um *login* tenha sido efetuado.

Gateway TCP: um processo *gateway TCP* verifica todos os pedidos de conexão TCP e as transmissões de segmento. Quando um *gateway TCP* é instalado, a configuração das conexões TCP pode ser controlada e a correção dos segmentos TCP pode ser verificada (alguns ataques de negação de serviço utilizam segmentos TCP malformados para entrar em sistemas operacionais de máquinas clientes). Quando desejado, eles podem ser direcionados por meio de um *gateway* em nível de aplicativo para verificação de conteúdo.

Gateway em nível de aplicativo: um processo *gateway* em nível de aplicativo atua como *proxy* para um processo aplicativo. Por exemplo, pode-se desejar uma política que permita conexões Telnet com certos *hosts* externos para determinados usuários internos. Quando um usuário executa um programa Telnet em seu computador local, ele tenta estabelecer uma conexão TCP com um *host* remoto. O pedido é interceptado pelo *gateway TCP*. O *gateway TCP* inicia um processo *proxy Telnet* e a conexão TCP original é direcionada para ele. Se o *proxy* aprovar a operação Telnet (o usuário está autorizado a usar o *host* solicitado), ele estabelece outra conexão com o *host* solicitado e, então, retransmite todos os segmentos TCP nas duas direções. Um processo *proxy* pode ser empregado para outros serviços, como por exemplo, para FTP.

Um *firewall* normalmente é composto por vários processos trabalhando em diferentes camadas de protocolo. É comum empregar mais de um computador em tarefas de *firewall*, por motivos de desempenho e tolerância a falhas. Em todas as configurações descritas a seguir, e ilustradas na Figura 3.21, mostramos um servidor Web público e um servidor FTP sem proteção. Eles contêm apenas informações que não exigem proteção contra acesso público, e seu *software* servidor garante que apenas usuários internos autorizados possam atualizá-las.

A filtragem de datagramas IP normalmente é feita por um roteador – um computador com pelo menos dois endereços de rede, em redes IP distintas, que executa um processo RIP, um processo de filtragem de datagramas IP e um mínimo possível de outros processos. O roteador/filtro só deve executar *software* confiável, de forma que se garanta a execução de suas políticas de filtragem. Isso envolve certificar-se de que nenhum processo do tipo cavalo de Troia possa ser executado nele, e que os *softwares* de filtragem e de roteamento

não tenham sido modificados, nem falsificados. A Figura 3.21a mostra uma configuração de *firewall* simples que conta apenas com a filtragem de datagramas IP e emprega um único roteador para esse fim. A configuração de rede da Figura 3.10, por motivos de desempenho e confiabilidade, possui dois roteadores/filtros atuando como *firewall*. Ambos obedecem a uma mesma política de filtragem e o segundo não aumenta a segurança do sistema.

Quando são exigidos processos TCP e *gateway* em nível de aplicativo, normalmente eles são executados em um computador separado, conhecido como *bastião*. (O termo se origina da construção de castelos fortificados; trata-se de uma torre de vigia saliente, a partir da qual o castelo pode ser defendido ou os defensores podem negociar com os que desejam entrar.) Um computador bastião é um *host* localizado dentro da intranet, protegido por um roteador/filtro IP, que executa o protocolo TCP e *gateways* em nível de aplicativo (Figura 3.21b). Assim como o roteador/filtro, o bastião deve executar *software* confiável. Em uma intranet bem segura, todos os acessos a serviços externos devem ser feitos via *proxy*. Os leitores podem estar familiarizados com o uso de *proxies* para acesso à Web. Eles são um exemplo do uso de *proxy* como *firewall*; frequentemente, são construídos de maneira a integrar um servidor de cache Web (descrito no Capítulo 2). Esse e outros processos *proxy* provavelmente exigirão processamento e recursos de armazenamento substanciais.

A segurança pode ser melhorada pelo emprego de dois roteadores/filtros em série, com o bastião e os servidores públicos localizados em uma sub-rede separada ligando os roteadores/filtros (Figura 3.21c). Essa configuração tem diversas vantagens para a segurança:

- Se a política aplicada pelo bastião for restrita, os endereços IP dos *hosts* da intranet não precisarão nem mesmo ser divulgados para o mundo exterior, e os endereços do mundo exterior não precisarão ser conhecidos dos computadores internos, pois toda comunicação externa passará pelos processos *proxies* que estão no bastião.
- Se o primeiro roteador/filtro for invadido ou comprometido, o segundo, que é invisível fora da intranet e, portanto, menos vulnerável, continuará a selecionar e a rejeitar datagramas IP indesejáveis.

Redes privadas virtuais • As redes privadas virtuais (VPNs, Virtual Private Networks) ampliam o limite de proteção para além de uma intranet local por meio do uso de canais seguros protegidos por criptografia em nível IP. Na Seção 3.4.4, delineamos as extensões de segurança IP disponíveis no IPv6 e no IPv4 com túneis IPSec [Thayer 1998]. Elas são a base para a criação de VPNs. As VPNs podem ser usadas por usuários externos individuais tanto para proteger suas conexões como para implementar um ambiente seguro entre intranets localizadas em diferentes *sites*, usando enlaces Internet públicos.

Por exemplo, talvez um membro de uma equipe precise se conectar à intranet de sua empresa por meio de um provedor de serviços de Internet. Uma vez conectado, ele deve ter acesso aos mesmos recursos que um usuário que está dentro da rede protegida por *firewall*. Isso pode ser obtido se seu *host* local implementar segurança IP. Neste caso, o *host* local compartilha uma ou mais chaves criptográficas com o *firewall* e elas são usadas para estabelecer um canal seguro no momento da conexão. Os mecanismos de canal seguro serão descritos em detalhes no Capítulo 11.

3.5 Estudos de caso: Ethernet, WiFi e Bluetooth

Até este ponto, discutimos os princípios envolvidos na construção de redes de computadores e descrevemos o IP, a “camada de rede virtual” da Internet. Para concluirmos o capítulo, descreveremos, nesta seção, os princípios e as implementações de três redes físicas.

No início dos anos 80, o IEEE (Institute of Electrical and Electronic Engineers), dos Estados Unidos, instituiu um comitê para especificar uma série de padrões para redes locais (o 802 Committee [IEEE 1990]), e seus subcomitês produziram uma série de especificações que se tornaram os principais padrões para redes locais. Na maioria dos casos, os padrões são baseados em outros já existentes no setor, que surgiram a partir de pesquisas feitas nos anos 70. Os subcomitês relevantes e os padrões publicados são dados na Figura 3.22.

Os vários padrões diferem entre si no desempenho, na eficiência, na confiabilidade e no custo, mas todos fornecem recursos para a interligação em rede com uma largura de banda relativamente alta em distâncias curtas e médias. O padrão Ethernet IEEE 802.3 domina o mercado das redes locais cabeadas e será descrito na Seção 3.5.1 como nosso representante da tecnologia de rede local cabeada. Embora estejam disponíveis implementações de Ethernet para várias larguras de banda, os princípios de operação são idênticos em todas elas.

O padrão *Token Ring* IEEE 802.5 foi um concorrente importante em grande parte dos anos 90, oferecendo vantagens em relação ao padrão Ethernet na eficiência e em seu suporte para garantia de largura de banda, mas agora desapareceu do mercado. Os leitores que estiverem interessados nessa tecnologia de rede local podem encontrar uma breve descrição no endereço www.cdk5.net/networking (em inglês). A popularização dos *switches* Ethernet (em oposição aos hubs) tem permitido que as redes Ethernet sejam configuradas de maneira a oferecer garantias de largura de banda e latência (conforme discutido na Seção 3.5.1, sub-seção “Ethernet para aplicações de tempo real e com exigências de qualidade de serviço”), e esse é um dos motivos do desaparecimento da tecnologia *token ring*.

O padrão *Token Bus* IEEE 802.4 foi desenvolvido para aplicações industriais com requisitos de tempo real. O padrão *Metropolitan Area Network* IEEE 802.6 cobre distâncias de até 50 km e é destinado para uso em redes que abrangem municípios e cidades.

O padrão *Wireless LAN* IEEE 802.11 surgiu bem mais tarde, mas sob o nome comercial de WiFi, e mantém uma posição de destaque no mercado, com produtos de muitos fornecedores, podendo ser encontrado na maioria dos dispositivos de computação móveis e portáteis. O padrão IEEE 802.11 é projetado para suportar comunicação em velocidades de até 54 Mbps, em distâncias de no máximo 150 m entre dispositivos equipados com transmissores/receptores sem fio simples. Descreveremos seus princípios de operação na Seção 3.5.2. Mais detalhes sobre as redes IEEE 802.11 podem ser encontrados em Crow *et al.* [1997] e em Kurose e Ross [2007].

<i>IEEE No.</i>	<i>Nome</i>	<i>Título</i>	<i>Referência</i>
802.3	Ethernet	CSMA/CD Networks (Ethernet)	[IEEE 1985a]
802.4		Token Bus Networks	[IEEE 1985b]
802.5		Token Ring Networks	[IEEE 1985c]
802.6		Metropolitan Area Networks	[IEEE 1994]
802.11	WiFi	Wireless Local Area Networks	[IEEE 1999]
802.15.1	Bluetooth	Wireless Personal Area Networks	[IEEE 2002]
802.15.4	ZigBee	Wireless Sensor Networks	[IEEE 2003]
802.16	WiMAX	Wireless Metropolitan Area Networks	[IEEE 2004a]

Figura 3.22 Padrões IEEE 802.

O padrão *Wireless Personal Area Network* (Bluetooth) IEEE 802.15.1 foi baseado em uma tecnologia desenvolvida em 1999, pela Ericsson, para transportar voz e dados digitais, com largura de banda baixa, entre dispositivos como PDAs, telefones móveis e fones de ouvido, e subsequentemente, em 2002, foi padronizado como o padrão IEEE 802.15.1. A Seção 3.5.3 traz uma descrição do Bluetooth.

O IEEE 802.15.4 (ZigBee) é outro padrão de WPAN destinado a fornecer comunicação de dados para equipamentos domésticos de baixa energia e largura de banda muito baixa, como controles remotos, alarme contra roubo, sensores de sistema de aquecimento e dispositivos como crachás inteligentes e leitores de etiqueta. Tais redes são denominadas *redes de sensores sem fio* e suas aplicações e características de comunicação serão discutidas no Capítulo 19.

O padrão *Wireless MAN* IEEE 802.16 (nome comercial: WiMAX) foi ratificado em 2004 e em 2005. O padrão IEEE 802.16 foi projetado como uma alternativa para os enlaces a cabo e DSL para conexão de “última milha” com casas e escritórios. Uma variante do padrão (IEEE 802.20) se destina a superar as redes 802.11 WiFi como principal tecnologia de conexão para computadores *notebook* e dispositivos móveis em áreas públicas externas e internas.

A tecnologia ATM surgiu a partir de importantes trabalhos de pesquisa e padronização nos setores de telecomunicações e computação, no final dos anos 80, início dos 90 [CCITT 1990]. Seu objetivo é fornecer uma tecnologia para a interligação de redes de longa distância com uma alta largura de banda para aplicações nas áreas de telefonia, comunicação de dados e multimídia (áudio e vídeo de alta qualidade). Embora sua evolução tenha sido mais lenta do que o esperado, atualmente a tecnologia ATM é dominante na interligação de redes de longa distância de alto desempenho. O ATM também foi visto, em aplicações de rede local, como um substituto para redes Ethernet, mas acabou sendo relegado a um segundo plano devido à concorrência com as redes Ethernet de 100 Mbps e 1.000 Mbps, que estão disponíveis a um custo muito menor. Mais detalhes sobre ATM e outras tecnologias de rede de alta velocidade podem ser encontrados em Tanenbaum [2003] e em Stallings [2002].

3.5.1 Ethernet

A Ethernet foi desenvolvida no Xerox Palo Alto Research Center, em 1973 [Metcalfe e Boggs 1976; Shoch *et al.* 1982; 1985], como parte de um projeto de pesquisa sobre estações de trabalho pessoais e sistemas distribuídos. O protótipo da Ethernet foi a primeira rede local de alta velocidade (para a época), demonstrando sua exequibilidade e utilidade ao permitir que computadores de uma única instalação se comunicassem com baixas taxas de erro e sem atraso de comutação. A Ethernet original funcionava em 3 Mbps e, atualmente, estão disponíveis redes Ethernet com larguras de banda variando de 10 Mbps a 1.000 Mbps.

Vamos descrever os princípios de operação da Ethernet de 10 Mbps especificada no padrão IEEE 802.3 [IEEE 1985a]. Essa foi a primeira tecnologia de rede local amplamente usada. Atualmente, a variante de 100 Mbps é a mais empregada, mas o princípio de operação é idêntico ao original. Concluímos esta seção com uma lista das variantes mais importantes da tecnologia de transmissão Ethernet e da largura de banda que disponibilizam. Para ver descrições mais abrangentes da Ethernet em todas as suas variações, consulte Spurgeon [2000].

Uma rede Ethernet é formada por um barramento simples que usa como meio de transmissão um ou mais segmentos contínuos de cabos ligados por *hubs* ou repetidores. Os *hubs* e repetidores são dispositivos de interligação de cabos simples, que permitem que um mesmo sinal se propague por todos eles. Várias redes Ethernet podem ser interligadas

por meio de *switches* ou pontes Ethernet. Os *switches* e as pontes operam no nível dos quadros Ethernet, encaminhando-os, apropriadamente, para as redes Ethernet adjacentes de destino. Quando interligadas, as várias redes Ethernet aparecem como uma única rede para as camadas de protocolo mais altas, como a IP (veja a Figura 3.10, onde as sub-redes IP 138.37.88.0 e 138.37.94.0 são compostas, cada uma, de várias redes Ethernet ligadas por dispositivos rotulados como *Eswitch*). Em particular, uma requisição ARP (Seção 3.4.2), por ser feita em *broadcast* Ethernet, é confinada dentro dos limites de uma rede Ethernet.

O método de operação das redes Ethernet é definido pela frase “detecção de portadora com múltiplo acesso e detecção de colisão”, abreviado por CSMA/CD (Carrier Sensing, Multiple Access with Collision Detection), e elas pertencem à classe das redes denominadas *barramento de disputa* ou *contenção*. Esse tipo de barramento usa um único meio de transmissão para ligar todos os *hosts*. O protocolo que gerencia o acesso ao meio é chamado de MAC (*medium access control*). Como um único enlace conecta todos os *hosts*, o protocolo MAC combina as funções de protocolo da camada física (responsável pela transmissão dos sinais que representam os dados) e da camada de enlace (responsável pelo envio de quadros para os *hosts*) em uma única camada de protocolo.

Difusão de pacotes • O método de comunicação em redes CSMA/CD é por difusão (*broadcast*) de pacotes de dados no meio de transmissão. Todas as estações estão continuamente “escutando” o meio para ver se há quadros endereçados a elas. Qualquer estação que queira enviar uma mensagem transmite um ou mais quadros (chamados de *frames* na especificação Ethernet) no meio. Cada quadro contém o endereço físico da estação de destino, o endereço físico da estação de origem e uma sequência de bits de comprimento variável, representando a mensagem a ser transmitida. A transmissão de dados ocorre a 10 Mbps (ou em velocidades mais altas, especificadas para redes Ethernet de 100 e 1.000 Mbps) e o comprimento dos quadros varia entre 64 e 1.518 bytes; portanto, o tempo para transmitir um quadro em uma rede Ethernet de 10 Mbps é de 50 a 1.200 microsegundos, dependendo de seu comprimento. A MTU especificada no padrão IEEE é 1.518 bytes, embora não exista motivo técnico algum para definir qualquer limite superior fixo em particular, exceto a necessidade de limitar os atrasos causados pela disputa de acesso ao meio.

O endereço físico da estação de destino normalmente se refere a uma única interface de rede. O *hardware* da interface de rede de cada estação recebe todos os quadros e compara o endereço de destino com seu endereço local. Os quadros endereçados para outras estações são ignorados e aqueles que correspondem ao endereço físico do *host* local são repassados para as camadas superiores. O endereço de destino também pode especificar um endereço de *broadcast* ou de *multicast*. Os endereços “normais”, também denominados *unicast*, são diferenciados dos endereços de *broadcast* e de *multicast* pelo seu bit de ordem mais alta (0 e 1, respectivamente). O endereço físico composto por todos os seus bits em 1 é reservado para uso como endereço de *broadcast* e é empregado quando um quadro deve ser recebido por todas as estações da rede. Isso é utilizado, por exemplo, para implementar o protocolo ARP. Qualquer estação que receba um quadro com um endereço de *broadcast* o passará para as camadas superiores. Um endereço de *multicast* especifica uma forma limitada de *broadcast*, o qual é recebido apenas por um grupo de estações cujas interfaces de rede foram configuradas para receber quadros identificados com esse endereço. Nem todas as interfaces de rede Ethernet tratam endereços *multicast*.

O protocolo de rede Ethernet (que fornece a transmissão de quadros Ethernet entre pares de *hosts*) é implementado pelo *hardware* da própria interface Ethernet; já as camadas superiores são implementadas em *software*.

Formato do quadro Ethernet • Os quadros (*frames*) transmitidos pelas estações na rede Ethernet têm o seguinte formato:

	bytes: 7	1	6	6	2	46 < comprimento < 1.500	4
Preâmbulo	S	Endereço de destino	Endereço de origem	Comprimento dos dados	Dados para transmissão		Soma de verificação

Fora os endereços de destino e origem já mencionados, os quadros incluem um prefixo de 8 bytes, um campo de comprimento, uma área de dados e uma soma de verificação. O prefixo é usado para propósitos de sincronização do *hardware* e consiste em um preâmbulo de sete bytes, cada um contendo o padrão de bits 10101010, seguido de um delimitador de início de quadro, um byte (S, no diagrama), com o padrão 10101011.

Apesar da especificação não permitir mais de 1.024 estações em uma única rede Ethernet, os endereços de origem e destino ocupam, cada um, seis bytes, fornecendo 2^{48} endereços diferentes. Isso permite que cada interface de *hardware* Ethernet receba um endereço exclusivo de seu fabricante, garantindo que todas as estações em qualquer conjunto de redes Ethernet interconectadas tenham endereços exclusivos. O IEEE (Institute of Electrical and Electronic Engineers), dos Estados Unidos, atua como autoridade para distribuição e alocação de endereços Ethernet, reservando intervalos separados para os diversos fabricantes de interfaces de *hardware* Ethernet. Esses endereços são denominados endereços MAC, pois são usados pela camada de controle de acesso ao meio. Na verdade, os endereços MAC alocados dessa maneira também têm sido adotados como endereços exclusivos para outros tipos de rede da família IEEE 802, incluindo 802.11 (WiFi) e 802.15.1 (Bluetooth).

A área de dados contém toda a mensagem que está sendo transmitida, ou parte dela (se o comprimento da mensagem ultrapassar 1.500 bytes). O limite inferior de 46 bytes para a área de dados garante um comprimento total de, no mínimo, 64 bytes para o quadro, que é necessário para garantir que as colisões sejam detectadas por todas as estações da rede, conforme explicado a seguir.

A sequência de verificação do quadro é uma soma de verificação gerada e inserida pelo remetente e usada pelo destino para validar quadros. Os quadros que contêm somas de verificação incorretas são simplesmente eliminados pelo destino. Esse é outro exemplo de o porquê, para garantir a transmissão de uma mensagem, um protocolo de camada de transporte, como o TCP, deve confirmar o recebimento de cada pacote e retransmitir todos que não foram confirmados. A incidência de erros em redes locais é tão pequena que o uso desse método de recuperação é totalmente satisfatório quando é exigida uma garantia da entrega e, caso contrário, permite o uso de um protocolo de transporte menos dispendioso, como o UDP.

Colisões • Mesmo no tempo relativamente curto que leva para transmitir quadros, existe uma probabilidade finita de que duas estações da rede tentem transmitir mensagens simultaneamente. Se uma estação tentar transmitir um quadro sem verificar se o meio está sendo usado por outras estações, poderá ocorrer uma colisão.

A Ethernet tem três mecanismos para tratar dessa possibilidade. O primeiro é chamado de *detecção de portadora*; o *hardware* da interface de cada estação “sente” a presença de um sinal no meio, conhecido como *portadora*, em analogia com a transmissão de rádio. Quando uma estação deseja transmitir um quadro, ela espera até que nenhum sinal esteja presente no meio e depois começa a transmitir.

Infelizmente, “sentir” a presença da portadora não evita todas as colisões. A possibilidade de colisão permanece devido ao tempo finito τ para que um sinal inserido em um

ponto no meio de transmissão (viajando a aproximadamente 2×10^8 metros por segundo) se propague a todos os outros pontos. Considere duas estações, A e B, que estão prontas para transmitir quadros quase ao mesmo tempo. Se A começa a transmitir primeiro, B pode verificar o meio e não detectar a portadora em dado momento $t < \tau$, após A ter começado a transmitir. Então, B começa a transmitir, *interferindo* na transmissão de A. Tanto o quadro de A como o de B serão danificados por essa interferência.

A técnica usada para se recuperar dessa interferência é chamada *deteção de colisão*. Ao mesmo tempo em que transmite, uma estação escuta o meio, e o sinal transmitido é comparado com o recebido; se eles diferirem, significa que houve uma colisão. Quando isso acontece, a estação para de transmitir e produz um *sinal de reforço de colisão (jamming)* para garantir que todas as estações reconheçam a colisão. Conforme já mencionamos, um comprimento de quadro mínimo é necessário para garantir que as colisões sejam sempre detectadas. Se duas estações transmitirem de forma aproximadamente simultânea, a partir de extremos opostos da rede, elas não identificarão a ocorrência de uma colisão por 2τ segundos (porque o primeiro remetente ainda deverá estar transmitindo ao receber o segundo sinal). Se os quadros demorarem menos que τ para serem transmitidos, a colisão não será notada, pois as estações de envio só verão o outro quadro depois de terem terminado de transmitir o seu próprio. As estações presentes em pontos intermediários receberão os dois quadros simultaneamente, resultando em corrupção dos dados.

Após o sinal de *jamming*, todas as estações que estão transmitindo e captando cancelam o quadro corrente. As estações que estão transmitindo precisam, então, tentar retransmitir seus quadros. Agora, surge uma dificuldade maior. Se, após o sinal de *jamming*, todas as estações envolvidas na colisão tentarem retransmitir seus quadros imediatamente, outra colisão provavelmente ocorrerá. Para evitar isso, é usada uma técnica conhecida como *back-off*. Cada uma das estações envolvidas na colisão escolhe um tempo de espera $n\tau$ antes de retransmitir. O valor de n é um número inteiro, aleatório, escolhido separadamente em cada estação e limitado por uma constante L , definida no *software* de rede. Ao ocorrer uma nova colisão, o valor de L é duplicado e esse procedimento é repetido, se necessário, por até dez tentativas.

Finalmente, o *hardware* da interface na estação destino calcula a sequência de verificação e a compara com a soma de verificação transmitida no quadro. Ao usar todas essas técnicas, as estações conectadas em uma rede Ethernet são capazes de gerenciar o uso do meio sem nenhum controle ou sincronismo centralizado.

Eficiência da Ethernet • A eficiência de uma rede Ethernet é a razão entre o número de quadros transmitidos com sucesso e o número máximo teórico que poderia ser transmitido sem colisões. Isso é afetado pelo valor de τ , pois o intervalo de 2τ segundos após o início da transmissão de um quadro é a “janela de oportunidade” para colisões – nenhuma colisão pode ocorrer depois de 2τ segundos após o início da transmissão de um quadro. A eficiência também é afetada pelo número de estações presentes na rede e por seu nível de atividade.

Para um cabo de 1 km, o valor de τ é menor do que 5 microsegundos, e a probabilidade de colisões é pequena o suficiente para garantir uma alta eficiência. A Ethernet pode atingir uma utilização de canal entre 80 e 95%, embora os atrasos causados pela disputa se tornem significativos quando se ultrapassa 50% de utilização. Como a carga na rede é variável, é impossível garantir a entrega de uma determinada mensagem dentro de um tempo fixo qualquer, pois a rede poderia estar completamente carregada quando a mensagem estivesse pronta para transmissão. Contudo a probabilidade de transferir a mensagem com determinado atraso é tão boa quanto ou melhor que outras tecnologias de rede.

	<i>10Base5</i>	<i>10BaseT</i>	<i>100BaseT</i>	<i>1000BaseT</i>
Taxa de dados	10 Mbps	10 Mbps	100 Mbps	1.000 Mbps
<i>Comprimentos de segmento (máx.)</i>				
Par trançado (UTP)	100 m	100 m	100 m	25 m
Cabo coaxial (STP)	500 m	500 m	500 m	25 m
Fibra multimodo	2.000 m	2.000 m	500 m	500 m
Fibra monomodo	25.000 m	25.000 m	20.000 m	2.000 m

Figura 3.23 Distâncias e taxas de transmissão de redes Ethernet.

Medidas empíricas do desempenho de uma rede Ethernet no Xerox PARC são relatadas por Shoch e Hupp [1980] e confirmam essa análise. Na prática, a carga nas redes Ethernet usadas em sistemas distribuídos varia bastante. Muitas redes são empregadas principalmente para interações cliente-servidor assíncronas, e essas interações operam, na maior parte do tempo, sem as estações estarem esperando oportunidade para transmitir. Essa situação é bastante diferente daquela encontrada por aplicações que exigem o acesso a um grande volume de dados ou das que transportam fluxos multimídia. Nesses dois últimos casos, a rede tende a ficar sobrecarregada.

Implementações físicas • A descrição anterior define o protocolo de camada MAC para as redes Ethernet. Sua ampla adoção pelo mercado resultou no barateamento do *hardware* necessário à execução dos algoritmos exigidos para sua implementação de tal forma que ele é incluído, por padrão, em vários tipos de computadores.

Existe uma grande gama de variações para implementar fisicamente uma rede Ethernet, oferecendo uma série de contrapartidas em termos de custo e desempenho. Essas variações resultam do uso de diferentes mídias de transmissão – cabo coaxial, par de cobre trançado (semelhante à fiação telefônica) e fibra óptica – com diferentes limites de alcance de transmissão e de velocidades de sinalização, que resultam em uma largura de banda maior. O IEEE adotou vários padrões para as implementações da camada física e um esquema de atribuição de nomes para distingui-los. São usados nomes como 10Base5 e 100BaseT, com a seguinte interpretação:

$< R > < B > < L >$

Onde: R = taxa de dados, em Mbps

B = tipo de sinalização do meio (banda base ou banda larga)

L = comprimento do segmento máximo, em centenas de metros ou T (hierarquia de cabos de par trançado)

Na Figura 3.23, são tabulados a largura de banda e a distância máxima de vários padrões atualmente disponíveis assim como os tipos de cabo. As configurações que terminam com a designação T são implementadas com cabeamento UTP (Unshielded Twisted Pair) – pares de fios trançados não blindados – comuns à fiação telefônica, e normalmente são organizadas em uma hierarquia de *hubs* em que os computadores são as folhas da árvore. Nesse caso, os comprimentos de segmento dados em nossa tabela são duas vezes a distância máxima permitida de um computador para um *hub*.

Ethernet para aplicações de tempo real e com exigências de qualidade de serviço • Frequentemente, argumenta-se que o protocolo MAC Ethernet é inherentemente inconveniente para aplicativos de tempo real ou que exigem qualidade de serviço devido à inca-

pacidade de garantir um atraso máximo para a entrega de quadros. Entretanto, deve-se salientar que a maioria das instalações de Ethernet agora são em *switches*, conforme ilustrado na Figura 3.10 e descrito na Seção 3.3.7 (em vez de hubs ou cabos com uma derivação para cada conexão, como era o caso anteriormente). O uso de *switches* resulta em um segmento separado para cada *host* sem quadros transmitidos nele além daqueles endereçados a esse *host*. Portanto, se o tráfego para o *host* é proveniente de uma única origem, não há disputa pelo meio – a eficiência é de 100% e a latência é constante. A possibilidade de disputa surge apenas no interior dos *switches* e, estes, frequentemente podem ser projetados para tratar simultaneamente vários quadros. Assim, uma instalação Ethernet comutada, aquela baseada em *switches*, pouco carregada, aproxima-se dos 100% de eficiência com uma latência baixa constante e, portanto, pode ser usada com sucesso nessas áreas de aplicação críticas.

Um passo a mais na direção do suporte em tempo real para protocolos MAC do estilo Ethernet é descrito em [Rether, Pradhan e Chiueh 1998], e um esquema semelhante é implementado em uma extensão de código-fonte aberto do Linux [RTnet]. Essas estratégias de *software* tratam do problema da disputa implementando, em nível aplicativo, um protocolo cooperativo para reservar fatias de tempo para uso do meio. Isso depende da colaboração de todos os *hosts* conectados a um segmento.

3.5.2 Rede local sem fio IEEE 802.11 (WiFi)

Nesta seção, resumimos as características especiais da comunicação sem fio (*wireless*) que devem ser tratadas por uma tecnologia de rede local sem fio e explicamos como o padrão IEEE 802.11 trata delas. O padrão IEEE 802.11 estende o princípio da detecção de portadora e acesso múltiplo (CSMA), empregado pela tecnologia Ethernet (IEEE 802.3), para se adequar às características da comunicação sem fio. O padrão 802.11 se destina a suportar comunicação entre computadores localizados dentro de cerca de 150 metros uns dos outros, em velocidades de até 54 Mbps.

A Figura 3.24 ilustra parte de uma intranet que inclui uma rede local sem fio. Vários equipamentos móveis sem fio se comunicam com o restante da intranet por meio de uma estação de base, que serve como *ponto de acesso* para a rede local cabeada. Uma rede sem fio que se conecta ao mundo exterior através de uma rede cabeada convencional, por meio de um ponto de acesso, é conhecida como *rede de infraestrutura*.

Uma configuração alternativa para redes sem fio é conhecida como *rede ad hoc*. As redes *ad hoc* não contêm ponto de acesso, nem estação de base. Elas são construídas dinamicamente, como resultado da detecção mútua de dois ou mais equipamentos móveis com interfaces sem fio na mesma vizinhança. Uma rede *ad hoc* se estabelece quando, por exemplo, em uma sala, dois ou mais *notebooks* se detectam e estabelecem uma comunicação entre si. Neste caso, então, eles poderiam compartilhar arquivos, ativando um processo servidor de arquivos em uma das máquinas.

No nível físico, as redes IEEE 802.11 usam sinais de radiofrequência (nas faixas livres de licença, 2,4 GHz e 5 GHz), ou sinalização infravermelha, como forma de transmissão. A versão baseada em radiofrequência tem maior atenção comercial e vamos descrevê-la. O padrão IEEE 802.11b foi a primeira variante a ter seu uso bastante difundido. Ele opera na faixa de 2,4 GHz e suporta comunicação de dados em até 11 Mbps. Esse tipo de rede, a partir de 1999, tornou-se bastante comum em muitos escritórios, casas e locais públicos, permitindo que computadores *notebook* e PDAs acessassem dispositivos interligados em rede local ou a Internet. O padrão IEEE 802.11g é um aprimoramento mais recente do padrão 802.11b, que usa a mesma faixa de 2,4 GHz, mas utiliza uma téc-

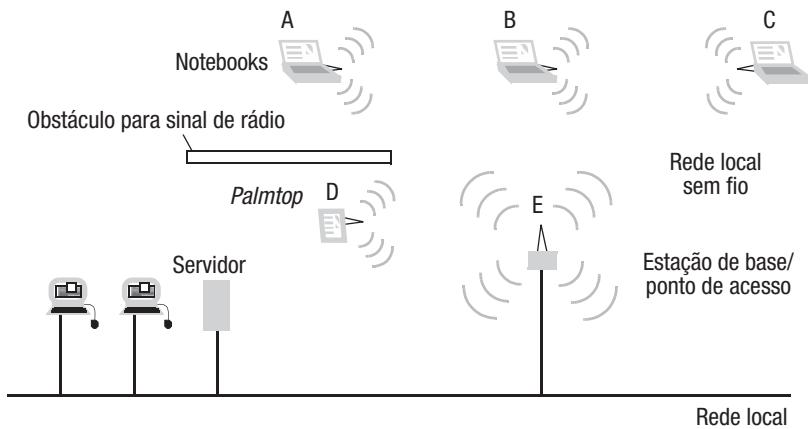


Figura 3.24 Configuração de rede local sem fio.

nica de sinalização diferente para atingir velocidades de até 54 Mbps. Finalmente, a variante 802.11a opera na faixa de 5 GHz e oferece 54 Mbps de largura de banda, mas com um alcance menor. Todas as variantes usam diversas técnicas de seleção de frequência e saltos de frequência para evitar interferência externa e mútua entre redes locais sem fio independentes, o que não vamos detalhar aqui. Em vez disso, vamos focar as alterações feitas no mecanismo de controle de acesso ao meio CSMA/CD para que, em todas as versões de 802.11, a transmissão de dados via sinal de rádio fosse possível.

Assim como o padrão Ethernet, o protocolo MAC 802.11 oferece oportunidades iguais a todas as estações para usar o canal de comunicação e uma estação transmitir diretamente para qualquer outra. Um protocolo MAC controla o uso do canal por várias estações. No que diz respeito à Ethernet, a camada MAC também executa as funções de camada física e de enlace de dados, distribuindo quadros de dados para os *hosts* de uma rede.

Vários problemas surgem do uso de ondas de rádio, em vez de fios, como meio de transmissão. Esses problemas se originam do fato de que os mecanismos de detecção da portadora e de colisão empregados nas redes Ethernet são eficazes somente quando a intensidade dos sinais é aproximadamente igual em toda a rede.

Lembremos que o objetivo da detecção de portadora é verificar se o meio está livre para que uma transmissão seja iniciada, e a detecção de colisão serve para determinar se houve interferências durante uma transmissão. Como a intensidade do sinal não é uniforme em toda a área na qual as redes locais sem fio operam, a detecção da portadora e de colisão pode falhar das seguintes maneiras:

Estações ocultas: o sinal de portadora pode não ser detectado mesmo se houver uma outra estação transmitindo. Isso está ilustrado na Figura 3.24. Se o *palmtop* D estiver transmitindo para a estação de base E, o *notebook* A poderá não captar o sinal de D, devido ao obstáculo para o sinal de rádio mostrado. Então, A poderia começar a transmitir, causando uma colisão em E, a menos que medidas fossem tomadas para evitar isso.

Desvanecimento do sinal (fading): a lei do inverso do quadrado da propagação de ondas eletromagnéticas nos diz que a intensidade dos sinais de rádio diminui rapidamente com a distância do transmissor. As estações de uma rede local sem fio podem ficar fora do alcance do sinal de rádio de outras estações da mesma rede.

Assim, na Figura 3.24, o *notebook* A talvez não possa detectar uma transmissão de C, embora cada uma delas possa transmitir com sucesso para B ou E. O desvanecimento anula tanto a detecção de portadora quanto a detecção de colisão.

Mascaramento de colisões: infelizmente, a técnica de “sentir” usada na Ethernet para detectar colisões não é muito eficiente em redes baseada em sinal de rádio. Devido à lei do inverso do quadrado, citada anteriormente, o sinal local gerado será sempre muito mais forte do que qualquer sinal originado em outro lugar, abafando a transmissão remota. Portanto, os *notebooks* A e C poderiam transmitir simultaneamente para E. Nenhum deles detectaria essa colisão, mas E receberia apenas uma transmissão truncada.

Apesar de sua falibilidade, a percepção de portadora não é dispensada nas redes IEEE 802.11; ela é ampliada pela adição de um mecanismo de *reserva de slot** no protocolo MAC. O esquema resultante é chamado de *detecção de portadora de acesso múltiplo com prevenção de colisão* (CSMA/CA).

Quando uma estação está pronta para transmitir, ela sente o meio. Se ela não detectar nenhum sinal de portadora, pode presumir que uma das seguintes condições é verdadeira:

1. o meio está disponível;
2. uma estação fora do alcance está no procedimento de solicitação de um *slot*;
3. uma estação fora do alcance está usando um *slot* que tinha reservado anteriormente.

O protocolo de reserva de *slot* envolve a troca de duas mensagens curtas (quadros) entre o emissor e o receptor. A primeira é um quadro *request to send* (RTS) do emissor para o receptor. A mensagem RTS especifica uma duração para o *slot* solicitado. O receptor responde com um quadro *clear to send* (CTS), repetindo a duração do *slot*. O efeito dessa troca de quadros é o seguinte:

- As estações dentro do alcance do emissor também receberão o quadro RTS e tomarão nota da duração.
- As estações dentro do alcance do receptor receberão o quadro CTS e tomarão nota da duração.

Como resultado, todas as estações dentro do alcance do emissor e do receptor não transmitirão pela duração do *slot* solicitado, deixando o canal livre para o emissor transmitir um quadro de dados de comprimento apropriado. Finalmente, a recepção bem-sucedida do quadro de dados é confirmada pelo receptor. Isso auxilia a tratar dos problemas de interferências externas no canal. O recurso de reserva de *slot* do protocolo MAC ajuda a evitar colisões das seguintes maneiras:

- Os quadros CTS ajudam a evitar os problemas de estação oculta e desvanecimento.
- Os quadros RTS e CTS possuem curta duração; portanto, o risco de colisões com eles é baixo. Se uma colisão ocorrer e for detectada, ou se um RTS não resultar em um CTS, o RTS será retransmitido usando um período de *back-off* aleatório, como na Ethernet.
- Quando os quadros RTS e CTS tiverem sido trocados corretamente, não deverão ocorrer colisões envolvendo os quadros de dados subsequentes, nem nos quadros

* N. de R. T.: *Slot* é o termo inglês usado para designar um encaixe ou espaço para se conectar periféricos em um barramento. Em comunicação de dados, ele assume a noção de intervalo de tempo, ou espaço temporal, proporcional à duração temporal de um quadro. Por questões de clareza no texto e de uso corrente, optamos por manter o termo original.

de confirmação, a não ser que um desvanecimento intermitente tenha impedido o outro participante de receber um deles.

Segurança • A privacidade e a integridade da comunicação é uma preocupação óbvia nas redes sem fio. Qualquer estação que esteja dentro do alcance e equipada com um receptor/transmissor poderá fazer parte da rede normalmente ou bisbilhotar as transmissões entre outras estações. A primeira tentativa para tratar dos problemas de segurança para o padrão 802.11 é intitulada WEP (Wired Equivalent Privacy). Infelizmente, o WEP é tudo, menos o que seu nome implica. Seu projeto era falho em vários aspectos, o que permitia uma invasão muito facilmente. Descreveremos suas vulnerabilidades e resumiremos o status atual relativo aos aprimoramentos na Seção 11.6.4.

3.5.3 Redes pessoais sem fio – Bluetooth IEEE 802.15.1

Bluetooth é uma tecnologia de rede pessoal sem fio que surgiu a partir da necessidade de conectar telefones móveis a PDAs, *notebooks* e outros equipamentos pessoais sem fios. Um grupo de interesse especial (SIG, Special Interest Group) de fabricantes de telefones móveis e computadores, liderados por L.M. Ericsson, desenvolveu uma especificação para uma rede pessoal sem fio (WPAN, Wireless Personal Area Network) para a transmissão de fluxos de voz digitalizada como dados [Haartsen *et al.* 1998]. A versão 1.0 do padrão Bluetooth foi publicada em 1999, e seu nome deve-se a um rei viking. Aqui, descrevemos a versão 1.1. Ela foi publicada em 2002, resolvendo alguns problemas. Então o grupo de trabalho do IEEE 802.15 o adotou como sendo o padrão 802.15.1 e publicou uma especificação para as camadas física e de enlace de dados [IEEE 2002].

As redes Bluetooth diferem substancialmente do IEEE 802.11 (WiFi), refletindo os requisitos de seu contexto de utilização e as metas de custo e de consumo de energia para os quais foram projetadas. O padrão Bluetooth foi projetado para permitir comunicação entre dispositivos pequenos e de baixo custo, como os fones de ouvido sem fio adaptados à orelha, que recebem fluxos de áudio de um telefone celular, ou interconexões entre computadores, telefones, PDAs e outros dispositivos móveis. A meta de custo era adicionar apenas cinco dólares ao preço final de um dispositivo portátil. A meta de consumo de energia era usar apenas uma pequena fração da energia total da bateria de um telefone celular ou PDA e operar por várias horas, mesmo com baterias pequenas, empregadas tipicamente em aparelhos acoplados ao corpo, como fones de ouvido.

As aplicações alvo do Bluetooth exigem menos largura de banda e um alcance de transmissão menor do que as aplicações de rede local sem fio do padrão 802.11. Isso é bom, pois o Bluetooth opera na mesma faixa de frequência livre de licença de 2,4 GHz que as redes WiFi, telefones sem fio e muitos outros sistemas de comunicação de serviços de emergência, sendo, portanto, muito ocupada. A transmissão se dá em baixa energia, saltando a uma taxa de 1.600 vezes por segundo entre 79 sub-faixas de 1 MHz da faixa de frequência permitida para minimizar os efeitos da interferência. A potência de saída dos dispositivos Bluetooth convencionais é de 1 miliwatt, fornecendo uma área de cobertura de apenas 10 metros; dispositivos de 100 miliwatts, com um alcance de até 100 metros, são permitidos para aplicações como redes domésticas. A eficiência no uso da energia é melhorada pela inclusão de um recurso de *alcance adaptativo*, que ajusta a potência da transmissão para um nível mais baixo, quando dispositivos que se comunicam estão próximos (conforme determinado pela intensidade dos sinais recebidos inicialmente).

Os nós Bluetooth se associam dinamicamente em pares, sem exigir nenhum conhecimento anterior da rede. O protocolo de associação será descrito a seguir. Após uma associação bem-sucedida, o nó iniciador tem a função de *mestre* e o outro, a de *escravo*. Uma

Piconet é uma rede associada dinamicamente, composta de um mestre e até sete escravos ativos. O mestre controla o uso do canal de comunicação, alocando *slots* de tempo para cada escravo. Um nó que esteja em mais de uma Piconet pode atuar como ponte, permitindo que os mestres se comuniquem – várias Piconets ligadas dessa maneira são chamadas de *Scatternet*. Muitos tipos de dispositivo têm a capacidade de atuar como mestre ou como escravo.

Todos os nós Bluetooth também são equipados com um endereço MAC globalmente exclusivo de 48 bits (veja a Seção 3.5.1), embora apenas o endereço MAC do mestre seja usado no protocolo. Quando um escravo se torna ativo em uma Piconet, ele recebe um endereço local temporário no intervalo de 1 a 7. O objetivo disso é reduzir o comprimento dos cabeçalhos do quadro. Além dos sete escravos ativos, uma Piconet pode conter até 255 nós *estacionados (parked)* no modo de baixa energia, esperando um sinal de ativação do mestre.

Protocolo de associação • Para economizar energia, os dispositivos permanecem no modo de suspensão ou *espera (standby)* antes que qualquer associação seja feita ou quando nenhuma comunicação recente ocorreu. No modo de espera, eles são acionados para captar mensagens de ativação em intervalos que variam de 0,64 a 2,56 segundos. Para se associar a um nó próximo conhecido (estacionado), o nó iniciador transmite uma sequência de 16 quadros de *página*, em 16 faixas de frequência, que pode ser repetida várias vezes. Para entrar em contato com qualquer nó desconhecido dentro do intervalo, o iniciador deve primeiro transmitir uma sequência de mensagens de *pergunta*. Essas sequências de transmissão podem ocupar até cerca de 5 segundos, no pior caso, levando a um tempo de associação máximo de 7 a 10 segundos.

A associação é seguida por uma troca de autenticação opcional, baseada em “fichas” de autenticação (*tokens*) fornecidas pelo usuário ou recebidas anteriormente, para garantir que a associação se dê com o nó pretendido e não com um impostor. Um escravo permanece sincronizado com o mestre, observando os quadros transmitidos regularmente por ele, mesmo quando eles não são endereçados para esse escravo. Um escravo que está inativo pode ser colocado no modo “estacionado” pelo mestre, liberando seu *slot* na Piconet para ser usado por outro nó.

Os requisitos de oferecer canais de comunicação síncronos, com qualidade de serviço adequada para a transmissão de áudio bidirecional em tempo real (por exemplo, entre um telefone e o fone de ouvido sem fio), assim como comunicação assíncrona para troca de dados, impuseram uma arquitetura de rede muito diferente do projeto de múltiplo acesso e melhor esforço das redes Ethernet e WiFi. A comunicação síncrona é obtida pelo uso de um protocolo de comunicação bidirecional simples entre um mestre e um de seus escravos, denominado enlace *síncrono orientado a conexão (SCO, Synchronous Connection Oriented)*, no qual mestre e escravo enviam quadros alternadamente. A comunicação assíncrona é obtida por um enlace *assíncrono sem conexão (ACL, Asynchronous ConnectionLess)*, no qual o mestre envia quadros de consulta periodicamente (*polling*) para os escravos e eles transmitem somente após receberem esses quadros.

Todas as variantes do protocolo Bluetooth usam quadros que respeitam a estrutura mostrada na Figura 3.25. Uma vez estabelecida uma Piconet, o *código de acesso* consiste em um preâmbulo fixo para sincronizar o emissor e o receptor e identificar o início de um *slot*, seguido de um código derivado do endereço MAC do mestre, que identifica, de forma exclusiva, a Piconet. Este último garante que os quadros sejam corretamente direcionados em situações em que existem várias Piconets sobrepostas. Como o meio provavelmente terá ruído, e a comunicação em tempo real não pode contar com retransmissões, cada bit do cabeçalho é transmitido de forma triplicada para fornecer redundância tanto da informação como da soma de verificação.

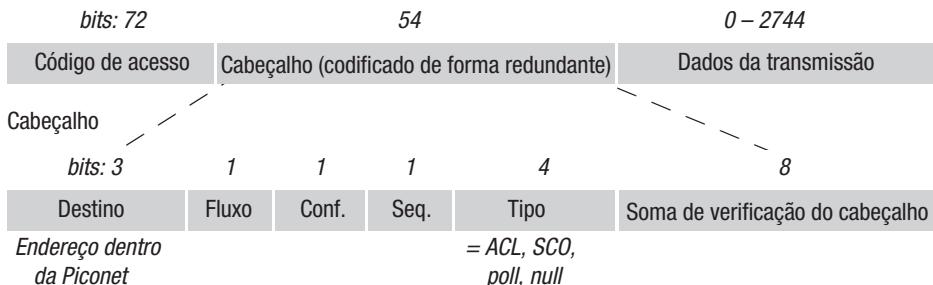


Figura 3.25 Estrutura de quadro Bluetooth.

O campo de endereço tem apenas três bits para permitir endereçamento a qualquer um dos sete escravos correntemente ativos. O endereço zero indica um *broadcast*. Existem campos para controle de fluxo, confirmação e número de sequência, cada um deles com largura equivalente a 1 bit. O bit de controle de fluxo é usado por um escravo para indicar ao mestre que seus *buffers* estão cheios; neste caso, o mestre deve esperar do escravo um quadro com um bit de confirmação diferente de zero. O bit de número de sequência é alternado entre 0 e 1 para cada novo quadro enviado para o mesmo nó; isso permite a detecção de quadros duplicados (isto é, retransmitidos).

Os enlaces SCO são usados em aplicações críticas em relação a tempo, como a transmissão de uma conversa de voz bidirecional. Os quadros têm de ser curtos para manter a latência baixa e há pouco interesse em identificar, ou retransmitir, pacotes corrompidos em tais aplicações, pois os dados retransmitidos chegariam tarde demais para serem úteis. Por isso, o SCO usa um protocolo simples e altamente redundante, no qual 80 bits de dados de voz são normalmente transmitidos de forma triplicada para produzir uma área de dados de 240 bits. Duas réplicas quaisquer de 80 bits que correspondam são aceitas como válidas.

Por outro lado, os enlaces ACL são usados para aplicações de transferência de dados, como a sincronização de agenda de endereços entre um computador e um telefone. A área de dados não é duplicada, mas pode conter uma soma de verificação interna que é verificada em nível aplicativo e, no caso de falha de retransmissão, pode ser solicitada.

Os dados são transmitidos em quadros que ocupam *slots* de tempo de duração de 625 microsssegundos, cronometrados e alocados pelo nó mestre. Cada pacote é transmitido em uma frequência diferente, em uma sequência de saltos definida pelo nó mestre. Como esses *slots* não são grandes o bastante para permitir uma área de dados de tamanho significativo, os quadros podem ser estendidos para ocupar um, três ou cinco *slots*. Essas características, e o método de transmissão física subjacente, resultam em um desempenho máximo de 1 Mbps para uma Piconet, o que acomoda até três canais duplex síncronos de 64 Kbps entre um mestre e seus escravos ou um canal para transferência de dados assíncrona em velocidades de até 723 Kbps. Esses valores são calculados para a versão mais redundante do protocolo SCO, conforme descrito anteriormente. São definidas outras variantes do protocolo que alteram a relação entre a robustez e a simplicidade (e, portanto, baixo custo computacional) dos dados triplicados para se obter um desempenho maior.

Ao contrário da maioria dos padrões de rede, o Bluetooth inclui especificações (chamadas de *perfis*) para vários protocolos em nível de aplicação, alguns dos quais são muito orientados para aplicativos em particular. O objetivo desses perfis é aumentar a probabilidade de que os dispositivos fabricados por diferentes fornecedores funcionem em conjunto. São incluídos 13 perfis de aplicativo: acesso genérico, descoberta de serviço, porta serial,

troca de objeto genérica, acesso a rede local, interligação em rede *dial-up*, fax, telefonia sem fio, sistema de intercomunicação, fones de ouvido, trocas de objetos (*push*), transferência de arquivo e sincronização. Outros perfis estão sendo elaborados, incluindo tentativas ambiciosas de transmitir música em alta qualidade e até vídeo por meio de Bluetooth.

O padrão Bluetooth ocupa um nicho especial na classe das redes locais sem fio. Ele cumpre seu objetivo de projeto de oferecer comunicação síncrona de áudio em tempo real com qualidade de serviço satisfatória (consulte o Capítulo 20 para uma discussão mais ampla sobre os problemas de qualidade do serviço), assim como permitir a transferência de dados assíncrona, usando *hardware* de custo muito baixo, compacto e portátil, com baixo consumo de energia e largura de banda limitada.

Sua principal desvantagem é o tempo que leva (até 10 segundos) para efetuar a associação de novos dispositivos. Isso impede seu uso para certas aplicações, especialmente quando os dispositivos estão se movendo uns em relação aos outros, impedindo sua utilização, por exemplo, para pagamento automatizado de pedágios em estradas ou para transmitir informações promocionais para usuários de telefonia móvel quando eles passam em frente a uma loja. Outra referência útil sobre interligação em rede Bluetooth é o livro de Bray e Sturman [2002].

A versão 2.0 do padrão Bluetooth, com desempenho de até 3 Mbps – suficiente para enviar áudio com qualidade de CD – foi lançada em 2004. Outros aprimoramentos incluíram um mecanismo de associação mais rápido e endereços de Piconet maiores. As versões 3 e 4 do padrão estavam em desenvolvimento quando este livro estava sendo produzido. A versão 3 integra um protocolo de controle Bluetooth com uma camada de transferência de dados WiFi para obter uma taxa de vazão (*throughput*) de até 24 Mbps. A versão 4 está sendo desenvolvida como uma tecnologia Bluetooth de energia ultrabaixa para equipamentos que exigem vida muito longa para a bateria.

3.6 Resumo

Focalizamos os conceitos e técnicas para a interligação de redes necessários à compreensão de sistemas distribuídos e os abordamos do ponto de vista de um projetista desse tipo de sistema. As redes de pacotes e os protocolos em camada fornecem a base da comunicação em sistemas distribuídos. As redes locais são baseadas na transmissão (difusão) de pacotes em um meio compartilhado; Ethernet é a tecnologia dominante. As redes de longa distância são baseadas na comutação de pacotes para direcionar pacotes para seus destinos em uma rede conectada. O roteamento é um mecanismo chave e, para tal, é usado uma variedade de algoritmos, dos quais vetor de distância é o mais simples, porém o mais eficaz. O controle de congestionamento é necessário para evitar o estouro dos *buffers* no receptor e nos nós intermediários.

A interligação de redes é construída dispondo-se, em camada, um protocolo de rede virtual sobre conjuntos de redes interconectadas por roteadores. Os protocolos TCP/IP permitem que os computadores se comuniquem na Internet de maneira uniforme, independentemente de estarem na mesma rede local ou em países diferentes. Os padrões da Internet incluem muitos protocolos em nível de aplicação que são convenientes para uso em aplicativos distribuídos remotos. O padrão IPv6 tem um espaço de endereçamento muito maior que o IPv4 e oferece suporte para novos requisitos de aplicativo, como qualidade do serviço e segurança, necessários para a evolução futura da Internet.

O MobileIP oferece uma infraestrutura para os usuários móveis permitindo a migração entre redes de longa distância e redes locais sem fio baseadas nos padrões IEEE 802.

Exercícios

- 3.1 Um cliente envia uma mensagem de requisição de 200 bytes para um serviço, o qual produz uma resposta contendo 5.000 bytes. Estime o tempo total gasto para completar o pedido em cada um dos casos a seguir, com as considerações de desempenho listadas abaixo:

- i) Usando comunicação não orientada a conexão (datagrama) (por exemplo, UDP).
- ii) Usando comunicação orientada a conexão (por exemplo, TCP).
- iii) O processo servidor está na mesma máquina que o cliente.

[Latência por pacote (local ou remoto,
acarretada no envio e na recepção): 5 ms
Tempo de estabelecimento da conexão (somente para TCP): 5 ms
Taxa de transferência de dados: 10 Mbps
MTU: 1.000 bytes
Tempo de processamento da requisição no servidor: 2 ms
Suponha que a rede esteja pouco carregada.]

páginas 82, 122

- 3.2 A Internet é grande demais para um roteador conter informações de roteamento para todos os destinos. Como o esquema de roteamento da Internet trata desse problema? páginas 98, 114

- 3.3 Qual é a função de um *switch* Ethernet? Quais tabelas ele mantém? páginas 105, 130

- 3.4 Faça uma tabela semelhante à Figura 3.5, descrevendo o trabalho feito pelo *software* em cada camada de protocolo, quando aplicativos Internet e o conjunto TCP/IP são implementados sobre uma rede Ethernet. páginas 94, 122, 130

- 3.5 Como o princípio fim-a-fim [Saltzer *et al.* 1984] foi aplicado no projeto da Internet? Considere como o uso de um protocolo de rede de circuito virtual, no lugar do IP, teria impacto sobre a exequibilidade da World Wide Web. páginas 61, 96, 106, [www.reed.com]

- 3.6 Podemos ter certeza de que dois computadores na Internet não têm os mesmos endereços IP? página 108

- 3.7 Compare a comunicação não orientada a conexão (UDP) e orientada a conexão (TCP) para a implementação de cada um dos seguintes protocolos nas camadas de aplicação ou de apresentação:

- i) acesso a terminal virtual (por exemplo, Telnet);
- ii) transferência de arquivo (por exemplo, FTP);
- iii) localização de usuário (por exemplo, rwho, finger);
- iv) navegação em informações (por exemplo, HTTP);
- v) chamada remota de procedimentos.

página 122

- 3.8 Explique como é possível uma sequência de pacotes transmitidos por meio de uma rede de longa distância chegarem ao seu destino em uma ordem diferente daquela em que foram enviados. Por que isso não pode acontecer em uma rede local? páginas 97, 131

- 3.9 Um problema específico que deve ser resolvido nos protocolos de acesso a terminal remoto, como Telnet, é a necessidade de transmitir eventos excepcionais, como “sinais de aborto (*kill*)” do “terminal” para o *host*, sem esperar a transmissão dos dados que está em andamento. O “sinal de aborto” deve chegar ao destino antes de quaisquer outras transmissões. Discuta a solução deste problema com protocolos orientados à conexão e não orientados à conexão. página 122

- 3.10 Quais são as desvantagens de usar *broadcast* em nível de rede para localizar recursos:
- i) em uma única Ethernet?
 - ii) em uma intranet?
- Até que ponto o *multicast* Ethernet é um aprimoramento em relação ao *broadcast*? *página 130*
- 3.11 Sugira um esquema que aprimore o MobileIP para fornecer acesso a um servidor Web em um dispositivo móvel que, às vezes, é conectado à Internet pelo telefone móvel e, outras vezes, tem uma conexão com fio com a Internet em um de vários locais. *página 120*
- 3.12 Mostre a sequência de alterações nas tabelas de roteamento da Figura 3.8 que ocorreriam (de acordo com o algoritmo RIP dado na Figura 3.9) após o enlace rotulado como 3 na Figura 3.7 ser desfeito. *páginas 98 a 101*
- 3.13 Use o diagrama da Figura 3.13 como base para uma ilustração que mostre a fragmentação e o encapsulamento de um pedido HTTP para um servidor e a resposta resultante. Suponha que o pedido seja uma mensagem HTTP curta, mas que a resposta inclua pelo menos 2.000 bytes de código HTML.
- páginas 93, 107*
- 3.14 Considere o uso de TCP em um cliente de terminal remoto Telnet. Como a entrada do teclado deve ser colocada no *buffer* do cliente? Investigue os algoritmos de Nagle e de Clark [Nagle 1984, Clark 1982] para controle de fluxo e compare-os com o algoritmo simples descrito na página 124, quando TCP for usado por:
- (a) um servidor Web;
 - (b) um aplicativo Telnet;
 - (c) um aplicativo gráfico remoto com entrada contínua de dados via mouse.
- páginas 102, 124*
- 3.15 Construa um diagrama de rede semelhante à Figura 3.10 para a rede local de sua instituição ou empresa. *página 104*
- 3.16 Descreva como você configuraria um *firewall* para proteger a rede local de sua instituição ou empresa. Quais pedidos de entrada e saída ele deve interceptar? *página 125*
- 3.17 Como um computador pessoal recentemente instalado, conectado a uma rede Ethernet, descobre os endereços IP dos servidores locais? Como ele os transforma em endereços Ethernet? *página 111*
- 3.18 Os *firewalls* podem evitar ataques de negação de serviço, como os descritos na página 112? Quais outros métodos estão disponíveis para tratar desses ataques? *páginas 112, 125*

4

Comunicação Entre Processos

- 4.1 Introdução
- 4.2 A API para protocolos Internet
- 4.3 Representação externa de dados e empacotamento
- 4.4 Comunicação por multicast (difusão seletiva)
- 4.5 Virtualização de redes: redes de sobreposição
- 4.6 Estudo de caso: MPI
- 4.7 Resumo

Este capítulo considera as características dos protocolos para comunicação entre processos em um sistema distribuído, isto é, comunicação entre processos.

A comunicação entre processos na Internet fornece tanto comunicação por datagrama como por fluxo (*stream*). As APIs para esses tipos de comunicação são apresentadas em Java, junto a uma discussão sobre seus modelos de falha. Elas fornecem blocos de construção alternativos para os protocolos de comunicação. Isso é complementado com um estudo dos protocolos para a representação de conjuntos de objetos de dados em mensagens e referências a objetos remotos. Juntos, esses serviços oferecem suporte para a construção de serviços de comunicação, conforme discutido nos dois próximos capítulos.

Todas as primitivas de comunicação entre processos discutidas anteriormente suportam comunicação ponto a ponto, sendo igualmente úteis para o envio de uma mensagem de um remetente para um grupo de destinatários. O capítulo também considera a comunicação por difusão seletiva (*multicast*), incluindo *multicast* IP e os conceitos fundamentais de confiabilidade e ordenamento de mensagens.

O *multicast* é um requisito importante para os aplicativos distribuídos e deve ser fornecido mesmo que o suporte subjacente para *multicast* IP não esteja disponível. Normalmente, isso é fornecido por uma rede de sobreposição construída sobre a rede TCP/IP subjacente. As redes de sobreposição também podem fornecer suporte para compartilhamento de arquivos, maior confiabilidade e distribuição de conteúdo.

O MPI (Message Passing Interface) é um padrão desenvolvido para fornecer uma API para um conjunto de operações de passagem de mensagens, com variantes síncronas e assíncronas.

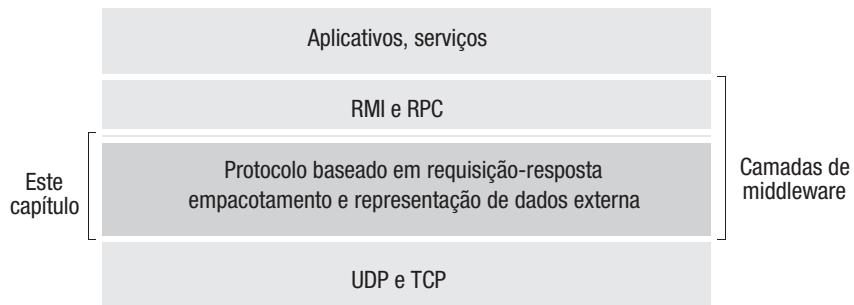


Figura 4.1 Camadas de *middleware*.

4.1 Introdução

Este capítulo e os dois seguintes estão relacionados aos aspectos da comunicação do *middleware*, embora os princípios sejam aplicados mais amplamente. Aqui, tratamos do projeto dos componentes mostrados na camada mais escura da Figura 4.1. A camada acima será discutida no Capítulo 5, que examina a invocação remota, e no Capítulo 6, que considera os paradigmas de comunicação indireta.

O Capítulo 3 discutiu os protocolos em nível de transporte UDP e TCP da Internet, sem dizer como *middlewares* e aplicativos poderiam utilizar esses protocolos. A próxima seção deste capítulo apresentará as características da comunicação entre processos e discutirá os protocolos UDP e TCP do ponto de vista do programador, apresentando a interface Java para cada um deles, junto com uma discussão sobre seus modelos de falha.

A interface de programa aplicativo para UDP fornece uma abstração de *passagem de mensagem* – a forma mais simples de comunicação entre processos. Isso permite que um processo remetente transmita uma única mensagem para um processo destino. Os pacotes independentes contendo essas mensagens são chamados de *datagramas*. Nas APIs Java e UNIX, o remetente especifica o destino usando um soquete – uma referência indireta a uma porta em particular usada pelo processo de destino que executa em um computador.

A interface de programa aplicativo para TCP fornece a abstração de um *fluxo (stream)* bidirecional entre pares de processos. A informação transmitida consiste em um fluxo contínuo de dados sem dar a noção de limites da mensagem, isto é, a noção de que ela tem um início e um fim. Os fluxos fornecem um bloco de construção para a comunicação produtor-consumidor. Um produtor e um consumidor formam um par de processos no qual a função do primeiro é produzir itens de dados e o segundo é consumi-los. Os itens de dados enviados pelo produtor para o consumidor são enfileirados na chegada do host destino até que o consumidor esteja pronto para recebê-los. O consumidor deve esperar quando nenhum item de dados estiver disponível. O produtor deve esperar, caso o armazenamento usado para conter os itens de dados enfileirados esteja cheio.

A Seção 4.3 se preocupa com o modo como os objetos e as estruturas de dados usados nos programas aplicativos podem ser transformados em uma forma conveniente para envio de mensagens pela rede, levando em consideração o fato de que diferentes computadores podem utilizar diferentes representações para tipos simples de dados. A seção também discutirá uma representação conveniente para referências a objeto em um sistema distribuído.

A Seção 4.4 abordará a comunicação por *multicast*: uma forma de comunicação entre processos na qual um processo de um grupo transmite a mesma mensagem para todos os membros do grupo de processos. Após explicar o *multicast IP*, a seção discutirá a necessidade de formas de *multicast* mais confiáveis.

A Seção 4.5 examinará o assunto cada vez mais importante das redes de sobreposição. Uma rede de sobreposição é uma rede construída sobre outra para permitir aos aplicativos direcionar mensagens para destinos não especificados por um endereço IP. As redes de sobreposição podem melhorar as redes TCP/IP por fornecer serviços alternativos, mais especializados. Elas são importantes no suporte para comunicação por *multicast* e na comunicação *peer-to-peer*.

Por fim, a Seção 4.6 apresentará o estudo de caso de um importante mecanismo de passagem de mensagens, o MPI, desenvolvido pela comunidade de computação de alto desempenho.

4.2 A API para protocolos Internet

Nesta seção, discutiremos as características gerais da comunicação entre processos e depois veremos os protocolos Internet como um exemplo, explicando como os programadores podem utilizá-los por meio de mensagens UDP ou por fluxos TCP.

A Seção 4.2.1 revê as operações de comunicação *send* e *receive* apresentadas na Seção 2.3.2, junto a uma discussão sobre como elas são sincronizadas e como os destinos das mensagens são especificados em um sistema distribuído. A Seção 4.2.2 apresenta os *soquetes*, que são empregados na interface para programação de aplicativos baseados em UDP e TCP. A Seção 4.2.3 discute o UDP e sua API Java. A Seção 4.2.4 discute o TCP e sua API Java. As APIs Java são orientadas a objetos, mas são semelhantes àquelas projetadas originalmente no sistema operacional Berkeley BSD 4.x UNIX; um estudo de caso sobre este último está disponível no site do livro [www.cdk5.net/IPC] (em inglês). Os leitores que estiverem estudando os exemplos de programação desta seção devem consultar a documentação Java *online*, ou Flanagan [2002], para ver a especificação completa das classes discutidas, que estão no pacote *java.net*.

4.2.1 As características da comunicação entre processos

A passagem de mensagens entre um par de processos pode ser suportada por duas operações de comunicação de mensagem: *send* e *receive*, definidas em termos de destinos e de mensagens. Para que um processo se comunique com outro, um deles envia (*send*) uma mensagem (uma sequência de bytes) para um destino e o outro processo, no destino, recebe (*receive*) a mensagem. Essa atividade envolve a comunicação de dados do processo remetente para o processo destino e pode implicar na sincronização dos dois processos. A Seção 4.2.3 fornece as definições para as operações *send* e *receive* na API Java para os protocolos Internet, com mais um estudo de caso sobre passagem de mensagens (MPI) oferecido na Seção 4.6.

Comunicação síncrona e assíncrona • Uma fila é associada a cada destino de mensagem. Os processos origem fazem as mensagens serem adicionadas em filas remotas, e os processos destino removem mensagens de suas filas locais. A comunicação entre os processos origem e destino pode ser síncrona ou assíncrona. Na forma síncrona de comunicação, os processos origem e destino são sincronizados a cada mensagem. Nesse caso, *send* e *receive* são operações que causam bloqueio. Quando um envio (*send*) é feito,

o processo origem (ou *thread*) é bloqueado até que a *recepção* (*receive*) correspondente seja realizada. Quando uma *recepção* é executada, o processo (ou *thread*) é bloqueado enquanto a mensagem não chegar.

Na forma *assíncrona* de comunicação, o uso da operação *send* é *não bloqueante*, no sentido de que o processo origem pode prosseguir assim que a mensagem tenha sido copiada para um *buffer local*, e a transmissão da mensagem ocorre em paralelo com o processo origem. A operação *receive* pode ter variantes com e sem bloqueio. Na variante *não bloqueante*, o processo destino prossegue sua execução após ter realizado a operação *receive*, a qual fornece um *buffer* para ser preenchido em *background*. Nesse caso, o processo deve receber separadamente uma notificação de que seu *buffer* possui dados a serem lidos, isso pode ser feito baseado em *polling* ou em interrupção.

Em um ambiente de sistema como Java, que suporta múltiplas *threads* em um único processo, a *recepção* bloqueante não tem desvantagens, pois ela pode ser executada por uma *thread*, enquanto outras *threads* do processo permanecem ativas; e a simplicidade de sincronização das *threads* destinos com a mensagem recebida é uma vantagem significativa. A comunicação não bloqueante parece ser mais eficiente, mas ela envolve uma complexidade extra no processo destino: a necessidade de ler uma mensagem recebida fora de seu fluxo normal de execução. Por esses motivos, os sistemas atuais geralmente não fornecem a forma de *recepção* não bloqueante.

Destinos de mensagem • O Capítulo 3 explicou que, nos protocolos Internet, as mensagens são enviadas para destinos identificados pelo par (*endereço IP, porta local*). Uma porta local é um destino de mensagem dentro de um computador, especificado como um valor inteiro. Uma porta tem exatamente um destino (as portas de *multicast* são uma exceção, veja a Seção 4.5.1), mas pode ter vários remetentes. Os processos podem usar várias portas para receber mensagens. Qualquer processo que saiba o número de uma porta pode enviar uma mensagem para ela. Geralmente, os servidores divulgam seus números de porta para os clientes acessarem.

Se o cliente usa um endereço IP fixo para se referir a um serviço, então esse serviço sempre deve ser executado no mesmo computador para que seu endereço permaneça válido. Para proporcionar transparência de localização, isso pode ser evitado com o uso da seguinte estratégia:

- Os programas clientes se referem aos serviços pelo nome e usam um servidor de nomes ou de associação (*binder*), veja a Seção 5.4.2, para transformar seus nomes em localizações de servidor no momento da execução. Isso permite que os serviços sejam movidos enquanto o sistema está em execução.

Confiabilidade • O Capítulo 2 definiu a comunicação confiável em termos de validade e integridade. No que diz respeito à propriedade da validade, um serviço de mensagem ponto a ponto pode ser descrito como confiável se houver garantia de que as mensagens foram entregues, independentemente da quantidade de pacotes que possam ter sido eliminados ou perdidos. Em contraste, um serviço de mensagem ponto a ponto pode ser descrito como não confiável se não houver garantia de entrega das mensagens. Quanto à integridade, as mensagens devem chegar não corrompidas e sem duplicação.

Ordenamento • Algumas aplicações exigem que as mensagens sejam entregues na *ordem de emissão* – isto é, na ordem em que foram transmitidas pela origem. A entrega de mensagens fora da ordem da origem é considerada uma falha por tais aplicações.

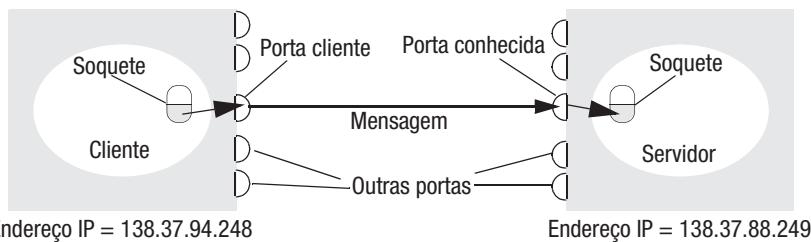


Figura 4.2 Soquetes e portas.

4.2.2 Soquetes

As duas formas de comunicação (**UDP** e **TCP**) usam a abstração de *soquete*, um ponto de destino para a comunicação entre processos. Os soquetes são originários do UNIX BSD, mas também estão presentes na maioria das versões do UNIX, incluindo o Linux, assim como no Windows e no Macintosh OS. A comunicação entre processos consiste em transmitir uma mensagem entre um soquete de um processo e um soquete de outro processo, conforme ilustrado na Figura 4.2. Para que um processo receba mensagens, seu soquete deve estar vinculado a uma porta local e a um dos endereços IP do computador em que é executado. As mensagens enviadas para um endereço IP e um número de porta em particular só podem ser recebidas por um processo cujo soquete esteja associado a esse endereço IP e a esse número de porta. Um processo pode usar o mesmo soquete para enviar e receber mensagens. Cada computador tem 2^{16} números de portas disponíveis para serem usados pelos processos para envio e recepção de mensagens. Qualquer processo pode fazer uso de várias portas para receber mensagens, mas um processo não pode compartilhar portas com outros processos no mesmo computador. (Os processos que usam *multicast* IP são uma exceção, pois compartilham portas – veja a Seção 4.4.1.) Entretanto, qualquer número de processos pode enviar mensagens para a mesma porta. Cada soquete é associado a um protocolo em particular – UDP ou TCP.

API Java para endereços Internet • A linguagem Java fornece uma classe, *InetAddress*, que representa endereços IP, para permitir a utilização dos protocolos TCP e UDP. Os usuários dessa classe se referem aos computadores pelos nomes de *host* DNS (Domain Name Service) (veja a Seção 3.4.7). As instâncias de *InetAddress* que contêm endereços IP podem ser criadas pela chamada ao método estático *InetAddress*, fornecendo-se um nome de *host* DNS como argumento. O método usa o DNS para obter o endereço IP correspondente. Por exemplo, para obter um objeto representando o endereço IP do *host* cujo nome DNS é *bruno.dcs.qmul.ac.uk*, use:

```
InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk")
```

Esse método pode disparar a exceção *UnknownHostException*. Note que o usuário da classe não precisa informar o valor explícito de um endereço IP. Na verdade, a classe encapsula os detalhes da representação dos endereços IP. Assim, a interface dessa classe não depende do número de bytes necessários para representar o endereço IP – 4 bytes no IPv4 e 16 bytes no IPv6.

4.2.3 Comunicação por datagrama UDP

Um datagrama enviado pelo protocolo UDP é transmitido de um processo origem para um processo destino sem a existência de confirmações ou novas tentativas de envio. Se ocorrer uma falha, a mensagem poderá não chegar. Um datagrama é transmitido entre processos quando um deles efetua um *send* e o outro, um *receive*. Para enviar ou receber mensagens, um processo precisa primeiro criar uma associação entre um soquete com um endereço IP e com uma porta do *host* local. Um servidor associa seu soquete a uma *porta de serviço* – que ele torna conhecida dos clientes para que estes possam enviar mensagens a ela. Um cliente vincula seu soquete a qualquer porta local livre. O método *receive* retorna, além da mensagem, o endereço IP e a porta da origem permitindo que o destinatário envie uma resposta a este.

Os parágrafos a seguir discutem alguns problemas relacionados à comunicação por datagrama:

Tamanho da mensagem: o processo destino precisa especificar um vetor de bytes de um tamanho em particular para receber as mensagens. Se a mensagem for grande demais para esse vetor, ela será truncada na chegada. O protocolo IP permite datagramas de até 2^{16} bytes (64 KB), incluindo seu cabeçalho e a área de dados. Entretanto, a maioria dos ambientes impõe uma restrição de tamanho de 8 kilobytes. Qualquer aplicativo que exija mensagens maiores do que o tamanho máximo deve fragmentá-las em porções desse tamanho. Geralmente, um aplicativo, por exemplo, o DNS, usará um tamanho que não seja excessivamente grande, mas adequado para o uso pretendido.

Bloqueio: normalmente, os soquetes fornecem operações *send* não bloqueantes e *receive* bloqueantes para comunicação por datagrama (um *receive* não bloqueante é uma opção possível em algumas implementações). A operação *send* retorna quando tiver repassado a mensagem para as camadas UDP e IP subjacentes, que são responsáveis por transmiti-la para seu destino. Ao chegar, a mensagem é posta em uma fila de recepção vinculada ao soquete associado à porta de destino. A mensagem é recuperada dessa fila quando uma operação *receive* for realizada, ou estiver com sua execução pendente, nesse soquete. Se nenhum processo tiver um soquete associado à porta de destino, as mensagens são descartadas.

O método *receive* bloqueia a execução do processo até que um datagrama seja recebido, a não ser que um tempo de espera limite tenha sido fornecido ao soquete. Se o processo que invoca o método *receive* tiver outra tarefa para fazer enquanto espera pela mensagem, ele deve tomar providências para usar *threads* separadas. As *threads* serão discutidas no Capítulo 7. Por exemplo, quando um servidor recebe uma mensagem de um cliente, normalmente uma tarefa é realizada; se o servidor for implementado usando várias *threads*, elas podem executar essas tarefas enquanto outra *thread* espera pelas mensagens de novos clientes.

Timeouts: a *recepção* bloqueante é conveniente para uso por um servidor que esteja esperando para receber requisições de seus clientes. Contudo, em algumas situações, não é adequado que um processo espere indefinidamente para receber algo, pois o processo remetente pode ter falhado ou a mensagem esperada pode ter se perdido. Para atender a tais requisitos, limites temporais (*timeouts*) podem ser configurados nos soquetes. É difícil escolher um *timeout* apropriado, porém ele deve ser grande, em comparação com o tempo exigido para transmitir uma mensagem.

Recepção anônima: o método *receive* não especifica uma origem para as mensagens. A invocação ao método *receive* obtém uma mensagem endereçada para seu

soquete, independentemente da origem. O método *receive* retorna o endereço IP e a porta local do processo origem, permitindo que o destinatário verifique de onde ela veio. Entretanto, é possível associar um soquete de datagrama a uma porta remota e a um endereço IP em particular, no caso em que se deseje apenas enviar e receber mensagens desse endereço.

Modelo de falhas • O Capítulo 2 apresentou um modelo de falhas para canais de comunicação e definiu a comunicação confiável em termos de duas propriedades: integridade e validade. A propriedade da integridade exige que as mensagens não devam estar corrompidas nem estejam duplicadas. O uso de uma soma de verificação garante que haja uma probabilidade insignificante de que qualquer mensagem recebida esteja corrompida. Os datagramas UDP sofrem das seguintes falhas:

Falhas por omissão: Ocasionalmente, mensagens podem ser descartadas devido a erros de soma de verificação ou porque não há espaço disponível no *buffer*, na origem ou no destino. Para simplificar a discussão, consideraremos as falhas por omissão de envio e por omissão de recepção (veja a Figura 2.15) como **falhas por omissão no canal de comunicação**.

Ordenamento: às vezes, as mensagens podem ser entregues em uma ordem diferente da que foram emitidas.

Os aplicativos que usam datagramas UDP podem efetuar seus próprios controles para atingir a qualidade de comunicação confiável que suas finalidades exigem. Um serviço de entrega confiável pode ser construído a partir de outro que sofra de falhas por omissão, pelo uso de confirmações. A Seção 5.2 discutirá como protocolos requisição-resposta confiáveis para comunicação cliente-servidor podem ser construídos sobre UDP.

Emprego de UDP • Para algumas aplicações, é aceitável usar um serviço que esteja exposto a falhas por omissão ocasionais. Por exemplo, o **Domain Name Service**, que pesquisa nomes DNS na Internet, é implementado sobre UDP. O **Voice Over IP (VoIP)** também é executado sobre UDP. Às vezes, os datagramas UDP são uma escolha atraente, pois não sofrem as sobrecargas necessárias à entrega de mensagens garantida. Existem três fontes de sobrecarga principais:

- a necessidade de armazenar informações de estado na origem e no destino;
- a transmissão de mensagens extras;
- a latência do remetente.

Os motivos dessas sobrecargas serão discutidos na Seção 4.2.4.

API Java para datagramas UDP • A API Java fornece comunicação por datagrama por meio de duas classes: *DatagramPacket* e *DatagramSocket*.

DatagramPacket: esta classe fornece um construtor para uma instância composta por um vetor de bytes (mensagem), o comprimento da mensagem, o endereço IP e o número da porta local do soquete de destino, como segue:

Pacote de datagrama

vetor de bytes contendo a mensagem | comprimento da mensagem | endereço IP | número da porta

Instâncias de *DatagramPacket* podem ser transmitidas entre processos quando um processo realiza uma operação *send* e outro, *receive*.

Essa classe fornece outro construtor para ser usado na recepção de mensagens. Seus argumentos especificam um vetor de bytes, para armazenamento

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
        // args fornece o conteúdo da mensagem e o nome de host do servidor
        DatagramSocket aSocket = null;
        try {
            aSocket = new DatagramSocket();
            byte[] m = args[0].getBytes();
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request =
                new DatagramPacket(m, m.length(), aHost, serverPort);
            aSocket.send(request);
            byte[] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData()));
        } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
        } catch (IOException e){System.out.println("IO: " + e.getMessage());
        } finally { if(aSocket!= null) aSocket.close();
    }
}
```

Figura 4.3 O cliente UDP envia uma mensagem para o servidor e obtém uma resposta.

da mensagem a ser recebida, e seu comprimento. Uma mensagem recebida é colocada no *DatagramPacket*, junto a seu comprimento e o endereço IP e a porta do soquete origem. A mensagem pode ser recuperada do *DatagramPacket* por meio do método *getData*. Os métodos *getPort* e *getAddress* acessam a porta e o endereço IP.

DatagramSocket: esta classe oferece mecanismos para criação de soquetes para envio e recepção de datagramas UDP. Ela fornece um construtor que recebe como argumento um número de porta, para os processos que precisam utilizar uma porta em particular, e um construtor sem argumentos que permite a obtenção dinâmica de um número de porta. Esses construtores podem provocar uma exceção *SocketException*, caso a porta já esteja em uso ou se, em ambientes UNIX, for especificada uma porta reservada (um número abaixo de 1024).

A classe *DatagramSocket* inclui os seguintes métodos:

send e *receive*: esses métodos servem para transmitir datagramas entre dois soquetes. O argumento de *send* é uma instância de *DatagramPacket* contendo uma mensagem e seu destino. O argumento de *receive* é um *DatagramPacket* vazio para se receber a mensagem, seu comprimento e origem. Os métodos *send* e *receive* podem causar exceções *IOException*.

setSoTimeout: este método permite o estabelecimento de um *timeout*. Com um *timeout* configurado, o método *receive* bloqueará pelo tempo especificado e, depois, causará uma exceção *InterruptedException*.

```

import java.net.*;
import java.io.*;
public class UDPServer{
public static void main(String args[]){
    DatagramSocket aSocket = null;
    try{
        aSocket = new DatagramSocket(6789);
        byte[] buffer = new byte[1000];
        while(true){
            DatagramPacket request = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(request);
            DatagramPacket reply = new DatagramPacket(request.getData(),
                request.getLength(), request.getAddress(), request.getPort());
            aSocket.send(reply);
        }
    } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
    } catch (IOException e) {System.out.println("IO: " + e.getMessage());
    } finally {if (aSocket!= null) aSocket.close();
    }
}
}

```

Figura 4.4 O servidor UDP recebe uma mensagem e a envia de volta para o cliente.

connect: este método é usado para contactar uma porta remota e um endereço IP em particular, no caso em que o soquete é capaz apenas de enviar e receber mensagens desse endereço.

A Figura 4.3 mostra o programa de um cliente que cria um soquete, envia uma mensagem para um servidor na porta 6789 e depois espera para receber uma resposta. Os argumentos do método *main* fornecem uma mensagem e o nome DNS do servidor. A mensagem é convertida em um vetor de bytes e o nome DNS é convertido em um endereço IP. A Figura 4.4 mostra o programa do servidor correspondente, o qual cria um soquete vinculado à porta de serviço (6789) e depois, em um laço, espera pelo recebimento de uma mensagem e responde, enviando-a de volta.

4.2.4 Comunicação por fluxo TCP

A API do protocolo TCP, que se originou do UNIX BSD 4.x, fornece a abstração de um fluxo de bytes no qual dados podem ser lidos (*receive*) e escritos (*send*). As seguintes características da rede são ocultas pela abstração de fluxo (*stream*):

Tamanho das mensagens: o aplicativo pode escolher o volume de dados que vai ser enviado ou recebido em um fluxo. Ele pode lidar com conjuntos de dados muito pequenos ou muito grandes. A implementação da camada TCP decide o volume de dados a coletar, antes de transmiti-los efetivamente como um ou mais datagramas IP. Ao chegar, os dados são entregues ao aplicativo, conforme solicitado. Se necessário, os aplicativos podem obrigar os dados a serem enviados imediatamente.

Mensagens perdidas: o protocolo TCP usa um esquema de confirmação. Como um simples exemplo desses esquemas (não é o usado no TCP), o lado remetente mantém um registro de cada datagrama IP enviado, e o lado destino confirma todas as

chegadas. Se o remetente não receber uma confirmação dentro de um tempo limite, ele retransmite a mensagem. O esquema de janela deslizante [Comer 2006], mais sofisticado, reduz o número de mensagens de confirmação exigidas.

Controle de fluxo: o protocolo TCP tenta combinar a velocidade dos processos que leem e escrevem em um fluxo. Se o processo que escreve (envia) for rápido demais para o que lê (recebe), então ele será bloqueado até que o leitor tenha consumido dados suficientes.

Duplicação e ordenamento de mensagens: identificadores de mensagem são associados a cada datagrama IP, o que permite ao destinatário detectar e rejeitar duplicatas ou reordenar as mensagens que chegam fora da ordem de emissão.

Destinos de mensagem: dois processos que estão em comunicação estabelecem uma conexão antes de poderem se comunicar por meio de um fluxo. Uma vez estabelecida a conexão, os processos simplesmente leem ou escrevem no fluxo, sem necessidade de usar endereços IP e portas. O estabelecimento de uma conexão envolve uma requisição de *connect*, do cliente para o servidor, seguido de uma requisição de *accept*, do servidor para o cliente, antes que qualquer comunicação possa ocorrer. Em um modelo cliente-servidor, isso causa uma sobrecarga considerável para cada requisição-resposta.

A API para comunicação por fluxo pressupõe que, quando dois processos estão estabelecendo uma conexão, um deles desempenha o papel de cliente e o outro desempenha o papel de servidor, mas daí em diante eles poderiam ser iguais. O papel de cliente envolve a criação de um soquete de fluxo vinculado a qualquer porta e, depois, um pedido *connect* solicitando uma conexão a um servidor, em uma determinada porta. O papel de servidor envolve a criação de um soquete de “escuta” (*listen*), vinculado à porta de serviço, para esperar que os clientes solicitem conexões. O soquete de “escuta” mantém uma fila de pedidos de conexão recebidos. Na abstração de soquete, quando o servidor aceita uma conexão, um novo soquete de fluxo é criado para que o servidor se comunique com um cliente, mantendo nesse meio-tempo seu soquete de “escuta” na porta de serviço para receber os pedidos *connect* de outros clientes.

O par de soquetes no cliente e no servidor são, na realidade, conectados por dois fluxos, um em cada direção. Assim, cada soquete tem um fluxo de entrada e um fluxo de saída. Um dos dois processos pode enviar informações para o outro, escrevendo em seu fluxo de saída, e o outro processo obtém as informações lendo seu fluxo de entrada.

Quando um aplicativo *encerra* (operação *close*) um soquete, isso indica que ele não escreverá mais nenhum dado em seu fluxo de saída. Os dados de seu *buffer* de saída são enviados para o outro lado do fluxo e colocados na fila de entrada do soquete de destino com uma indicação de que o fluxo está desfeito. O processo no destino pode ler os dados da fila, mas todas as outras leituras depois que a fila estiver vazia resultarão em uma indicação de fim de fluxo. Quando um processo termina, ou falha, todos os seus soquetes são encerrados e qualquer processo que tente se comunicar com ele descobrirá que sua conexão foi desfeita.

Os parágrafos a seguir tratam de alguns problemas importantes relacionados à comunicação por fluxo:

Correspondência de itens de dados: dois processos que estejam se comunicando precisam concordar quanto ao conteúdo dos dados transmitidos por um fluxo. Por exemplo, se um processo escreve (envia) um valor *int* em um fluxo, seguido de um valor *double*, então o outro lado deverá ler um valor *int*, seguido de um valor

double. Quando dois processos não cooperam corretamente no uso de um fluxo, o processo leitor pode causar erros ao interpretar os dados, ou ser bloqueado devido a dados insuficientes no fluxo.

Bloqueio: os dados gravados em um fluxo são mantidos em uma fila no soquete de destino. Quando um processo tentar ler dados de um canal de entrada, obterá dados da fila ou será bloqueado até que dados se tornem disponíveis. O processo que escreve dados em um fluxo pode ser bloqueado pelo mecanismo de controle de fluxo TCP, caso o soquete no outro lado já esteja armazenando o volume máximo de dados permitido pelo protocolo.

Threads: quando um servidor aceita uma conexão, ele geralmente cria uma nova *thread* para se comunicar com o novo cliente. A vantagem de usar uma *thread* separada para cada cliente é que o servidor pode bloquear quando estiver esperando por dados, sem atrasar os outros clientes. Em um ambiente em que *threads* não são suportadas, uma alternativa é testar, antes de tentar lê-lo, se a entrada está disponível; por exemplo, em um ambiente UNIX, a chamada de sistema *select* pode ser usada para esse propósito.

Modelo de falhas • Para satisfazer a propriedade da integridade da comunicação confiável, os fluxos TCP usam somas de verificação para detectar e rejeitar pacotes corrompidos, assim como números de sequência para detectar e rejeitar pacotes duplicados. Quanto à propriedade da validade, os fluxos TCP usam *timeout* e retransmissões para lidar com pacotes perdidos. Portanto, há garantia de que as mensagens sejam entregues, mesmo quando alguns dos pacotes das camadas inferiores são perdidos.

No entanto, se a perda de pacotes em uma conexão ultrapassar um limite, ou se a rede que está conectando dois processos for rompida ou se tornar seriamente congestionada, o software TCP responsável pelo envio de mensagens não receberá nenhum tipo de confirmação e, após certo tempo, declarará que a conexão está desfeita. Assim, o protocolo TCP não fornece comunicação confiável, pois não garante a entrega de mensagens diante de todas as dificuldades possíveis.

Quando uma conexão é desfeita, um processo, ao tentar ler ou escrever algo nela, receberá uma notificação de erro. Isso tem os seguintes efeitos:

- os processos que estão usando a conexão não poderão distinguir entre falha de rede e falha do processo no outro lado da conexão;
- os processos que estão se comunicando não poderão identificar se as mensagens que enviaram recentemente foram recebidas ou não.

Emprego de TCP • Muitos serviços frequentemente usados são executados em conexões TCP com números de porta reservados. Eles incluem os seguintes:

HTTP: o protocolo de transferência de hipertexto é usado para comunicação entre navegadores e servidores Web; ele será discutido na Seção 5.2.

FTP: o protocolo de transferência de arquivos permite a navegação em diretórios em um computador remoto e que arquivos sejam transferidos de um computador para outro por meio de uma conexão.

Telnet: o serviço telnet dá acesso a um computador remoto por meio de uma sessão de terminal.

SMTP: o protocolo de transferência de correio eletrônico é usado para enviar correspondência entre computadores.

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[ ]) {
        // os argumentos fornecem a mensagem e o nome de host de destino
        Socket s = null;
        try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream(s.getInputStream());
            DataOutputStream out =
                new DataOutputStream(s.getOutputStream());
            out.writeUTF(args[0]); // UTF é uma codificação de string; veja a Seção 4.3
            String data = readUTF();
            System.out.println("Received: " + data);
        }catch (UnknownHostException e){
            System.out.println("Sock:" +e.getMessage());
        } catch (EOFException e){System.out.println("EOF:" +e.getMessage());}
        } catch (IOException e){System.out.println("IO:" +e.getMessage());}
        } finally {if(s!=null) try {s.close();}catch (IOException e){/*close falhou*/}}
    }
}
```

Figura 4.5 O cliente TCP estabelece uma conexão com o servidor, envia uma requisição e recebe uma resposta.

API Java para fluxos TCP • A interface Java para fluxos TCP é fornecida pelas classes *ServerSocket* e *Socket*.

ServerSocket: esta classe se destina a ser usada por um servidor para criar um soquete em uma porta de serviço para receber requisições de *connect* dos clientes. Seu método *accept* recupera um pedido *connect* da fila ou, se a fila estiver vazia, bloqueia até que chegue um. O resultado da execução de *accept* é uma instância de *Socket* – um soquete para dar acesso aos fluxos para comunicação com o cliente.

Socket: esta classe é usada pelos dois processos de uma conexão. O cliente usa um construtor para criar um soquete, especificando o nome DNS do *host* e a porta do servidor. Esse construtor não apenas cria um soquete associado a uma porta local, mas também o conecta com o computador remoto e com o número de porta especificado. Ele pode causar uma exceção *UnknownHostException*, caso o nome de *host* esteja errado, ou uma exceção *IOException*, caso ocorra um erro de E/S.

A classe *Socket* fornece os métodos *getInputStream* e *getOutputStream* para acessar os dois fluxos associados a um soquete. Os tipos de retorno desses métodos são *InputStream* e *OutputStream*, respectivamente – classes abstratas que definem métodos para ler e escrever os bytes. Os valores de retorno podem ser usados como argumentos de construtores para fluxos de entrada e saída. Nossa exemplo usa *DataInputStream* e *DataOutputStream*, que permitem que representações binárias de tipos de dados primitivos sejam lidas e escritas de forma independente de máquina.

```

import java.net.*;
import java.io.*;
public class TCPServer {
    public static void main (String args[ ]) {
        try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
                Socket clientSocket = listenSocket.accept();
                Connection c = new Connection(clientSocket);
            }
        } catch(IOException e) {System.out.println("Listen :" +e.getMessage());}
    }
}
class Connection extends Thread {
    DataInputStream in;
    DataOutputStream out;
    Socket clientSocket;
    public Connection (Socket aClientSocket) {
        try{
            clientSocket = aClientSocket;
            in = new DataInputStream(clientSocket.getInputStream());
            out=new DataOutputStream(clientSocket.getOutputStream());
            this.start();
        } catch(IOException e) {System.out.println("Connection:" +e.getMessage());}
    }
    public void run(){
        try { // an echo server
            String data = in.readUTF();
            out.writeUTF(data);
        } catch(EOFException e) {System.out.println("EOF:" +e.getMessage());}
        } catch(IOException e) {System.out.println("IO:" +e.getMessage());}
        } finally { try {clientSocket.close();}catch (IOException e){/*close falhou*/}}
    }
}

```

Figura 4.6 O servidor TCP estabelece uma conexão para cada cliente e, em seguida, ecoa o pedido do cliente.

A Figura 4.5 mostra um programa cliente no qual os argumentos do método *main* fornecem uma mensagem e o nome DNS do servidor. O cliente cria um soquete vinculado ao nome DNS do servidor e à porta de serviço 7896. Ele produz um *DataInputStream* e um *DataOutputStream* a partir dos fluxos de entrada e saída do soquete e, em seguida, escreve a mensagem em seu fluxo de saída e espera para ler uma resposta em seu fluxo de entrada. O programa servidor da Figura 4.6 abre um soquete em sua porta de “escuta”, ou *listen*, (7896) e recebe os pedidos *connect*. A cada pedido que chega, uma nova *thread* é criada para se comunicar com o cliente. A *thread* cria um *DataInputStream* e um *DataOutputStream* a partir dos fluxos de entrada e saída de seu soquete e, em seguida, espera para ler uma mensagem e enviá-la de volta.

Como nossa mensagem consiste em uma cadeia de caracteres (*string*), os processos cliente e servidor usam o método *writeUTF* de *DataOutputStream* para escrevê-la no fluxo de saída e o método *readUTF* de *DataInputStream* para lê-la do fluxo de entrada. UTF-8 é uma codificação que representa *strings* em um formato específico, que será descrito na Seção 4.3.

Quando um processo tiver encerrado (*close*) seu soquete, não poderá mais usar seus fluxos de entrada e saída. O processo para o qual ele tiver enviado dados ainda poderá lê-los em sua fila, mas as leituras feitas após essa fila ficar vazia resultarão em uma exceção *EOFException*. As tentativas de usar um soquete já encerrado ou de escrever em um fluxo desfeito resultarão em uma exceção *IOException*.

4.3 Representação externa de dados e empacotamento

As informações armazenadas nos programas em execução são representadas como estruturas de dados – por exemplo, pela associação de um conjunto de objetos –, enquanto que as informações presentes nas mensagens são sequências puras de bytes. Independente da forma de comunicação usada, as estruturas de dados devem ser simplificadas (convertidas em uma sequência de bytes) antes da transmissão e reconstruídas na sua chegada. Os dados transmitidos nas mensagens podem corresponder a valores de tipos de dados primitivos diferentes, e nem todos os computadores armazenam tipos de dados primitivos, como os inteiros, na mesma ordem. A representação interna de números em ponto flutuante também difere entre as arquiteturas de processadores. Existem duas variantes para a ordenação de inteiros: ordem *big-endian*, na qual o byte mais significativo aparece na primeira posição, e a ordem *little-endian*, na qual ele aparece por último. Outro problema é o conjunto de códigos usado para representar caracteres: por exemplo, a maioria dos aplicativos em sistemas como o UNIX usa codificação de caracteres ASCII, com um byte por caractere, enquanto o padrão Unicode permite a representação de textos em muitos idiomas diferentes e usa dois bytes por caractere.

Um dos métodos a seguir pode ser usado para permitir que dois computadores troquem valores de dados binários:

- Os valores são convertidos para um formato externo, acordado antes da transmissão e convertidos para a forma local, na recepção; se for sabido que os dois computadores são do mesmo tipo, a conversão para o formato externo pode ser omitida.
- Os valores são transmitidos no formato do remetente, junto a uma indicação do formato usado, e o destinatário converte os valores, se necessário.

Note, entretanto, que os bytes em si nunca têm a ordem de seus bits alterada durante a transmissão. Para suportar RMI ou RPC, todo tipo de dados que possa ser passado como argumento, ou retornado como resultado, deve ser simplificado, e os valores de dados primitivos individuais, representados em um formato comum. Um padrão aceito para a representação de estruturas de dados e valores primitivos é chamado de *representação externa de dados*.

Empacotamento (marshalling) é o procedimento de pegar um conjunto de itens de dados e montá-los em uma forma conveniente para transmissão em uma mensagem. *Desempacotamento (unmarshalling)* é o procedimento inverso de desmontá-los na chegada para produzir um conjunto de itens de dados equivalente no destino. Assim, o empacotamento consiste na transformação de itens de dados estruturados e valores primitivos em uma representação externa de dados. Analogamente, o desempacotamento consiste na

geração de valores primitivos a partir de sua representação externa de dados e na reconstrução das estruturas de dados.

Serão discutidas três estratégias alternativas para representação externa de dados e empacotamento (com uma quarta considerada no Capítulo 21, quando examinarmos a estratégia do Google para a representação de dados estruturados):

- A representação comum de dados do CORBA, que está relacionada a uma representação externa dos tipos estruturados e primitivos que podem ser passados como argumentos e resultados na invocação a métodos remotos no CORBA. Ela pode ser usada por diversas linguagens de programação (veja o Capítulo 8).
- A serialização de objetos da linguagem Java, que está relacionada à simplificação e à representação externa de dados de um objeto, ou de uma árvore de objetos, que precise ser transmitida em uma mensagem ou armazenada em um disco. Isso é usado apenas pela linguagem Java.
- A XML ou Extensible Markup Language, que define um formato textual para representar dados estruturados. Ela se destinava, originalmente, a documentos contendo dados estruturados textuais autodescritivos; por exemplo, documentos Web. Porém, agora também é usada para representar dados enviados em mensagens trocadas por clientes e servidores em serviços Web (veja o Capítulo 9).

Nos dois primeiros casos, as atividades de empacotamento e desempacotamento se destinam a serem executadas por uma camada de *middleware*, sem nenhum envolvimento por parte do programador de aplicativo. Mesmo no caso da XML, que é textual e, portanto, mais acessível para tratar de codificação, o *software* para empacotar e desempacotar está disponível para praticamente todas as plataformas e ambientes de programação comumente usados. Como o empacotamento exige a consideração de todos os mínimos detalhes da representação dos componentes primitivos de objetos compostos, esse procedimento é bastante propenso a erros se executado manualmente. A compactação é outro problema que pode ser tratado no projeto de procedimentos de empacotamento gerados automaticamente.

Nas duas primeiras estratégias, os tipos de dados primitivos são empacotados em uma forma binária. Na terceira estratégia (XML), os tipos de dados primitivos são representados textualmente. A representação textual de um valor de dados geralmente será maior do que a representação binária equivalente. O protocolo HTTP, que será descrito no Capítulo 5, é outro exemplo de estratégia textual.

Outro problema com relação ao projeto de métodos de empacotamento é se os dados empacotados devem incluir informações relativas ao tipo de seu conteúdo. Por exemplo, a representação usada pelo CORBA inclui apenas os valores dos objetos transmitidos – nada a respeito de seus tipos. Por outro lado, tanto a serialização Java, como a XML, incluem informações sobre o tipo, mas de maneiras diferentes. A linguagem Java coloca todas as informações de tipo exigidas na forma serializada, mas os documentos XML podem se referir a conjuntos de nomes (com tipos) definidos externamente, chamados *espacos de nomes*.

Embora estejamos interessados no uso de representação externa de dados para os argumentos e resultados de RMIs e de RPCs, ela tem um uso mais genérico quando é empregada para representar estruturas de dados, objetos ou documentos estruturados em uma forma conveniente para transmissão em mensagens ou para armazenamento em arquivos.

Duas outras técnicas para a representação de dados externos são dignas de nota. O Google usa uma estratégia chamada de *buffers de protocolo* para capturar a representação de dados armazenados e de dados transmitidos. Essa estratégia vai ser examinada

<i>Tipo</i>	<i>Representação</i>
<i>sequence</i>	comprimento (<i>unsigned long</i>) seguido de seus elementos, em ordem
<i>string</i>	comprimento (<i>unsigned long</i>) seguido pelos caracteres que o compõem (um caractere pode ocupar mais de um <i>byte</i>)
<i>array</i>	elementos de vetor, fornecidos em ordem (nenhum comprimento especificado, pois é fixo)
<i>struct</i>	na ordem da declaração dos componentes
<i>enumerated</i>	<i>unsigned long</i> (os valores são especificados pela ordem declarada)
<i>union</i>	identificador de tipo seguido do membro selecionado

Figura 4.7 CDR do CORBA para tipos construídos.

na Seção 20.4.1. Também há um interesse considerável em JSON (JavaScript Object Notation) como uma estratégia de representação de dados externos [www.json.org]. Considerados em conjunto, os *buffers* de protocolo e JSON representam um passo na direção de estratégias mais leves para a representação de dados (quando comparadas, por exemplo, à XML).

4.3.1 Representação comum de dados (CDR) do CORBA

O CDR do CORBA é a representação externa de dados definida no CORBA 2.0 [OMG 2004a]. O CDR pode representar todos os tipos de dados que são usados como argumentos e valores de retorno em invocações a métodos remotos no CORBA. Eles consistem em 15 tipos primitivos, os quais incluem *short* (16 bits), *long* (32 bits), *unsigned short*, *unsigned long*, *float* (32 bits), *double* (64 bits), *char*, *boolean* (TRUE, FALSE), *octet* (8 bits) e *any* (que pode representar qualquer tipo primitivo ou construído), junto a uma variedade de tipos compostos. Eles estão descritos na Figura 4.7. Cada argumento ou resultado em uma invocação remota é representado por uma sequência de bytes na mensagem de invocação ou resultado.

Tipos primitivos: o CDR define uma representação para as ordens *big-endian* e *little-endian*. Os valores são transmitidos na ordem do remetente, que é especificada em cada mensagem. Se exigir uma ordem diferente, o destinatário a transforma. Por exemplo, um valor *short* de 16 bits ocupa dois bytes na mensagem e, para a ordem *big-endian*, os bits mais significativos ocupam o primeiro byte e os bits menos significativos ocupam o segundo byte. Cada valor primitivo é colocado em um índice na sequência de bytes, de acordo com seu tamanho. Suponha que a sequência de bytes seja indexada a partir de zero. Então, um valor primitivo com tamanho de n bytes (onde $n = 1, 2, 4$ ou 8) é anexado à sequência, em um índice que é um múltiplo de n no fluxo de bytes. Os valores em ponto flutuante seguem o padrão IEEE – no qual o sinal, o expoente e a parte fracionária estão nos bytes 0– n para a ordem *big-endian* e ao contrário na ordem *little-endian*. Os caracteres são representados por um código acordado entre cliente e servidor.

Tipos construídos ou compostos: os valores primitivos que compreendem cada tipo construído são adicionados a uma sequência de bytes, em uma ordem específica, como se vê na Figura 4.7.

<i>Índice na sequência de bytes</i>	<i>4 bytes</i>	<i>Observações sobre a representação</i>
0–3	5	<i>Comprimento do string</i>
4–7	"Smit"	'Smith'
8–11	"h__"	
12–15	6	<i>Comprimento do string</i>
16–19	"Lond"	'London'
20–23	"on__"	
24–27	1984	<i>unsigned long</i>

Figura 4.8 Mensagem no CDR do CORBA.

A Figura 4.8 mostra uma mensagem no CDR do CORBA contendo três campos de um *struct* cujos tipos respectivos são *string*, *string* e *unsigned long*. A figura mostra a sequência de bytes, com quatro bytes em cada linha. A representação de cada *string* consiste em um valor *unsigned long*, dando seu comprimento, seguido dos caracteres do *string*. Por simplicidade, presumimos que cada caractere ocupa apenas um byte. Os dados de comprimento variável são preenchidos com zero para que tenha uma forma padrão para permitir a comparação de dados empacotados ou de sua soma de verificação. Note que cada valor *unsigned long*, que ocupa quatro bytes, começa em um índice que é múltiplo de quatro. A figura não distingue entre as ordens *big-endian* e *little-endian*. Embora o exemplo da Figura 4.8 seja simples, o CDR do CORBA pode representar qualquer estrutura de dados composta por tipos primitivos e construídos, mas sem usar ponteiros.

Outro exemplo de representação de dados externa é o padrão XDR da Sun, que está especificado na RFC 1832 [Srinivasan 1995b] e é descrito em www.cdk5.net/IPC (em inglês). Ele foi desenvolvido pela Sun para uso nas mensagens trocadas entre clientes e servidores NFS (veja o Capítulo 13).

O tipo de um item de dados não é fornecido com a representação de dados na mensagem, seja no CDR do CORBA ou no padrão XDR da Sun. Isso porque pressupõe-se que o remetente e o destinatário tenham conhecimento comum da ordem e dos tipos dos itens de dados de uma mensagem. Em particular para RMI, ou para RPC, cada invocação de método passa argumentos de tipos específicos e o resultado é um valor de um tipo em particular.

Empacotamento no CORBA • As operações de empacotamento (*marshalling*) podem ser geradas automaticamente a partir da especificação dos tipos dos itens de dados a serem transmitidos em uma mensagem. Os tipos das estruturas de dados e os tipos dos itens de dados básicos estão descritos no IDL (Interface Definition Language) do CORBA (veja a Seção 8.3.1), que fornece uma notação para descrever os tipos dos argumentos e resultados dos métodos RMI. Por exemplo, poderíamos usar o IDL do CORBA para descrever a estrutura de dados na mensagem da Figura 4.8 como segue:

```
struct Person{
    string name;
    string place;
    unsigned long year;
};
```

O compilador da interface CORBA (veja o Capítulo 5) gera as operações de empacotamento e desempacotamento apropriadas para os argumentos e resultados dos métodos remotos a partir das definições dos tipos de seus parâmetros e resultados.

4.3.2 Serialização de objetos Java

No Java RMI, tanto objetos como valores de dados primitivos podem ser passados como argumentos e resultados de invocações de método. Um objeto é uma instância de uma classe Java. Por exemplo, a classe Java equivalente a *struct Person* definida no IDL do CORBA poderia ser:

```
public class Person implements Serializable {
    private String name;
    private String place;
    private int year;
    public Person(String aName, String aPlace, int aYear) {
        name = aName;
        place = aPlace;
        year = aYear;
    }
    // seguido dos métodos para acessar as variáveis de instância
}
```

Essa classe diz que implementa a interface *Serializable*, a qual não tem métodos. Dizer que uma classe implementa a interface *Serializable* (que é fornecida no pacote *java.io*) tem o efeito de permitir que suas instâncias sejam serializadas.

Em Java, o termo *serialização* se refere à atividade de simplificar um objeto, ou um conjunto de objetos conectados, em uma forma sequencial conveniente para ser armazenada em disco ou transmitida em uma mensagem; por exemplo, como um argumento ou resultado de uma RMI. A *desserialização* consiste em restaurar o estado de um objeto ou conjunto de objetos a partir de sua forma serializada. Pressupõe-se que o processo que faz a desserialização não tenha nenhum conhecimento anterior dos tipos dos objetos na forma serializada. Portanto, qualquer informação sobre a classe de cada objeto é incluída na forma serializada. Essa informação permite que o destinatário carregue a classe apropriada quando um objeto é desserializado.

A informação sobre uma classe consiste em seu nome e em um número de versão. O número da versão deve mudar quando forem feitas alterações na classe. Ele pode ser estabelecido pelo programador ou calculado automaticamente como uma mistura do nome da classe, suas variáveis de instância, métodos e interfaces. O processo que desserializa um objeto pode verificar se ele tem a versão correta da classe.

Os objetos Java podem conter referências para outros objetos. Quando um objeto é serializado, todos os objetos que ele referencia são serializados junto, para garantir que, quando o objeto for reconstruído, todas as suas referências possam ser completadas no destino. As referências são serializadas por meio de *identificadores (handlers)* – neste caso, o *identificador* é uma referência a um objeto dentro da forma serializada; por exemplo, o próximo número em uma sequência de valores inteiros positivos. O procedimento de serialização deve garantir que exista uma correspondência biunívoca entre referências de objeto e seus identificadores. Ele também deve garantir que cada objeto seja gravado apenas uma vez – na segunda ocorrência de um objeto, ou em ocorrências subsequentes, é gravado o identificador, em vez de o objeto.

<i>Valores serializados</i>				<i>Explicação</i>
Person	Número da versão de 8 bytes	h0		<i>Nome da classe, número da versão</i>
3	int year	java.lang.String name	java.lang.String place	<i>Número, tipo e nome das variáveis de instância</i>
1984	5 Smith	6 London	h1	<i>Valores das variáveis de instância</i>

Na realidade, a forma serializada inclui marcas adicionais de tipos; h0 e h1 são identificadores.

Figura 4.9 Indicação da forma serializada Java.

Para serializar um objeto, a informação de sua classe é escrita por extenso, seguida dos tipos e nomes de suas variáveis de instância. Se as variáveis de instância pertencerem a novas classes, então suas informações de classe também deverão ser escritas por extenso, seguidas dos tipos e nomes de suas variáveis de instância. Esse procedimento recursivo continua até que a informação da classe e os tipos e nomes das variáveis de instância de todas as classes necessárias tenham sido escritas por extenso. Cada classe recebe um identificador (*handle*) e nenhuma classe é gravada mais do que uma vez no fluxo de bytes – os identificadores são gravados em seu lugar, onde for necessário.

O conteúdo das variáveis de instância que são tipos primitivos, como inteiros, caracteres, booleanos, bytes e longos, são gravados em um formato binário portável, usando métodos da classe *ObjectOutputStream*. Os *strings* e os caracteres são gravados pelo método *writeUTF*, usando o formato Universal Transfer Format (UTF-8), o qual permite que caracteres ASCII sejam representados de forma inalterada (em um byte), enquanto os caracteres Unicode são representados por vários bytes. Os *strings* são precedidos pelo número de bytes que ocupam no fluxo.

Como exemplo, considere a serialização do objeto a seguir:

Person p = new Person("Smith", "London", 1984);

A forma serializada está ilustrada na Figura 4.9, que omite os valores dos identificadores (*handles*) e das informações de tipo que indicam objetos, classes, *strings* e outros recursos na forma serializada completa. A primeira variável de instância (1984) é um valor inteiro de comprimento fixo; a segunda e a terceira variáveis de instância são *strings* e são precedidas por seus comprimentos.

Para fazer uso da serialização Java, por exemplo, para serializar o objeto *Person*, é necessário criar uma instância da classe *ObjectOutputStream* e invocar seu método *writeObject*, passando o objeto *Person* como argumento. Para desserializar um objeto de um fluxo de dados, é necessário abrir o fluxo como *ObjectInputStream* e utilizar o método *readObject* para reconstruir o objeto original. O uso dessas duas classes é semelhante ao uso de *DataOutputStream* e *DataInputStream*, ilustrado nas Figuras 4.5 e 4.6.

A serialização e desserialização dos argumentos e resultados de invocações remotas geralmente são executadas automaticamente pela camada de *mideware*, sem nenhuma participação do programador do aplicativo. Se necessário, os programadores que tiverem requisitos especiais podem fazer sua própria versão dos métodos que leem e escrevem objetos. Para descobrir como fazer isso e para obter mais informações sobre serialização em Java, leia o exercício dirigido sobre serialização de objetos [[java.sun.com II](#)]. Outra maneira pela qual um programador pode modificar os efeitos da serialização é declarando as variáveis que não devem ser serializadas como *transientes*. Exemplos de variáveis que não devem ser serializadas são referências a recursos locais, como arquivos e soquetes.

O uso de reflexão • A linguagem Java suporta *reflexão* – a capacidade de fazer perguntas sobre as propriedades de uma classe, como os nomes e tipos de suas variáveis de instância e métodos. Isso também permite que classes sejam criadas a partir de seus nomes e que seja criado, para uma determinada classe, um construtor com argumentos de determinados tipos de dados. A reflexão torna possível fazer serialização e desserialização de maneira completamente genérica. Isso significa que não há necessidade de gerar funções de empacotamento especiais para cada tipo de objeto, conforme descrito anteriormente para CORBA. Para saber mais sobre reflexão, veja Flanagan [2002].

A serialização de objetos Java usa reflexão para descobrir o nome da classe do objeto a ser serializado e os nomes, tipos e valores de suas variáveis de instância. Isso é tudo que é necessário para a forma serializada.

Para a desserialização, o nome da classe na forma serializada é usado para criar uma classe. Isso é usado, então, para criar um novo construtor, com tipos de argumento correspondentes àqueles especificados na forma serializada. Finalmente, o novo construtor é usado para criar um novo objeto, com variáveis de instância cujos valores são lidos da forma serializada.

4.3.3 XML (Extensible Markup Language)

A XML é uma linguagem de marcação que foi definida pelo World Wide Web Consortium (W3C) para uso na Web. Em geral, o termo *linguagem de marcação* se refere a uma codificação textual que representa um texto e os detalhes de sua estrutura ou de sua aparência. Tanto a XML como a HTML foram derivadas da SGML (Standardized Generalized Markup Language) [ISO 8879], uma linguagem de marcação muito complexa. A HTML (veja a Seção 1.6) foi projetada para definir a aparência de páginas Web. A XML foi projetada para elaborar documentos estruturados para a Web.

Os itens de dados XML são rotulados com *strings* de marcação (*tags*). As *tags* são usadas para descrever a estrutura lógica dos dados e para associar pares atributo-valor às estruturas lógicas. Isto é, na XML, as *tags* estão relacionadas à estrutura do texto que englobam, em contraste com a HTML, na qual as *tags* especificam como um navegador poderia exibir o texto. Para ver uma especificação da XML, consulte as páginas sobre XML fornecidas pelo W3C, no endereço www.w3.org VII.

A XML é usada para permitir que clientes se comuniquem com serviços Web e para definir as interfaces e outras propriedades desses mesmos serviços. Entretanto, a XML também é usada de muitas outras maneiras. Ela é utilizada no arquivamento e na recuperação de sistemas – embora um repositório de arquivos XML possa ser maior do que seu equivalente binário, ele tem a vantagem de poder ser lido em qualquer computador. Outros exemplos do uso da XML incluem a especificação de interfaces com o usuário e a codificação de arquivos de configuração em sistemas operacionais.

A XML é extensível, pois os usuários podem definir suas próprias *tags*, em contraste com a HTML, que usa um conjunto fixo de *tags*. Entretanto, se um documento XML se destina a ser usado por mais de um aplicativo, os nomes das *tags* devem ser combinados entre eles. Por exemplo, os clientes normalmente usam mensagens SOAP para se comunicar com serviços Web. O SOAP (veja a Seção 9.2.1) é um formato XML cujas *tags* são publicadas para serem usadas pelos serviços Web e seus clientes.

Algumas representações externas de dados (como o CDR do CORBA) não precisam ser autodescritivas, pois pressupõe-se que o cliente e o servidor que estejam trocando uma mensagem têm conhecimento anterior da ordem e dos tipos das informações que ela contém. Entretanto, a XML foi projetada para ser usada por vários aplicativos,

```

<person id="123456789">
    <name>Smith</name>
    <place>London</place>
    <year>1984</year>
    <!-- a comment -->
</person>

```

Figura 4.10 Definição em XML da estrutura Person.

para diferentes propósitos. A capacidade de prover *tags*, junto com o uso de espaços de nomes para definir o significado das próprias *tags*, tornou isso possível. Além disso, o uso de *tags* permite que os aplicativos selecionem apenas as partes de um documento que precisam processar, e isso não será afetado pela adição de informações que são relevantes para outros aplicativos.

Os documentos XML, sendo textuais, podem ser lidos por seres humanos. Na prática, a maioria dos documentos XML é gerada e lida por *software* de processamento de XML, mas a capacidade de ler código XML pode ser útil quando as coisas dão errado. Além disso, o uso de texto torna a XML independente de qualquer plataforma específica. O uso de uma representação textual, em vez de binária, junto com o uso de *tags*, torna as mensagens muito maiores, o que faz com que elas exijam tempos de processamento e transmissão maiores, assim como mais espaço de armazenamento. Uma comparação da eficiência das mensagens usando o formato XML SOAP e o CDR do CORBA é dada na Seção 9.2.4. Entretanto, os arquivos e as mensagens podem ser compactados – a HTTP versão 1.1 permite que os dados sejam compactados, o que economiza largura de banda durante a transmissão.

Elementos e atributos XML • A Figura 4.10 mostra a definição XML da estrutura *Person* que foi usada para ilustrar o empacotamento no CDR do CORBA e em Java. Ela mostra que a XML consiste em *tags* e dados do tipo caractere. Os dados do tipo caractere, por exemplo, *Smith* ou *1984*, são os dados reais. Assim como na HTML, a estrutura de um documento XML é definida por pares de *tags* incluídas entre sinais de menor e maior. Na Figura 4.10, *<name>* e *<place>* são *tags*. Assim como na HTML, o layout geralmente pode ser usado para melhorar a legibilidade. Na XML, os comentários são denotados da mesma maneira que na HTML.

Elementos: um elemento na XML consiste em um conjunto de dados do tipo caractere delimitados por *tags* de início e de fim correspondentes. Por exemplo, um dos elementos na Figura 4.10 consiste no dado *Smith*, contido dentro do par de *tags* *<name>... </name>*. Note que o elemento com a *tag* *<name>* é incluído no elemento com o par de *tags* *<person id="123456789">... </person>*. A capacidade de um elemento de incluir outro permite a representação de dados hierárquicos – um aspecto muito importante da XML. Uma *tag* vazia não tem conteúdo e é terminada com */>*, em vez de *>*. Por exemplo, a *tag* vazia *<european/>* poderia ser incluída dentro da *tag* *<person>...</person>*.

Atributos: opcionalmente, uma *tag* de início pode incluir pares de nomes e valores de atributo associados, como em *id="123456789"*, conforme mostrado anteriormente. A sintaxe é igual à da HTML, em que um nome de atributo é seguido de um sinal de igualdade e um valor de atributo entre aspas. Múltiplos valores de atributo são separados por espaços.

É uma questão de escolha definir quais itens serão representados como elementos e quais serão representados como atributos. Um elemento geralmente é um contêiner

para dados, enquanto um atributo é usado para rotular esses dados. Em nosso exemplo, *123456789* poderia ser um identificador usado pelo aplicativo, enquanto *name*, *place* e *year* poderiam ser exibidos. Além disso, se os dados contêm subestruturas ou várias linhas, eles devem ser definidos como um elemento. Os atributos servem para valores simples.

Nomes: os nomes de *tags* e atributos na XML geralmente começam com uma letra, mas também podem começar com um sublinhado ou com dois-pontos. Os nomes continuam com letras, dígitos, hífens, sublinhados, dois-pontos ou pontos-finais. Letras maiúsculas e minúsculas são levadas em consideração, isto é, os nomes em XML são *case-sensitive*. Os nomes que começam com *xml* são reservados.

Dados binários: todas as informações nos elementos XML devem ser expressas com dados do tipo caractere, mas a questão é: como representamos elementos criptografados ou *hashing* de códigos de segurança – os quais, como veremos na Seção 9.5, são usados em XML. A resposta é que eles podem ser representados na notação *base64* [Freed e Borenstein 1996], que utiliza apenas os caracteres alfanuméricos, junto a +, / e =, que têm significado especial.

Análise (parsing) e documentos bem formados • Um documento XML deve ser bem formado – isto é, ele deve obedecer às regras sobre sua estrutura. Uma regra básica é que toda *tag* de início tem uma *tag* de fim correspondente. Outra regra básica é que todas as *tags* devem ser corretamente aninhadas, por exemplo *<x>..<y>..</y>..</x>* está correto, enquanto *<x>..<y>....</x>.. </y>*, não. Finalmente, todo documento XML deve ter um único elemento-raiz que englobe todos os outros elementos. Essas regras tornam muito simples implementar analisadores gramaticais (*parsers*) de documentos XML. Um analisador, ao ler um documento XML que não está bem formado, relata um erro fatal.

CDATA: normalmente, os analisadores de XML verificam o conteúdo dos elementos, pois ele pode conter mais estruturas aninhadas. No entanto, se o texto precisa conter um sinal de maior (ou menor) ou aspas, ele deve ser representado de uma maneira especial; por exemplo, < representa o sinal de menor. Entretanto, se uma seção não deve ser analisada por qualquer motivo – por exemplo, se contiver caracteres especiais – ela pode ser denotada como *CDATA*. Por exemplo, se um nome de lugar precisasse incluir um apóstrofo, ele poderia ser especificado de uma das duas maneiras a seguir:

```
<place> King&apos Cross </place>
<place> <![CDATA [King's Cross]]></place>
```

Prólogo XML: todo documento XML deve ter um prólogo como sua primeira linha. O prólogo deve especificar pelo menos a versão de XML que está sendo usada (que, atualmente, é a 1.0). Por exemplo:

```
<?XML version = "1.0" encoding = "UTF-8" standalone = "yes"?>
```

O prólogo também pode especificar a codificação (UTF-8 é o padrão e foi explicado na Seção 4.3.2). O termo *codificação* se refere ao conjunto de códigos usados para representar caracteres – sendo o código ASCII o melhor exemplo conhecido. Note que, no prólogo XML, o código ASCII é especificado como *us-ascii*. Outras codificações possíveis incluem ISO-8859-1 (ou Latin-1), uma codificação de oito bits cujos primeiros 128 valores são ASCII, sendo o restante usado para representar os caracteres dos idiomas da Europa Ocidental. Outras codificações de oito bits estão disponíveis para representar outros alfabetos; por exemplo, grego ou cirílico.

Um atributo adicional pode ser usado para informar se o documento é único ou se é dependente de definições externas.

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place >
    <pers:year> 1984 </pers:year>
</person>
```

Figura 4.11 Ilustração do uso de espaço de nomes na estrutura *Person*.

Espaços de nomes na XML • Tradicionalmente, os espaços de nomes fornecem uma maneira para dar escopo aos nomes. Um espaço de nomes XML é um conjunto de nomes para uma coleção de tipos e atributos de elemento, que é referenciado por um URL. Um espaço de nomes da XML pode ser usado por qualquer outro documento XML, referindo-se ao seu URL.

Qualquer elemento que utilize um espaço de nomes XML pode especificar esse espaço como um atributo chamado *xmlns*, cujo valor é um URL que faz referência a um arquivo que contém as definições do espaço de nomes. Por exemplo:

```
xmlns:pers = "http://www.cdk5.net/person"
```

O nome que aparece após *xmlns*, neste caso, *pers*, pode ser usado como prefixo para se referir aos elementos de um espaço de nomes em particular, como mostrado na Figura 4.11. O prefixo *pers* está vinculado a *http://www.cdk5.net/person* para o elemento *person*. Um espaço de nomes se aplica dentro do contexto do par de *tags* de início e fim que o engloba, a não ser que seja sobreescrita por uma nova declaração de espaço de nomes interna a si. Um documento XML pode ser definido em termos de vários espaços de nomes diferentes, cada um dos quais seria referenciado por um prefixo exclusivo.

A convenção de espaço de nomes permite que um aplicativo utilize vários conjuntos de definições externas em diferentes espaços de nomes, sem o risco de conflito de nomes.

Esquemas XML • Um esquema XML [[www.w3.org VIII](http://www.w3.org/VIII)] define os elementos e atributos que podem aparecer em um documento, o modo como os elementos são aninhados, a ordem, o número de elementos e se um elemento está vazio ou se pode conter texto. Para cada elemento, ele define o tipo e o valor padrão. A Figura 4.12 fornece um exemplo de esquema que define os tipos de dados e a estrutura da definição XML da estrutura *Person* da Figura 4.10.

A intenção é que uma definição de esquema possa ser compartilhada por muitos documentos diferentes. Um documento XML, definido de forma a obedecer um esque-

```
<xsd:schema xmlns:xsd = URL das definições de esquema XML >
    <xsd:element name= "person" type = "personType"/>
    <xsd:complexType name="personType">
        <xsd:sequence>
            <xsd:element name = "name" type="xs:string"/>
            <xsd:element name = "place" type="xs:string"/>
            <xsd:element name = "year" type="xs:positiveInteger"/>
        </xsd:sequence>
        <xsd:attribute name= "id" type = "xs:positiveInteger"/>
    </xsd:complexType>
</xsd:schema>
```

Figura 4.12 Um esquema XML para a estrutura *Person*.

ma em particular, também pode ser validado por meio desse esquema. Por exemplo, o remetente de uma mensagem SOAP pode usar um esquema XML para codificá-la, e o destinatário usará o mesmo esquema XML para validá-la e decodificá-la.

Definições de tipo de documento: As definições de tipo de documento (DTDs, Document Type Definitions) [www.w3.org VI] foram fornecidas como parte da especificação XML 1.0 para definir a estrutura de documentos XML e ainda são amplamente usadas com esse propósito. A sintaxe das DTDs é diferente do restante da XML e é bastante limitada no sentido do que pode especificar; por exemplo, ela não pode descrever tipos de dados, e suas definições são globais, impedindo que nomes de elemento sejam duplicados. As DTDs não são usadas para definir serviços Web, embora possam ser usadas para definir documentos que são transmitidos por esses.

APIs para acessar XML • Analisadores e geradores de XML estão disponíveis para as linguagens de programação mais usadas. Por exemplo, existe *software* Java para escrever objetos Java como XML (empacotamento) e para criar objetos Java a partir de tais estruturas (desempacotamento). *Software* semelhante está disponível em Python para tipos de dados e objetos Python.

4.3.4 Referências a objetos remotos

Esta seção se aplica apenas às linguagens que suportam o modelo de objeto distribuído, como Java e CORBA. Ela não é relevante para XML.

Quando um cliente invoca um método em um objeto remoto, uma mensagem de invocação é enviada para o processo servidor que contém o objeto remoto. Essa mensagem precisa especificar qual objeto em particular deve ter seu método executado. Uma *referência de objeto remoto* é o identificador de um objeto remoto, válido em todo um sistema distribuído. A referência de objeto remoto é passada na mensagem de invocação para especificar qual objeto deve ser ativado. O Capítulo 5 mostrará que as referências de objeto remoto também são passadas como argumentos e retornadas como resultados de invocações a métodos remotos, que cada objeto remoto tem uma única referência e que essas referências podem ser comparadas para ver se dizem respeito ao mesmo objeto remoto. Agora, discutiremos a representação externa das referências de objeto remoto.

As referências de objeto remoto devem ser geradas de uma forma que garanta sua exclusividade no espaço e no tempo. Em geral, podem existir muitos processos contendo objetos remotos; portanto, as referências de objeto remoto devem ser únicas entre todos os processos, nos vários computadores de um sistema distribuído. Mesmo após um objeto remoto, associado a uma determinada referência, ter sido excluído, é importante que a referência de objeto remoto não seja reutilizada, pois seus invocadores em potencial podem manter referências obsoletas. Qualquer tentativa de invocar um objeto excluído deve produzir um erro, em vez de permitir o acesso a um objeto diferente.

Existem várias maneiras de garantir a exclusividade de uma referência de objeto remoto. Uma delas é construir uma referência concatenando o endereço IP de seu computador e o número de porta do processo que a criou, com a hora de sua criação e um número de objeto local. O número de objeto local é incrementado sempre que um objeto é criado nesse processo.

Juntos, o número de porta e a hora produzem um identificador de processo exclusivo nesse computador. Com essa estratégia, as referências de objeto remoto podem ser representadas com um formato como o que aparece na Figura 4.13. Nas implementações mais simples de RMI, os objetos remotos residem no processo que os criou e existem apenas enquanto esse processo continua a ser executado. Nesses casos, a referência de objeto remoto

<i>32 bits</i>	<i>32 bits</i>	<i>32 bits</i>	<i>32 bits</i>
Endereço IP	Número de porta	Hora	Número do objeto Interface do objeto remoto

Figura 4.13 Representação de uma referência de objeto remoto.

pode ser usada como endereço para objeto remoto. Em outras palavras, as mensagens de invocação são enviadas para o endereço IP, o processo e a porta fornecidos pela referência remota.

Para permitir que os objetos remotos sejam migrados para um processo diferente em outro computador, a referência de objeto remoto não deve ser usada como endereço do objeto remoto. A Seção 8.3.3 discutirá uma forma de referência de objeto remoto que permite aos objetos serem invocados em diferentes servidores enquanto existirem.

Os sistemas *peer-to-peer*, Pastry e Tapestry, descritos no Capítulo 10, usam uma forma de objeto remoto que é completamente independente da localização. As mensagens são direcionadas para recursos por meio de um algoritmo de roteamento distribuído.

O último campo da referência de objeto remoto mostrada na Figura 4.13 contém informações sobre a interface do objeto remoto; por exemplo, o nome da interface. Essas informações são importantes para todo processo que recebe uma referência de objeto remoto como argumento ou resultado de uma invocação remota, pois ele precisa conhecer os métodos oferecidos pelo objeto remoto. Esse ponto será abordado novamente na Seção 5.4.2.

4.4 Comunicação por multicast (difusão seletiva)

A troca de mensagens aos pares não é o melhor modelo para a comunicação de um processo com um grupo de outros processos, como o que ocorre, por exemplo, quando um serviço é implementado por meio de diversos processos em computadores diferentes para fornecer tolerância a falhas ou melhorar a disponibilidade. Nestes casos, o emprego de *multicast* é mais apropriado – trata-se de uma operação que permite o envio de uma única mensagem para cada um dos membros de um grupo de processos de tal forma que membros participantes do grupo ficam totalmente transparentes para o remetente. Existem diversas possibilidades para o comportamento desejado de *multicast*. A mais simples não fornece garantias a respeito da entrega ou do ordenamento das mensagens.

As mensagens *multicast* fornecem uma infraestrutura útil para a construção de sistemas distribuídos com as seguintes características:

1. *Tolerância à falha baseada em serviços replicados*: um serviço replicado consiste em um grupo de servidores. As requisições do cliente são difundidas para todos os membros do grupo, cada um dos quais executando uma operação idêntica. Mesmo quando alguns dos membros falham, os clientes ainda podem ser atendidos.
2. *Localização de servidores de descoberta na interligação em rede espontânea*: a Seção 1.3.2 discute os serviços de descoberta para interligação em rede espontânea. Mensagens *multicast* podem ser usadas por servidores e clientes para localizar os serviços de descoberta disponíveis, para registrar suas interfaces ou para pesquisar as interfaces de outros serviços no sistema distribuído.
3. *Melhor desempenho através da replicação de dados*: os dados são replicados para aumentar o desempenho de um serviço – em alguns casos, as réplicas são postas

nos computadores dos usuários. Sempre que os dados mudam, o novo valor é enviado por *multicast* para os processos que gerenciam as réplicas.

4. *Propagação de notificações de evento:* o *multicast* para um grupo pode ser usado para notificar os processos de quando algo acontece. Por exemplo, no Facebook, quando alguém muda seu status, todos os seus amigos recebem notificações. Do mesmo modo, os protocolos de publicar-assinar podem fazer uso de *multicast* de grupo para disseminar eventos para os assinantes.

A seguir, apresentamos o *multicast IP* para, depois, examinarmos as necessidades do uso de comunicação em grupo para ver quais delas são atendidas pelo *multicast IP*. Para as que não são, propomos mais algumas propriedades dos protocolos de comunicação em grupo, além daquelas fornecidas pelo *multicast IP*.

4.4.1 Multicast IP – uma implementação de comunicação por difusão seletiva

Esta seção discute o *multicast IP* e apresenta a API Java que suporta essa funcionalidade por meio da classe *MulticastSocket*.

Multicast IP • O *multicast IP* permite que o remetente transmita um único datagrama IP para um conjunto de computadores que formam um grupo de *multicast*. O remetente não conhece as identidades dos destinatários individuais nem o tamanho do grupo. Um grupo *multicast* é especificado por um endereço IP classe D (veja a Figura 3.15) – isto é, um endereço cujos primeiros 4 bits são 1110 no protocolo IPv4. Note que os datagramas IP são endereçados para computadores – as portas pertencem aos níveis TCP e UDP.

O fato de ser membro de um grupo *multicast* permite a um computador receber datagramas IP enviados para o grupo. A participação como membro de grupos *multicast* é dinâmica, permitindo aos computadores entrarem ou saírem a qualquer momento e participarem de um número arbitrário de grupos. É possível enviar datagramas para um grupo *multicast* sem ser membro.

Para a programação de aplicativos, o *multicast IP* está disponível apenas por meio de UDP. Um programa aplicativo faz uso de *multicast* enviando datagramas UDP com endereços *multicast* e números de porta normais. Um aplicativo se junta a um grupo *multicast* fazendo seu soquete se unir ao grupo. Em nível IP, um computador pertence a um grupo *multicast* quando um ou mais de seus processos tem soquetes pertencentes a esse grupo. Quando uma mensagem *multicast* chega em um computador, cópias são encaminhadas para todos os soquetes locais que tiverem se juntado ao endereço de *multicast* especificado e estejam vinculados ao número de porta especificado. Os detalhes a seguir são específicos do protocolo IPv4:

Roteadores multicast: os datagramas IP podem ser enviados em *multicast* em uma rede local e na Internet. As transmissões locais exploram a capacidade de *multicast* da rede local, por exemplo, de uma Ethernet. Na Internet, fazem uso de roteadores com suporte a *multicast*, os quais encaminham um único datagrama para roteadores membros em outras redes, onde são novamente transmitidos para os membros locais. Para limitar a distância da propagação de um datagrama *multicast*, o remetente pode especificar o número de roteadores pelos quais pode passar – o que é chamado tempo de vida (*time to live*) ou, abreviadamente, TTL. Para entender como os roteadores sabem quais outros roteadores possuem membros de um grupo *multicast*, veja Comer [2007].

Alocação de endereços multicast: conforme discutido no Capítulo 3, os endereços da Classe D (isto é, endereços no intervalo 224.0.0.0 a 239.255.255.255) são reservados para tráfego *multicast* e são gerenciados globalmente pelo IANA (Internet Assigned Numbers Authority). O gerenciamento desse espaço de endereçamento é revisto anualmente, com a prática atual documentada na RPC 3171 [Albanna *et al.* 2001]. Esse documento define um particionamento do espaço de endereçamento em vários blocos, incluindo:

- Bloco de controle de rede local (224.0.0.0 a 224.0.0.225), para tráfego *multicast* dentro de determinada rede local.
- Bloco de controle de Internet (224.0.1.0 a 224.0.1.225).
- Bloco de controle *ad hoc* (224.0.2.0 a 224.0.255.0) para tráfego que não se encaixa em nenhum outro bloco.
- Bloco de escopo administrativo (239.0.0.0 a 239.255.255.255), que é usado para implementar mecanismo de escopo para tráfego *multicast* (para restringir a propagação).

Os endereços *multicast* podem ser permanentes ou temporários. Existem grupos permanentes mesmo quando não existem membros – seus endereços são atribuídos pela autoridade da Internet e abrangem os vários blocos mencionados anteriormente. Por exemplo, 224.0.1.1 no bloco Internet é reservado para o protocolo NTP (Network Time Protocol), conforme discutido no Capítulo 14, e o intervalo de 224.0.6.000 a 224.0.6.127 no bloco *ad hoc* é reservado para o projeto ISIS (consulte os Capítulos 6 e 18). Existem endereços reservados para diversos propósitos, desde protocolos específicos da Internet até determinadas organizações que fazem muito uso de tráfego *multicast*, incluindo emissoras de multimídia e instituições financeiras. Uma lista completa dos endereços reservados pode ser encontrada no site do IANA [www.iana.org II].

O restante dos endereços *multicast* está disponível para os grupos temporários, os quais devem ser criados antes de serem usados e que deixam de existir quando todos os membros tiverem saído. Quando um grupo temporário é criado, ele exige um endereço *multicast* livre para evitar participação acidental em um grupo já existente. O *multicast* provido pelo protocolo IP não trata desse problema diretamente. Se for usado de forma local, soluções relativamente simples são possíveis, por exemplo, configurando o TTL com um valor pequeno, tornando improvável a escolha do mesmo endereço de outro grupo. Entretanto, os programas que usam *multicast* IP em toda a Internet exigem uma solução mais sofisticada para o problema. A RFC 2908 [Thaler *et al.* 2000] descreve uma arquitetura de alocação de endereços *multicast* (MALLOC, Multicast Address Allocation Architecture) para aplicativos em nível de Internet, a qual aloca endereços exclusivos por determinado período e em determinado escopo. Assim, a proposta está intrinsecamente ligada aos mecanismos de escopo mencionados anteriormente. É adotada uma solução cliente-servidor, por meio da qual os clientes solicitam um endereço *multicast* de um servidor de alocação de endereços *multicast* (MAAS, Multicast Address Allocation Server), o qual deve, então, comunicar-se entre os domínios para garantir que as alocações sejam exclusivas por determinado tempo e escopo.

Modelo de falhas para datagramas multicast • No IP, o envio de datagramas *multicast* tem as mesmas características de falhas dos datagramas UDP – isto é, sofre de falhas por omissão. O efeito sobre um *multicast* é que não há garantia de que as mensagens sejam

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
    public static void main(String args[]){
        // args fornece o conteúdo da mensagem e o grupo multicast de destino (por exemplo, "228.5.6.7")
        MulticastSocket s =null;
        try {
            InetAddress group = InetAddress.getByName(args[1]);
            s = new MulticastSocket(6789);
            s.joinGroup(group);
            byte [] m = args[0].getBytes();
            DatagramPacket messageOut =
                new DatagramPacket(m, m.length, group, 6789);
            s.send(messageOut);
            byte[] buffer = new byte[1000];
            for(int i=0; i< 3; i++) { // obtém mensagens de outros participantes do grupo
                DatagramPacket messageIn =
                    new DatagramPacket(buffer, buffer.length);
                s.receive(messageIn);
                System.out.println("Received:" + new String(messageIn.getData()));
            }
            s.leaveGroup(group);
        } catch (SocketException e){System.out.println("Socket: " + e.getMessage());
        } catch (IOException e){System.out.println("IO: " + e.getMessage());
        } finally { if(s != null) s.close();}
    }
}
```

Figura 4.14 Um processo se une a um grupo *multicast* para enviar e receber datagramas.

entregues para um membro do grupo em particular, mesmo em face de uma única falha por omissão; ou seja, alguns dos membros do grupo, mas não todos, podem recebê-las. Como não há garantias de que uma mensagem será entregue para um membro de um grupo, esse tipo de comunicação é denominado de *multicast não confiável*. O *multicast confiável* será discutido no Capítulo 15.

API Java para multicast IP • A API Java fornece uma interface de datagrama para *multicast* IP por meio da classe *MulticastSocket*, que é uma subclasse de *DatagramSocket* com a capacidade adicional de se unir a grupos *multicast*. A classe *MulticastSocket* fornece dois construtores alternativos, permitindo que soquetes sejam criados de forma a usar uma porta local especificada (como, por exemplo, a 6789, ilustrada na Figura 4.14) ou qualquer porta local livre. Um processo pode se unir a um grupo *multicast*, invocando o método *joinGroup* em seu soquete. Efetivamente, o soquete se junta a um grupo *multicast* em determinada porta e receberá, nessa porta, os datagramas enviados para este grupo por processos existentes em outros computadores. Um processo pode sair de um grupo especificado invocando o método *leaveGroup* em seu soquete *multicast*.

No exemplo da Figura 4.14, os argumentos do método *main* especificam uma mensagem e o endereço *multicast* de um grupo (por exemplo, “228.5.6.7”). Após se unir a

esse grupo *multicast*, o processo cria uma instância de *DatagramPacket* contendo a mensagem e a envia por intermédio de seu soquete *multicast* para o grupo *multicast* na porta 6789. Depois disso, por meio desse mesmo soquete, o processo recebe três mensagens *multicast* destinadas a esse grupo e a essa porta. Quando várias instâncias desse programa são executadas simultaneamente em diferentes computadores, todas elas se unem ao mesmo grupo e cada uma deve receber sua própria mensagem e as mensagens daqueles que se uniram depois dela.

A API Java permite que o TTL seja configurado para um soquete *multicast* por meio do método *setTimeToLive*. O padrão é 1, permitindo que o *multicast* se propague apenas na rede local.

Um aplicativo implementado sobre *multicast* IP pode usar mais de uma porta. Por exemplo, o aplicativo MultiTalk [mbone], que permite a grupos de usuários manterem conversas baseadas em texto, possui uma porta para enviar e receber dados e outra para trocar dados de controle.

4.4.2 Confiabilidade e ordenamento

A seção anterior expôs o modelo de falhas do *multicast* IP, isto é, ele sofre de falhas por omissão. Para os envios *multicast* feitos em uma rede local que possui suporte nativo para que um único datagrama chegue a vários destinos, qualquer um deles pode, individualmente, descartar a mensagem porque seu *buffer* está cheio. Além disso, um datagrama enviado de um roteador *multicast* para outro pode ser perdido, impedindo, assim, que todos os destinos que estejam além desse roteador recebam a mensagem.

Outro fator é que qualquer processo pode falhar. Se um roteador *multicast* falhar, os membros do grupo que estiverem além desse roteador não receberão a mensagem, embora os membros locais possam receber.

O ordenamento é outro problema. Os datagramas IP enviados por várias redes interligadas não chegam necessariamente na ordem em que foram emitidos, com o possível efeito de que alguns membros do grupo recebam os datagramas de um único remetente em uma ordem diferente dos outros membros. Além disso, as mensagens enviadas por dois processos diferentes não chegarão necessariamente na mesma ordem em todos os membros do grupo.

Alguns exemplos dos efeitos da confiabilidade e do ordenamento • Agora, consideremos o efeito da semântica da falha no *multicast* IP nos quatro exemplos de uso de replicação da introdução da Seção 4.4.

1. *Tolerância a falhas baseada em serviços replicados*: considere um serviço replicado que consiste nos membros de um grupo de servidores que começam no mesmo estado inicial e sempre executam as mesmas operações, na mesma ordem, de modo a permanecerem consistentes uns com os outros. Essa aplicação *multicast* impõe que todas as réplicas, ou nenhuma delas, devam receber cada pedido para executar uma operação – se uma perder um pedido, ela se tornará inconsistente com relação às outras. Na maioria dos casos, esse serviço exige que todos os membros recebam as mensagens de requisição na mesma ordem dos outros.
2. *Localização dos servidores de descoberta na interligação em rede espontânea*: desde que qualquer processo que queira localizar os servidores de descoberta faça, após sua inicialização, o envio periódico de requisições em *multicast*, uma requisi-

ção ocasionalmente perdida não será um problema na localização de um servidor de descoberta. Na verdade, o Jini usa *multicast* IP em seu protocolo de localização dos servidores de descoberta. Isso será descrito na Seção 19.2.1.

3. *Melhor desempenho através de dados replicados*: considere o caso em que os próprios dados replicados, em vez de as operações sobre eles, são distribuídos por meio de mensagens de *multicast*. O efeito de mensagens perdidas e ordem inconsistente dependeria do método de replicação e da importância de todas as réplicas estarem totalmente atualizadas.
4. *Propagação de notificações de evento*: o aplicativo em particular determina a qualidade exigida do *multicast*. Por exemplo, os serviços de pesquisa Jini utilizam *multicast* IP para anunciar sua existência (veja a Seção 19.2.1).

Esses exemplos sugerem que algumas aplicações exigem um protocolo *multicast* que seja mais confiável do que o oferecido pelo IP. Portanto, há necessidade de um *multicast confiável* – no qual qualquer mensagem transmitida ou é recebida por todos os membros de um grupo ou não é recebida por nenhum deles. Os exemplos também sugerem que algumas aplicações têm forte necessidade de ordem, cujo máximo rigor é chamado de *multicast totalmente ordenado*, em que todas as mensagens transmitidas para um grupo chegam a todos os membros na mesma ordem.

O Capítulo 15 definirá e mostrará como implementar *multicast confiável* e várias garantias de ordenamento, incluindo o totalmente ordenado.

4.5 Virtualização de redes: redes de sobreposição

A vantagem dos protocolos de comunicação da Internet é que eles fornecem, por intermédio de suas APIs (Seção 4.2), um conjunto muito eficiente de elementos básicos para a construção de *software* distribuído. Contudo, uma crescente variedade de diferentes tipos de aplicativo (incluindo, por exemplo, o compartilhamento de arquivos *peer-to-peer* e o Skype) coexistem na Internet. Seria impraticável tentar alterar os protocolos da Internet de acordo com cada um dos muitos aplicativos executados por meio deles – o que poderia melhorar um, poderia ser prejudicial para outro. Além disso, o serviço de transporte de datagramas IP é implementado por um grande e sempre crescente número de tecnologias de rede. Esses dois fatores têm provocado o interesse na virtualização das redes.

A virtualização de redes [Petersen *et al.* 2005] ocupa-se com a construção de muitas redes virtuais diferentes sobre uma rede já existente, como a Internet. Cada rede virtual pode ser projetada para suportar um aplicativo distribuído em particular. Por exemplo, uma rede virtual poderia suportar *streaming* de multimídia, como iPlayer da BBC, BoxeeTV [boxee.tv] ou Hulu [hulu.com], e coexistir com outra que suportasse um game *online* para vários jogadores, ambas funcionando na mesma rede subjacente. Isso sugere uma resposta para o dilema levantado pelo princípio fim-a-fim de Salzer (consulte a Seção 2.3.3): uma rede virtual específica para um aplicativo pode ser construída sobre uma rede já existente e ser otimizada para esse aplicativo em particular, sem alterar as características da rede subjacente.

O Capítulo 3 mostrou que as redes de computador possuem esquemas de endereçamento, protocolos e algoritmos de roteamento; da mesma maneira, cada rede virtual tem seu próprio esquema de endereçamento, protocolos e algoritmos de roteamento específicos, mas redefinidos para satisfazer as necessidades de tipos de aplicativo em particular.

4.5.1 Redes de sobreposição

Uma rede de sobreposição (*overlay*) é uma rede virtual consistindo em nós e enlaces virtuais, a qual fica sobre uma rede subjacente (como uma rede IP) e oferece algo que de outro modo não é fornecido:

- um serviço personalizado de acordo com as necessidades de um tipo de aplicativo ou um serviço de nível mais alto em particular, como distribuição de conteúdo multimídia;
- funcionamento mais eficiente em determinado ambiente interligado em rede; por exemplo, roteamento em uma rede *ad hoc*;
- um recurso adicional; por exemplo, comunicação por *multicast* ou segura.

Isso leva a uma grande variedade de tipos de sobreposição, conforme mostra a Figura 4.15. As redes de sobreposição têm as seguintes vantagens:

- Elas permitem a definição de novos serviços de rede sem exigir mudanças na rede subjacente – um ponto crucial, dado o nível de padronização nessa área e as dificuldades de correção da funcionalidade de roteador subjacente.
- Elas estimulam a experimentação com serviços de rede e a personalização de serviços para tipos de aplicativo específicos.
- Várias sobreposições podem ser definidas e coexistir, sendo o resultado final uma arquitetura de rede mais aberta e extensível.

As desvantagens são que as redes de sobreposição introduzem um nível extra de indireção (e portanto podem acarretar queda no desempenho) e aumentam a complexidade dos serviços de rede, quando comparadas, por exemplo, com a arquitetura relativamente simples das redes TCP/IP.

As redes de sobreposição podem ser relacionadas ao conhecido conceito de camadas lógicas (conforme apresentado nos Capítulos 2 e 3). Sobreposições são camadas lógicas, mas que existem fora da arquitetura padrão (como a pilha TCP/IP) e exploram os graus de liberdade resultantes. Em particular, os desenvolvedores de sobreposição estão livres para redefinir os elementos básicos de uma rede, conforme mencionado anteriormente, inclusive o modo de endereçamento, os protocolos empregados e a estratégia de roteamento, frequentemente introduzindo estratégias radicalmente diferentes, mais personalizadas para os tipos de aplicativo específicos dos ambientes operacionais. Por exemplo, as tabelas de *hashing* distribuídas introduzem um estilo de endereçamento baseado em um espaço de chaves e também constróem uma topologia de maneira tal que um nó ou possui a chave ou tem uma referência para um nó que está mais próximo do proprietário da chave. Esse estilo de roteamento é conhecido como roteamento baseado em chave. A topologia aparece mais comumente na forma de anel.

Exemplificamos o uso bem-sucedido de uma rede de sobreposição discutindo o Skype. Mais exemplos de sobreposições vão ser dados ao longo de todo o livro. Por exemplo, o Capítulo 10 apresenta detalhes dos protocolos e das estruturas adotadas pelo compartilhamento de arquivos *peer-to-peer*, junto a mais informações sobre as tabelas de *hashing* distribuídas. O Capítulo 19 considera as redes *ad hoc* sem fio e as redes tolerantes à interrupção no contexto da computação móvel e ubíqua, e o Capítulo 20 examina o suporte de redes de sobreposição para *streaming* de multimídia.

Motivação	Tipo	Descrição
<i>Adequado às necessidades do aplicativo</i>	Tabelas de <i>hashing</i> distribuídas	Uma das classes mais importantes de redes de sobreposição, oferecendo um serviço que gerencia um mapeamento de chaves para valores em um número potencialmente grande de nós, de maneira completamente descentralizada (semelhante a uma tabela de <i>hashing</i> padrão, mas em um ambiente de rede).
	Compartilhamento de arquivos <i>peer-to-peer</i>	Estruturas de sobreposição que se concentram na construção de mecanismos de endereçamento e roteamento personalizados para suportar a descoberta cooperativa e o uso de arquivos (por exemplo, <i>download</i>).
	Redes de distribuição de conteúdo	Sobreposições que incluem diversas estratégias de replicação, uso de cache e posicionamento para fornecer desempenho aprimorado em termos de entrega de conteúdo para usuários web; usados para aceleração na web e para oferecer o desempenho em tempo real exigido por fluxos de vídeo [www.kontiki.com].
<i>Adequado ao estilo de rede</i>	Redes <i>ad hoc</i> sem fio	Redes de sobreposição que fornecem protocolos de roteamento personalizados para redes <i>ad hoc</i> sem fio, incluindo esquemas proativos que efetivamente constroem uma topologia de roteamento sobre nós subjacentes e esquemas reativos que estabelecem rotas sob demanda, normalmente suportadas por inundação.
	Redes tolerantes a rompimento	Sobreposições projetadas para operar em ambientes hostis que sofrem de falhas significativas de nó ou enlace e, potencialmente, grandes atrasos.
<i>Oferecimento de recursos adicionais</i>	<i>Multicast</i>	Um dos primeiros usos das redes de sobreposição na Internet, fornecendo acesso a serviços <i>multicast</i> em que não há roteadores <i>multicast</i> ; complementa o trabalho de Van Jacobson, Deering e Casner, com sua implementação do MBone (ou <i>Multicast Backbone</i>) [mbone].
	Resiliência	Busca melhoria na robustez e na disponibilidade de caminhos de Internet [nms.csail.mit.edu].
	Segurança	Redes de sobreposição que oferecem maior segurança sobre a rede IP subjacente, incluem redes privadas virtuais, por exemplo, conforme discutido na Seção 3.4.8.

Figura 4.15 Tipos de sobreposição.

4.5.2 Skype: um exemplo de rede de sobreposição

O Skype é um aplicativo *peer-to-peer* que oferece voz sobre IP (VoIP). Possui também troca de mensagens instantâneas, videoconferência e interfaces para o serviço de telefonia padrão, por meio de SkypeIn e SkypeOut. O software foi desenvolvido

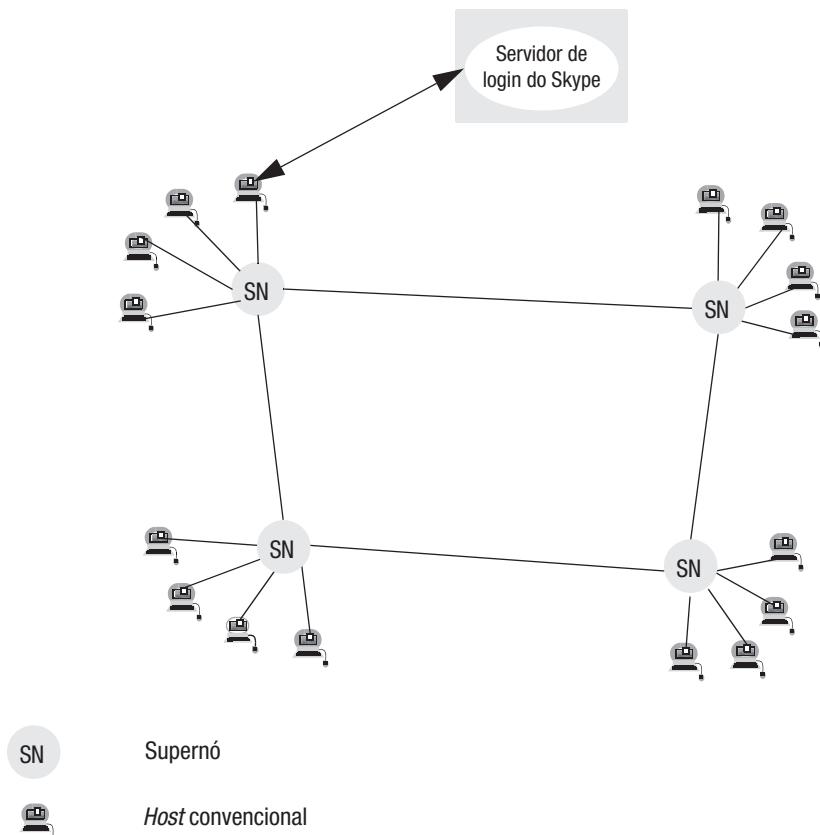


Figura 4.16 Arquitetura de sobreposição do Skype.

pelo Kazaa em 2003 e, assim, compartilha muitas das características do aplicativo de compartilhamento de arquivos *peer-to-peer* do Kazaa [Leibowitz *et al.* 2003]. Ele é amplamente distribuído, com uma estimativa de 370 milhões de usuários no início de 2009.

O Skype é um excelente estudo de caso do uso de redes de sobreposição em sistemas reais (e de grande escala), indicando como funcionalidade avançada pode ser fornecida de maneira específica do aplicativo e sem modificar a arquitetura básica da Internet. **O Skype é uma rede virtual no sentido de que estabelece conexões entre pessoas (assinantes do Skype correntemente ativos). Nenhum endereço IP ou porta é necessário para estabelecer uma chamada.** A arquitetura da rede virtual que dá suporte ao Skype não é amplamente publicada, mas pesquisadores estudaram o Skype usando diversos métodos, incluindo análise de tráfego, sendo que agora seus princípios são de domínio público. Muitos dos detalhes da descrição a seguir foram extraídos do artigo de Baset e Schulzrinne [2006], o qual contém um estudo pormenorizado do comportamento do Skype.

Arquitetura do Skype • O Skype é baseado em uma infraestrutura *peer-to-peer* que consiste em máquinas normais de usuário (referidas como *hosts*) e supernós – os quais são *hosts* normais do Skype que têm recursos suficientes para cumprir sua função avançada. Os supernós são selecionados de acordo com a demanda, com base em diversos critérios, incluindo a largura de banda disponível, a acessibilidade (a máquina deve ter um endereço IP global e não estar oculta por um roteador com NAT habilitado, por exemplo) e também a disponibilidade (com base no período de tempo em que o Skype esteve funcionando continuamente nesse nó). Essa estrutura global está ilustrada na Figura 4.16.

Conexão de usuário • Os usuários do Skype são autenticados por meio de um servidor de *login* conhecido. Então, eles entram em contato com um supernó selecionado. Para se conseguir isso, cada cliente mantém uma cache de identidades de supernó (isto é, pares endereço IP e número de porta). No primeiro *login*, essa cache é preenchida com os endereços de cerca de sete supernós e, com o passar do tempo, o cliente constrói e mantém um conjunto muito maior (talvez várias centenas).

Busca de usuários • O principal objetivo dos supernós é fazer a pesquisa eficiente de índices globais de usuários, os quais são distribuídos pelos supernós. A pesquisa é orchestrada pelo supernó escolhido do cliente e envolve expandir a busca para outros supernós até que o usuário seja encontrado. Em média, oito supernós são contactados. Uma busca de usuário normalmente leva de três a quatro segundos para *hosts* que têm um endereço IP global (e um pouco mais – de cinco a seis segundos – se estiver atrás de um roteador com NAT habilitado). De acordo com as experiências, parece que os nós intermediários envolvidos na busca colocam os resultados na cache para melhorar o desempenho.

Conexão de voz • Uma vez encontrado o usuário desejado, o Skype estabelece uma conexão de voz entre as duas partes, usando TCP para sinalizar pedidos e términos de chamada e UDP ou TCP para o *streaming* de áudio. UDP é preferido, mas TCP, junto ao uso de um nó intermediário, é usado em determinadas circunstâncias para contornar *firewalls* (consulte Baset e Schulzrinne [2006] para detalhes). O *software* usado para codificar e decodificar áudio desempenha um papel chave no fornecimento da chamada de excelente qualidade normalmente obtida no Skype, e os algoritmos associados são cuidadosamente personalizados para operar no ambiente da Internet em 32 kbit/s e acima.

4.6 Estudo de caso: MPI

A passagem de mensagens foi apresentada na Seção 4.2.1, que esboça os princípios básicos da troca de mensagens entre dois processos usando operações *send* e *receive*. A variante síncrona da passagem de mensagens é conseguida com o bloqueio de chamadas de *send* e *receive*, enquanto a variante assíncrona exige uma forma de *send* sem bloqueio. O resultado final é um paradigma de programação distribuída leve, eficiente e mínima de muitas maneiras.

Esse estilo de programação distribuída é atraente nos tipos de sistema em que o desempenho é fundamental, mais notadamente na computação de alto desempenho. Nesta seção, apresentamos um estudo de caso do padrão Message Passing Interface, desenvolvido pela comunidade de computação de alto desempenho. O padrão MPI apareceu pela primeira vez em 1994 no MPI Forum [www mpi-forum.org], como uma reação contra a ampla variedade de estratégias patenteadas que estavam sendo usadas para passagem de

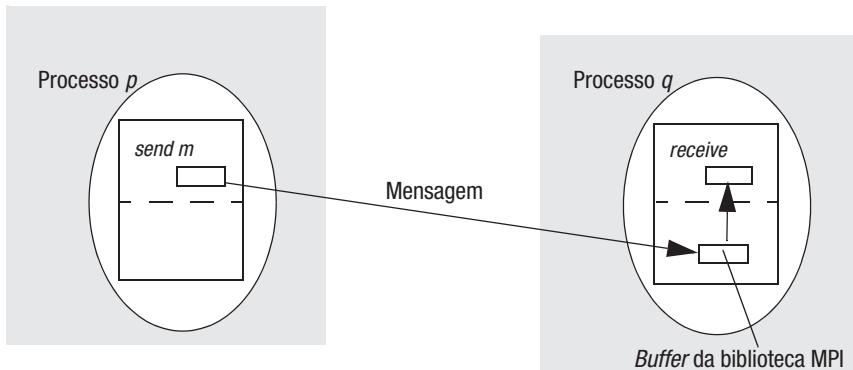


Figura 4.17 Uma visão geral da comunicação ponto a ponto no padrão MPI.

mensagens nesse setor. O padrão também teve forte influência na área de grades computacionais (*grid*), discutida no Capítulo 9 – por exemplo, por meio do desenvolvimento do GridMPI [www.gridmpi.org]. O objetivo do MPI Forum era manter a simplicidade, a praticabilidade e a eficiência inerentes da estratégia da passagem de mensagens, mas melhorar isso com portabilidade, apresentando uma interface padronizada, independente do sistema operacional ou da interface de soquete específica da linguagem de programação. O padrão MPI também foi projetado de forma a ser flexível, e o resultado é uma especificação de passagem de mensagens abrangente em todas as suas variantes (com mais de 115 operações). Os aplicativos usam a interface MPI por intermédio de uma biblioteca de passagem de mensagens – disponível para diversos sistemas operacionais e linguagens de programação, incluindo C++ e Fortran.

O modelo arquitetônico subjacente do padrão MPI é relativamente simples e aparece na Figura 4.17. Ele é semelhante ao modelo apresentado na Seção 4.2.1, mas com a capacidade acrescentada de ter *buffers* da biblioteca MPI explicitamente no remetente e no destinatário, gerenciados pela biblioteca MPI e utilizados para manter dados em trânsito. Note que essa figura mostra um único caminho do remetente para o destinatário por intermédio do *buffer* da biblioteca MPI do destinatário (outras opções, por exemplo usando o *buffer* da biblioteca MPI do remetente, aparecerão a seguir).

Para termos uma ideia dessa complexidade, vamos examinar diversas variantes da operação *send* resumidas na Figura 4.18. Isso é um refinamento da visão da passagem de mensagens apresentada na Seção 4.2.1, oferecendo mais escolhas e controle e realmente separando semântica da passagem de mensagens síncrona/assíncrona e com bloqueio/sem bloqueio.

Vamos começar examinando as quatro operações de bloqueio apresentadas na coluna associada da Figura 4.18. O segredo para entender esse conjunto de operações é compreender que bloqueio é interpretado como “bloqueado até que seja seguro retornar”, no sentido de que dados do aplicativo foram copiados no ambiente MPI e, assim, estão em trânsito ou foram entregues e, portanto, o *buffer* do aplicativo pode ser reutilizado (por exemplo, para a próxima operação *send*). Isso possibilita várias interpretações do que significa “ser seguro para retornar”. A operação *MPI_Send* é genérica e simplesmente exige que esse nível de segurança seja fornecido (na prática, isso é frequentemente implementado usando-se *MPI_Ssend*). *MPI_Ssend* é exatamente igual à passagem de mensagens síncrona (e com bloqueio), conforme apresentado na Seção 4.2.1, com segurança

<i>Operações send</i>	<i>Com bloqueio</i>	<i>Sem bloqueio</i>
<i>Genéricas</i>	<i>MPI_Send</i> : o remetente bloqueia até que seja seguro retornar; isto é, até que a mensagem esteja em trânsito ou tenha sido entregue e que, portanto, o <i>buffer</i> de aplicativo do remetente possa ser reutilizado.	<i>MPI_Isend</i> : a chamada retorna imediatamente e o programador recebe um identificador de pedido de comunicação, o qual, então, pode ser usado para verificar o andamento da chamada via <i>MPI_Wait</i> ou <i>MPI_Test</i> .
<i>Síncronas</i>	<i>MPI_Ssend</i> : o remetente e o destinatário são sincronizados e a chamada só retornará quando a mensagem tiver sido entregue no endereço do destinatário.	<i>MPI_Issend</i> : semelhante a <i>MPI_Isend</i> , mas com <i>MPI_Wait</i> e <i>MPI_Test</i> indicando se a mensagem foi entregue no endereço do destinatário.
<i>Com buffer</i>	<i>MPI_Bsend</i> : o remetente aloca explicitamente um <i>buffer</i> da biblioteca MPI (usando uma chamada de <i>MPI_Buffer_attach</i> separada), e a chamada retorna quando os dados são copiados com sucesso nesse <i>buffer</i> .	<i>MPI_Ibsend</i> : semelhante a <i>MPI_Isend</i> , mas com <i>MPI_Wait</i> e <i>MPI_Test</i> indicando se a mensagem foi copiada no <i>buffer</i> MPI do remetente e, portanto, está em trânsito.
<i>Prontas</i>	<i>MPI_Rsend</i> : a chamada retorna quando o <i>buffer</i> de aplicativo do remetente pode ser reutilizado (como com <i>MPI_Send</i>), mas o programador também está indicando para a biblioteca que o destinatário está pronto para receber a mensagem, resultando em uma possível otimização da implementação subjacente.	<i>MPI_Irsend</i> : o efeito é igual ao de <i>MPI_Isend</i> , mas com <i>MPI_Rsend</i> o programador está indicando para a implementação subjacente que o destinatário está realmente pronto para receber (resultando nas mesmas otimizações).

Figura 4.18 Operações *send* selecionadas no padrão MPI.

interpretada como entregue, enquanto *MPI_Bsend* tem semântica mais fraca, no sentido de que a mensagem foi copiada para o *buffer* da biblioteca MPI previamente alocado e ainda está em trânsito. *MPI_Rsend* é uma operação bastante curiosa, na qual o programador especifica que sabe que o destinatário está pronto para receber a mensagem. Se isso é conhecido, a implementação subjacente pode ser otimizada, pois não há necessidade de verificar se existe um *buffer* disponível para receber a mensagem, evitando um *handshake*. Claramente, essa é uma operação muito perigosa, que vai falhar se a suposição a respeito de estar pronto for inválida. Na figura é possível observar a elegante simetria das operações *send* sem bloqueio, desta vez definidas na semântica das operações associadas *MPI_Wait* e *MPI_Test* (note também a convenção de atribuição de nomes coerente em todas as operações).

O padrão também suporta operações *receive* com e sem bloqueio (*MPI_recv* e *MPI_Irecv*, respectivamente), e as variantes de *send* e *receive* podem formar pares em qualquer combinação, oferecendo ao programador um rico controle sobre a semântica da passagem de mensagens. Além disso, o padrão define um rico conjunto de primitivas de comunicação entre vários processos (referido como comunicação coletiva), incluindo, por exemplo, as operações *scatter* (um para muitos) e *gather* (muitos para um).

4.7 Resumo

A primeira seção deste capítulo mostrou que os protocolos de transmissão da Internet fornecem dois blocos básicos a partir dos quais os protocolos em nível de aplicativos podem ser construídos. Há um compromisso interessante entre os dois protocolos: o protocolo UDP fornece um recurso simples de passagem de mensagem, que sofre de falhas por omissão, mas não tem penalidades de desempenho incorporadas. Por outro lado, em boas condições, o protocolo TCP garante a entrega das mensagens, mas à custa de mensagens adicionais e com latência e custos de armazenamento mais altos.

A segunda seção mostrou três estilos alternativos de empacotamento. O CORBA e seus predecessores optam por empacotar os dados para uso pelos destinatários que têm conhecimento anterior dos tipos de seus componentes. Em contraste, quando a linguagem Java serializa dados, ela inclui informações completas sobre os tipos de seu conteúdo, permitindo ao destinatário reconstruí-los puramente a partir do conteúdo. A linguagem XML, assim como a Java, inclui informações completas sobre os tipos de dados. Outra grande diferença é que o CORBA exige uma especificação dos tipos dos dados a serem empacotados (no IDL) para gerar os métodos de empacotamento e desempacotamento, enquanto a linguagem Java usa reflexão para serializar e desserializar objetos. Uma variedade de meios é usada para gerar código XML, dependendo do contexto. Por exemplo, muitas linguagens de programação, incluindo a Java, fornecem processadores para fazer a transformação entre objetos em nível de XML e de linguagem.

Mensagens *multicast* são usadas na comunicação entre os membros de um grupo de processos. O *multicast* IP é disponibilizado tanto para redes locais como para a Internet e tem a mesma semântica de falhas que os datagramas UDP; porém, apesar de sofrer de falhas por omissão, é uma ferramenta útil para muitas aplicações *multicast*. Algumas aplicações têm requisitos mais restritos – em particular, o de que a distribuição *multicast* deva ser atômica; isto é, ela deve ter uma distribuição do tipo tudo ou nada. Outros requisitos do *multicast* estão relacionados ao ordenamento das mensagens, o mais exigente, que impõe que todos os membros de um grupo devem receber todas as mensagens na mesma ordem.

Multicast também pode ser suportado pelas redes de sobreposição nos casos em que, por exemplo, *multicast* IP não é suportado. Mais geralmente, as redes de sobreposição oferecem um serviço de virtualização da arquitetura da rede, permitindo que serviços especializados sejam criados sobre a infraestrutura de interligação em rede subjacente, por exemplo, UDP ou TCP. As redes de sobreposição resolvem parcialmente os problemas associados ao princípio fim-a-fim de Saltzer, permitindo a geração de abstrações de rede mais específicas para o aplicativo.

O capítulo terminou com um estudo de caso da especificação MPI desenvolvida pela comunidade de computação de alto desempenho e que apresenta suporte flexível para a passagem de mensagens, junto a suporte adicional para passagem de mensagens por comunicação coletiva.

Exercícios

- | | |
|--|--|
| 4.1 É conceitivelmente útil que uma porta tenha vários receptores?
4.2 Um servidor cria uma porta que ele utiliza para receber pedidos dos clientes. Discuta os problemas de projeto relativos ao relacionamento entre o nome dessa porta e os nomes usados pelos clientes. | <i>páginas 148</i>
<i>páginas 148</i> |
|--|--|

- 4.3 Os programas das Figuras 4.3 e 4.4 estão disponíveis no endereço [www.cdk5.net/ipc] (em inglês). Utilize-os para fazer uma série de testes para determinar as condições nas quais os datagramas, às vezes, são descartados. Dica: o programa cliente deve ser capaz de variar o número de mensagens enviadas e seus tamanhos; o servidor deve detectar quando uma mensagem de um cliente específico é perdida. *páginas 150*
- 4.4 Use o programa da Figura 4.3 para fazer um programa cliente que leia repetidamente uma linha de entrada do usuário, a envie para o servidor em uma mensagem datagrama UDP e receba uma mensagem do servidor. O cliente estabelece um tempo limite em seu soquete para que possa informar o usuário quando o servidor não responder. Teste este programa cliente com o servidor da Figura 4.4. *páginas 150*
- 4.5 Os programas das Figuras 4.5 e 4.6 estão disponíveis no endereço [www.cdk5.net/ipc] (em inglês). Modifique-os de modo que o cliente leia repetidamente uma linha de entrada do usuário e a escreva no fluxo. O servidor deve ler repetidamente o fluxo, imprimindo o resultado de cada leitura. Faça uma comparação entre o envio de dados em mensagens de datagrama UDP e por meio de um fluxo. *página 153*
- 4.6 Use os programas desenvolvidos no Exercício 4.5 para testar o efeito sobre o remetente quando o receptor falha e vice-versa. *página 153*
- 4.7 O XDR da Sun empacota dados convertendo-os para uma forma *big-endian* antes de transmiti-los. Discuta as vantagens e desvantagens desse método em comparação com o CDR do CORBA. *página 160*
- 4.8 O XDR da Sun alinha cada valor primitivo em um limite de quatro bytes, enquanto o CDR da CORBA alinha um valor primitivo de tamanho *n* em um limite de *n* bytes. Discuta os compromissos na escolha dos tamanhos ocupados pelos valores primitivos. *página 160*
- 4.9 Por que não há nenhuma tipagem de dados explícita no CDR do CORBA? *página 160*
- 4.10 Escreva um algoritmo, em pseudocódigo, para descrever o procedimento de serialização mostrado na Seção 4.3.2. O algoritmo deve mostrar quando identificadores são definidos ou substituídos por classes e instâncias. Descreva a forma serializada que seu algoritmo produziria para uma instância da seguinte classe *Couple*.

```
class Couple implements Serializable{  
    private Person one;  
    private Person two;  
    public Couple(Person a, Person b) {  
        one = a;  
        two = b;  
    }  
}
```

página 162

- 4.11 Escreva um algoritmo, em pseudocódigo, para descrever a desserialização da forma serializada produzida pelo algoritmo definido no Exercício 4.10. Dica: use reflexão para criar uma classe a partir de seu nome, um construtor a partir de seus tipos de parâmetro e uma nova instância de um objeto a partir do construtor e dos valores de argumento. *página 162*
- 4.12 Por que dados binários não podem ser representados diretamente em XML, por exemplo, como valores em Unicode? Os elementos XML podem transportar *strings* representados como *base64*. Discuta as vantagens ou desvantagens de usar esse método para representar dados binários. *página 164*

- 4.13 Defina uma classe cujas instâncias representem referências de objeto remoto. Ela deve conter informações semelhantes àquelas mostradas na Figura 4.13 e deve fornecer métodos de acesso necessários para os protocolos de nível mais alto (consulte requisição-resposta, no Capítulo 5, para ver um exemplo). Explique como cada um dos métodos de acesso será usado por esse protocolo. Dê uma justificativa para o tipo escolhido para a variável de instância que contém informações sobre a interface do objeto remoto. *página 168*
- 4.14 O *multicast* IP fornece um serviço que sofre de falhas por omissão. Faça uma série de testes, baseada no programa da Figura 4.14, para descobrir as condições sob as quais uma mensagem *multicast* às vezes é eliminada por um dos membros do grupo *multicast*. O kit de testes deve ser projetado de forma a permitir a existência de vários processos remetentes. *página 170*
- 4.15 Esboce o projeto de um esquema que utilize retransmissões de mensagem *multicast* IP para superar o problema das mensagens descartadas. Seu esquema deve levar em conta os seguintes pontos:
- i) podem existir vários remetentes;
 - ii) geralmente apenas uma pequena parte das mensagens é descartada;
 - iii) os destinatários podem não enviar uma mensagem necessariamente dentro de um limite de tempo em particular.
- Suponha que as mensagens que não são descartadas chegam na ordem do remetente. *página 173*
- 4.16 Sua solução para o Exercício 4.15 deve ter superado o problema das mensagens descartadas no *multicast* IP. Em que sentido sua solução difere da definição de *multicast* confiável? *página 173*
- 4.17 Imagine um cenário no qual mensagens *multicast* enviadas por diferentes clientes são entregues em ordens diferentes a dois membros do grupo. Suponha que esteja em uso alguma forma de retransmissões de mensagem, mas que as mensagens que não são descartadas cheguem na ordem do remetente. Sugira o modo como os destinos poderiam remediar essa situação. *página 173*
- 4.18 Reveja a arquitetura da Internet apresentada no Capítulo 3 (consulte as Figuras 3.12 e 3.14). Que impacto a introdução das redes de sobreposição tem sobre essa arquitetura, em particular sobre a visão conceitual do programador em relação à Internet? *página 175*
- 4.19 Quais são os principais argumentos para a adoção de uma estratégia de supernós no Skype? *página 177*
- 4.20 Conforme discutido na Seção 4.6, o padrão MPI oferece diversas variantes da operação *send*, incluindo *MPI_Rsend*, que presume que o destinatário está pronto para receber no momento do envio. Quais otimizações são possíveis na implementação se essa suposição estiver correta e quais são as repercussões no caso de ser falsa? *página 180*

5

Invocação Remota

- 5.1 Introdução
- 5.2 Protocolos de requisição-resposta
- 5.3 Chamada de procedimento remoto
- 5.4 Invocação a método remoto
- 5.5 Estudo de caso: RMI Java
- 5.6 Resumo

Este capítulo examina os paradigmas de invocação remota apresentados no Capítulo 2 (as técnicas de comunicação indireta serão tratadas no Capítulo 6). Começaremos examinando o serviço mais primitivo, a comunicação por requisição-resposta, que representa aprimoramentos relativamente pequenos nas primitivas de comunicação entre processos, discutidas no Capítulo 4. Em seguida, o capítulo examina as duas técnicas de invocação remota mais importantes para comunicação em sistemas distribuídos:

- A estratégia de chamada de procedimento remoto (RPC) estende a abstração de programação comum da chamada de procedimento para os ambientes distribuídos, permitindo que um processo chame um procedimento em um nó remoto como se fosse local.
- A invocação a método remoto (RMI) é semelhante à RPC, mas para objetos distribuídos, com vantagens adicionais em usar conceitos de programação orientada a objetos em sistemas distribuídos e em estender o conceito de referência de objeto para o ambiente distribuído global e permitir o uso de referências de objeto como parâmetros em invocações remotas.

O capítulo também apresenta a RMI Java como um estudo de caso da estratégia de invocação a método remoto (uma maior compreensão pode ser obtida no Capítulo 8, no qual examinaremos o CORBA).

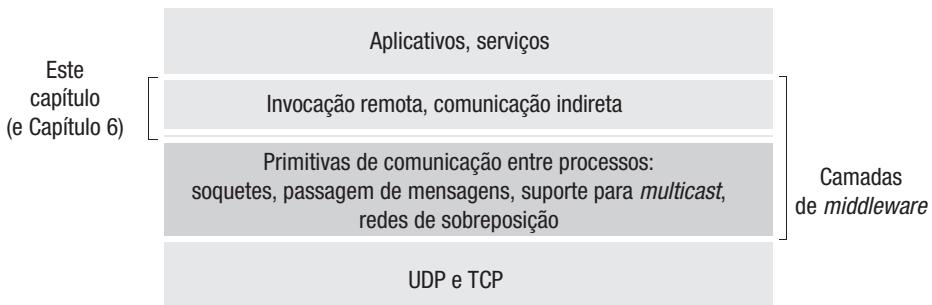


Figura 5.1 Camadas de *middleware*.

5.1 Introdução

Este capítulo trata de como os processos (ou entidades, em um nível de abstração mais alto, como objetos ou serviços) se comunicam em um sistema distribuído, examinando, em particular, os paradigmas de invocação remota definidos no Capítulo 2:

- Os *protocolos de requisição-resposta* representam um padrão sobre a passagem de mensagens e suportam a troca bilateral de mensagens, como a encontrada na computação cliente-servidor. Em particular, tais protocolos fornecem suporte de nível relativamente baixo para solicitar a execução de uma operação remota e suporte direto para RPC e RMI, discutidas a seguir.
- O mais antigo, e talvez mais conhecido exemplo de modelo mais amigável para o programador foi a extensão do modelo de chamada de procedimentos convencional para os sistemas distribuídos, a *chamada de procedimento remoto* (*RPC*, *Remote Procedure Call*), que permite aos programas clientes chamarem procedimentos de forma transparente em programas servidores que estejam sendo executados em processos separados e, geralmente, em computadores diferentes do cliente.
- Nos anos 90, o modelo da programação baseada em objetos foi ampliado para permitir que objetos de diferentes processos se comunicassem por intermédio da *invocação a método remoto* (*RMI*, *Remote Method Invocation*). A RMI é uma extensão da invocação a método local que permite a um objeto que está em um processo invocar os métodos de um objeto que está em outro processo.

Note que usamos o termo “RMI” para nos referirmos à invocação a método remoto de uma maneira genérica – isso não deve ser confundido com exemplos particulares de invocação a método remoto, como a RMI Java.

Retornando ao diagrama apresentado pela primeira vez no Capítulo 4 (e reproduzido na Figura 5.1), este capítulo, junto ao Capítulo 6, continua nosso estudo dos conceitos de *middleware*, abordando a camada acima da comunicação entre processos. Em particular, as Seções 5.2 até 5.4 enfocam os estilos de comunicação listados anteriormente, e a Seção 5.5 apresenta um estudo de caso mais complexo, a RMI Java.

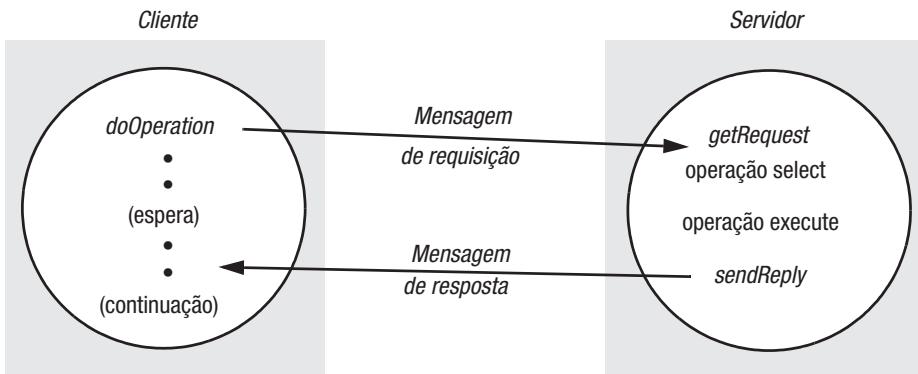


Figura 5.2 Comunicação por requisição-resposta.

5.2 Protocolos de requisição-resposta

Esta forma de comunicação é projetada para suportar as funções e as trocas de mensagens em interações cliente e servidor típicas. No caso normal, a comunicação por requisição-resposta é síncrona, pois o processo cliente é bloqueado até que a resposta do servidor chegue. Ela também pode ser confiável, pois a resposta do servidor é efetivamente um confirmação para o cliente. A comunicação por requisição-resposta assíncrona é uma alternativa útil em situações em que os clientes podem recuperar as respostas posteriormente – veja a Seção 7.5.2.

As trocas entre cliente e servidor estão descritas nos parágrafos a seguir, em termos das operações *send* e *receive* na API Java para datagramas UDP, embora muitas implementações atuais usem TCP. Um protocolo construído sobre datagramas evita sobreargas desnecessárias associadas ao protocolo TCP. Em particular:

- As confirmações são redundantes, pois as requisições são seguidas por respostas.
- O estabelecimento de uma conexão envolve dois pares extras de mensagens, além do par exigido por uma requisição e uma resposta.
- O controle de fluxo é redundante para a maioria das invocações, que passam apenas pequenos argumentos e resultados.

O protocolo de requisição-resposta • O protocolo que descrevemos aqui é baseado em um trio de primitivas de comunicação: `doOperation`, `getRequest` e `sendReply`, como mostrado na Figura 5.2. Esse protocolo de requisição-resposta combina pedidos com respostas. Ele pode ser projetado para fornecer certas garantias de entrega. Se forem usados datagramas UDP, as garantias de entrega deverão ser fornecidas pelo protocolo de requisição-resposta, o qual pode usar a mensagem de resposta do servidor como confirmação da mensagem de requisição do cliente. A Figura 5.3 descreve, em linhas gerais, as três primitivas de comunicação.

O método `doOperation` é usado pelos clientes para invocar operações remotas. Seus argumentos especificam o servidor remoto e a operação a ser invocada, junto às informações adicionais (argumentos) exigidas pela operação. Seu resultado é um vetor de bytes contendo a resposta. Presume-se que o cliente que chama `doOperation` empacota os argumentos em um vetor de bytes e desempacota os resultados do vetor de bytes retornado. O primeiro argumento de `doOperation` é uma instância da classe `RemoteRef`, que representa

```
public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)
    Envia uma mensagem de requisição para o servidor remoto e retorna a resposta.
    Os argumentos especificam o servidor remoto, a operação a ser invocada e
    os argumentos dessa operação.

public byte[] getRequest ();
    Lê uma requisição do cliente por meio da porta do servidor.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
    Envia a mensagem de resposta reply para o cliente como seu endereço de Internet e porta.
```

Figura 5.3 Operações do protocolo de requisição-resposta.

referências para servidores remotos. Essa classe fornece métodos para obter o endereço de Internet e a porta do servidor associado. O método *doOperation* envia uma mensagem de requisição para o servidor, cujo endereço de Internet e porta são especificados na referência remota dada como argumento. Após enviar a mensagem de requisição, *doOperation* invoca *receive* para obter uma mensagem de resposta, a partir da qual extrai o resultado e o retorna para o chamador. O processo (cliente) que executa a operação *doOperation* é bloqueado até que o servidor execute a operação solicitada e transmita uma mensagem de resposta para ele.

getRequest é usado por um processo servidor para obter requisições de serviço, como mostrado na Figura 5.3. Quando o servidor tiver invocado a operação especificada, ele usa *sendReply* para enviar a mensagem de resposta ao cliente. Quando a mensagem de resposta é recebida pelo cliente, a operação *doOperation* original é desbloqueada, e a execução do programa cliente continua.

As informações a serem transmitidas em uma mensagem de requisição ou em uma mensagem de resposta aparecem na Figura 5.4. O primeiro campo indica se o tipo de mensagem (*messageType*) é *Request* ou *Reply*. O segundo campo, *requestId*, contém um identificador de mensagem. Um *doOperation* no cliente gera um *requestId* para cada mensagem de requisição, e o servidor copia esses identificadores nas mensagens de resposta correspondentes. Isso permite que *doOperation* verifique se uma mensagem de resposta é o resultado da requisição atual e não de uma chamada anterior atrasada. O terceiro campo é uma referência remota (*remoteReference*). O quarto campo é um identificador da operação (*OperationId*) a ser invocada. Por exemplo, as operações de uma interface poderiam ser numeradas como 1, 2, 3,...; se o cliente e o servidor usam uma linguagem comum que suporta reflexão, uma representação da operação em si pode ser colocada nesse campo.

messageType	<i>int (0=Request, 1=Reply)</i>
requestId	<i>int</i>
remoteReference	<i>RemoteRef</i>
OperationId	<i>int ou operação</i>
arguments	<i>// vetor de bytes</i>

Figura 5.4 Estrutura da mensagem de requisição-resposta.

Identificadores de mensagem • Qualquer esquema que envolva o gerenciamento de mensagens para fornecer propriedades adicionais, como entrega de mensagens confiável ou comunicação por requisição-resposta, exige que cada mensagem tenha um identificador exclusivo por meio do qual possa ser referenciada. Um identificador de mensagem consiste em duas partes:

1. um *requestId*, que é extraído pelo processo remetente a partir de uma sequência crescente de valores inteiros;
2. um identificador do processo remetente; por exemplo, sua porta e endereço de Internet.

A primeira parte torna o identificador exclusivo para o remetente e a segunda o torna exclusivo no sistema distribuído. (A segunda parte pode ser obtida independentemente – por exemplo, se UDP estiver em uso, a partir da mensagem recebida.)

Quando o valor de *requestId* atinge o máximo para um inteiro sem sinal (por exemplo, $2^{32} - 1$), ele volta a zero. A única restrição aqui é que o tempo de vida de um identificador de mensagem deve ser bem menor do que o tempo que leva para esgotar os valores na sequência de inteiros.

Modelo de falhas do protocolo de requisição-resposta • Se as três primitivas *doOperation*, *getRequest* e *sendReply* forem implementadas sobre datagramas UDP, elas sofrerão das mesmas falhas de comunicação. Ou seja:

- Sofrerão de falhas por omissão.
- Não há garantias de entrega das mensagens na ordem do envio.

Além disso, o protocolo pode sofrer uma falha de processos (veja a Seção 2.4.2). Supomos que os processos têm falhas de colapso, isto é, quando param, permanecem parados – e não produzem comportamento bizantino.

Para levar em conta as ocasiões em que um servidor falhou ou que uma mensagem de requisição ou resposta é perdida, *doOperation* usa um tempo limite (*timeout*) quando está esperando receber a mensagem de resposta do servidor. A ação executada quando ocorre um tempo limite depende das garantias de entrega oferecidas.

Tempos limite (timeouts) • Existem várias opções para o que *doOperation* deva fazer após esgotar um tempo limite (*timeout*). A opção mais simples é retornar imediatamente de *doOperation*, com uma indicação para o cliente de que *doOperation* falhou. Essa não é a estratégia mais comum – o tempo limite pode ter esgotado devido à perda da mensagem de requisição ou de resposta e, neste último caso, a operação foi executada. Para levar em conta a possibilidade de mensagens perdidas, *doOperation* envia a mensagem de requisição repetidamente, até receber uma resposta ou estar razoavelmente seguro de que o atraso se deve à falta de resposta do servidor e não à perda de mensagens. Finalmente, quando *doOperation* retornar, indicará isso para o cliente por meio de uma exceção, dizendo que nenhum resultado foi recebido.

Descarte de mensagens de requisição duplicadas • Nos casos em que a mensagem de requisição é retransmitida, o servidor pode recebê-la mais de uma vez. Por exemplo, o servidor pode receber a primeira mensagem de requisição, mas demorar mais do que o tempo limite do cliente para executar o comando e retornar a resposta. Isso pode levar o servidor a executar uma operação mais de uma vez para a mesma requisição. Para evitar isso, o protocolo é projetado de forma a reconhecer mensagens sucessivas (do mesmo cliente) com o mesmo identificador de requisição e eliminar as duplicatas. Se o servidor ainda não enviou a resposta, não precisa executar nenhuma ação especial – ele transmitirá a resposta quando tiver terminado de executar a operação.

Mensagens de resposta perdidas • Se o servidor já tiver enviado a resposta quando receber uma requisição duplicada, precisará executar a operação novamente para obter o resultado, a não ser que tenha armazenado o resultado da execução original. Alguns servidores podem executar suas operações mais de uma vez e obter os mesmos resultados. Uma *operação idempotente* é aquela que pode ser efetuada repetidamente com o mesmo efeito, como se tivesse sido executada exatamente uma vez. Por exemplo, uma operação para pôr um elemento em um conjunto é idempotente, pois sempre terá o mesmo efeito no conjunto, toda vez que for executada, enquanto uma operação para incluir um elemento em uma sequência não é idempotente, pois amplia a sequência sempre que é executada. Um servidor cujas operações são todas idempotentes não precisa adotar medidas especiais para evitar suas execuções mais de uma vez.

Histórico • Para os servidores que exigem retransmissão das respostas sem executar novamente as operações, pode-se usar um histórico. O termo *histórico* é usado para se referir a uma estrutura que contém um registro das mensagens (respostas) que foram transmitidas. Uma entrada em um histórico contém um identificador de requisição, uma mensagem e um identificador do cliente para o qual ela foi enviada. Seu objetivo é permitir que o servidor retransmita as mensagens de resposta quando os processos clientes as solicitarem. Um problema associado ao uso de um histórico é seu consumo de memória. Um histórico pode se tornar muito grande, a menos que o servidor possa identificar quando não há mais necessidade de retransmissão das mensagens.

Como os clientes só podem fazer uma requisição por vez, o servidor pode interpretar cada nova requisição como uma confirmação de sua resposta anterior. Portanto, o histórico precisa conter apenas a última mensagem de resposta enviada a cada cliente. Entretanto, o volume de mensagens de resposta no histórico de um servidor pode ser um problema quando ele tiver um grande número de clientes. Isso é combinado com o fato de que, quando um processo cliente termina, ele não confirma a última resposta recebida – portanto, as mensagens no histórico normalmente são descartadas após determinado período de tempo.

Estilos de protocolos de troca • Três protocolos, que produzem diferentes comportamentos na presença de falhas de comunicação, são usados para implementar vários tipos de comportamento de requisição. Eles foram identificados originalmente por Spector [1982]:

- o protocolo *request* (*R*);
- o protocolo *request-reply* (*RR*);
- o protocolo *request-reply-acknowledge reply* (*RRA*).

As mensagens passadas nesses protocolos estão resumidas na Figura 5.5. No protocolo R, uma única mensagem *Request* é enviada pelo cliente para o servidor. O protocolo R pode ser usado quando não existe nenhum valor a ser retornado do método remoto e o cliente não exige confirmação de que a operação foi executada. O cliente pode prosseguir imediatamente após a mensagem de requisição ser enviada, pois não há necessidade de esperar por uma mensagem de resposta. Esse protocolo é implementado sobre datagramas UDP e, portanto, sofre das mesmas falhas de comunicação.

O protocolo RR é útil para a maioria das trocas cliente-servidor, pois é baseado no protocolo de requisição-resposta. Não são exigidas mensagens de confirmação especiais, pois uma mensagem de resposta (*reply*) do servidor é considerada como confirmação do recebimento da mensagem de requisição (*request*) do cliente. Analogamente, uma chamada subsequente de um cliente pode ser considerada como uma confirmação da mensagem de resposta de um servidor. Conforme vimos anteriormente, as falhas de

Nome	Mensagens enviadas pelo		
	Cliente	Servidor	Cliente
R	<i>Requisição</i>		
RR	<i>Requisição</i>	<i>Resposta</i>	
RRA	<i>Requisição</i>	<i>Resposta</i>	<i>Resposta de confirmação</i>

Figura 5.5 Protocolos RPC.

comunicação ocasionadas pela perda de datagramas UDP podem ser mascaradas pela retransmissão das requisições com filtragem duplicada e pelo registro das respostas em um histórico para retransmissões.

O protocolo RRA é baseado na troca de três mensagens: requisição, resposta e confirmação. A mensagem de *confirmação* contém o *requestId* da mensagem de resposta que está sendo confirmada. Isso permitirá que o servidor descarte entradas de seu histórico. A chegada de um *requestId* em uma mensagem de confirmação será interpretada como a acusação do recebimento de todas as mensagens de resposta com valores de *requestId* menores; portanto, a perda de uma mensagem de confirmação não é muito prejudicial ao sistema. Embora o protocolo RRA envolva uma mensagem adicional, ela não precisa bloquear o cliente, pois a confirmação pode ser transmitida após a resposta ter sido entregue ao cliente. Contudo, ela utiliza recursos de processamento e rede. O Exercício 5.10 sugere uma otimização para o protocolo RRA.

Uso de TCP para implementar o protocolo de requisição-resposta • A seção 4.2.3 mencionou que frequentemente é difícil decidir-se sobre um tamanho apropriado para o *buffer* de recebimento de datagramas. No protocolo de requisição-resposta, isso se aplica aos *buffers* usados pelo servidor para receber mensagens de requisição e pelo cliente para receber respostas. O comprimento limitado dos datagramas (normalmente, 8 quilobytes) pode não ser considerado adequado para uso em sistemas de RMI ou RPC transparentes, pois os argumentos ou resultados dos procedimentos podem ser de qualquer tamanho.

O desejo de evitar a implementação de protocolos que tratem de requisições e respostas em múltiplos pacotes é um dos motivos para se escolher a implementação de protocolos de requisição-resposta com TCP, permitindo a transmissão de argumentos e resultados de qualquer tamanho. Em particular, a serialização de objetos Java é um protocolo que permite o envio de argumentos e resultados por meio de fluxos entre cliente e servidor, tornando possível que conjuntos de objetos de qualquer tamanho sejam transmitidos de maneira confiável. Se o protocolo TCP for usado, isso garantirá que as mensagens de requisição e de resposta sejam entregues de modo confiável; portanto, não há necessidade de o protocolo de requisição-resposta tratar da retransmissão de mensagens, filtrar duplicatas ou lidar com históricos. Além disso, o mecanismo de controle de fluxo permite que argumentos e resultados grandes sejam passados, sem se tomar medidas especiais para não sobrecarregar o destinatário. Assim, a escolha do protocolo TCP simplifica a implementação de protocolos de requisição-resposta. Se sucessivas requisições e respostas entre o mesmo par cliente-servidor são enviadas por um mesmo fluxo, a sobrecarga de conexão necessária não se aplica a toda invocação remota. Além disso, a sobrecarga em razão das mensagens de confirmação TCP é reduzida quando uma mensagem de resposta é gerada logo após a mensagem de requisição.

Entretanto, se o aplicativo não exige todos os recursos oferecidos pelo TCP, um protocolo mais eficiente, especialmente personalizado, pode ser implementado sobre

UDP. Por exemplo, o NFS da Sun não exige suporte para mensagens de tamanho ilimitado, pois transmite blocos de arquivo de tamanho fixo entre o cliente e o servidor. Além disso, suas operações são projetadas para serem idempotentes, de modo que não importa se as operações são executadas mais de uma vez para retransmitir mensagens de resposta perdidas, tornando desnecessário manter um histórico.

HTTP: um exemplo de protocolo de requisição-resposta • O Capítulo 1 apresentou o protocolo HTTP (HyperText Transfer Protocol), usado pelos navegadores Web para fazer pedidos para servidores Web e receber suas respostas. Para recapitular, os servidores Web gerenciam recursos implementados de diferentes maneiras:

- como dados – por exemplo, o texto de uma página em HTML, uma imagem ou a classe de um *applet*;
- como um programa – por exemplo, *servlets* [java.sun.com III] ou programas em PHP ou Python, que podem ser executados no servidor Web.

As requisições do cliente especificam um URL que inclui o nome DNS de um servidor Web e um número de porta opcional no servidor Web, assim como o identificador de um recurso nesse servidor.

O protocolo HTTP especifica as mensagens envolvidas em uma troca de requisição-resposta, os métodos, os argumentos, os resultados e as regras para representá-los (empacotá-los) nas mensagens. Ele suporta um conjunto fixo de métodos (*GET*, *PUT*, *POST*, etc.) que são aplicáveis a todos os seus recursos. Ele é diferente dos protocolos descritos anteriormente, em que cada serviço tem seu próprio conjunto de operações. Além de invocar métodos sobre recursos Web, o protocolo permite negociação de conteúdo e autenticação com senha:

Negociação de conteúdo: as requisições dos clientes podem incluir informações sobre qual representação de dados elas podem aceitar (por exemplo, linguagem ou tipo de mídia), permitindo que o servidor escolha a representação mais apropriada para o usuário.

Autenticação: credenciais e desafios (*challenges*) são usados para suportar autenticação com senha. Na primeira tentativa de acessar uma área protegida com senha, a resposta do servidor contém um desafio aplicável ao recurso. O Capítulo 11 explicará os desafios. Quando um cliente recebe um desafio, obriga o usuário a digitar um nome e uma senha e envia as credenciais associadas às requisições subsequentes.

O protocolo HTTP é implementado sobre TCP. Na versão original do protocolo, cada interação cliente-servidor consiste nas seguintes etapas:

- O cliente solicita uma conexão com o servidor na porta HTTP padrão ou em uma porta especificada no URL.
- O cliente envia uma mensagem de requisição para o servidor.
- O servidor envia uma mensagem de resposta para o cliente.
- A conexão é fechada.

Entretanto, estabelecer e fechar uma conexão para cada troca de requisição-resposta é dispendioso, sobrecarregando o servidor e causando o envio de muitas mensagens pela rede. Lembremos que os navegadores geralmente fazem vários pedidos para o mesmo servidor – por exemplo, para obter uma imagem de uma página recentemente fornecida – uma versão posterior do protocolo (HTTP 1.1, veja RFC 2616 [Fielding *et al.* 1999]) usa *conexões persistentes* – conexões que permanecem abertas durante uma sequência de

<i>método</i>	<i>URL ou nome de caminho</i>	<i>versão de HTTP</i>	<i>cabeçalhos</i>	<i>corpo da mensagem</i>
GET	http://www.dcs.qmul.ac.uk/index.html	HTTP/ 1.1		

Figura 5.6 Mensagem de requisição HTTP.

trocas de requisição-resposta entre cliente e servidor. Uma conexão persistente pode ser encerrada a qualquer momento, tanto pelo cliente como pelo servidor, pelo envio de uma indicação para o outro participante. Os servidores encerrará uma conexão persistente quando ela estiver ociosa por determinado período de tempo. É possível que um cliente receba uma mensagem do servidor dizendo que a conexão está encerrada no meio do envio de outra requisição (ou requisições). Neste caso, o navegador enviará novamente as requisições, sem envolvimento do usuário, desde que as operações envolvidas sejam idempotentes. Por exemplo, o método *GET*, descrito a seguir, é idempotente. Onde estão envolvidas operações não idempotentes, o navegador deve consultar o usuário para saber o que fazer em seguida.

As requisições e respostas são empacotadas nas mensagens como *strings* de texto ASCII, mas os recursos podem ser representados como sequências de bytes e podem ser compactados. A utilização de texto na representação externa de dados simplificou o uso de HTTP pelos programadores de aplicativos que trabalham diretamente com o protocolo. Neste contexto, uma representação textual não aumenta muito o comprimento das mensagens.

Os recursos implementados como dados são fornecidos como estruturas do tipo MIME em argumentos e resultados. MIME (Multipurpose Internet Mail Extensions) é um padrão para envio de dados de múltiplas partes, especificado no RFC 2045 [Freed e Borenstein 1996], contendo, por exemplo, texto, imagens e som em mensagens de correio eletrônico. Os dados são prefixados com seu *tipo MIME* para que o destinatário saiba como manipulá-los. Um tipo MIME especifica um tipo e um subtipo, por exemplo, *text/plain*, *text/html*, *image/gif* ou *image/jpeg*. Os clientes também podem especificar os tipos MIME que desejam aceitar.

Métodos HTTP • Cada requisição de cliente especifica o nome de um método a ser aplicado em um recurso no servidor e o URL desse recurso. A resposta fornece o *status* da requisição. As mensagens de requisição-resposta também podem conter dados de recurso, o conteúdo de um formulário ou a saída de um programa executado no servidor Web. Os métodos incluem o seguinte:

GET: solicita o recurso cujo URL é dado como argumento. Se o URL se referir a dados, o servidor Web responderá retornando os dados identificados por esse URL. Se o URL se referir a um programa, então o servidor Web executará o programa e retornará sua saída para o cliente. Argumentos podem ser adicionados no URL; por exemplo, o método *GET* pode ser usado para enviar o conteúdo de um formulário como argumento para um programa. A operação *GET* pode ser condicionada à data em que um recurso foi modificado pela última vez. *GET* também pode ser configurado para obter partes dos dados.

Com *GET*, toda a informação da requisição é fornecida no URL (veja, por exemplo, a *string* de consulta da Seção 1.6).

HEAD: esta requisição é idêntica a *GET*, mas não retorna nenhum dado. Entretanto, retorna todas as informações sobre os dados, como a hora da última modificação, seu tipo ou seu tamanho.

POST: especifica o URL de um recurso (por exemplo, um programa) que pode tratar dos dados fornecidos no corpo do pedido. O processamento executado nos dados depende da função do programa especificado no URL. Esse método é feito para lidar com:

- o fornecimento de um bloco de dados para um processo de manipulação de dados, como um *servlet* – por exemplo, enviando um formulário Web para comprar algo em um *site*;
- o envio de uma mensagem para uma lista de distribuição ou da atualização de detalhes de membros da lista;
- a ampliação de um banco de dados com uma operação *append*.

PUT: solicita que os dados fornecidos na requisição sejam armazenados no URL informado, como uma modificação de um recurso já existente ou como um novo recurso.

DELETE: o servidor exclui o recurso identificado pelo URL fornecido. Nem sempre os servidores permitem essa operação; nesse caso, a resposta indicará a falha.

OPTIONS: o servidor fornece ao cliente uma lista de métodos que podem ser aplicados no URL dado (por exemplo, *GET*, *HEAD*, *PUT*) e seus requisitos especiais.

TRACE: o servidor envia de volta a mensagem de requisição. Usado para propósitos de diagnóstico.

As operações *PUT* e *DELETE* são idempotentes, mas *POST* não necessariamente o é, pois pode alterar o estado de um recurso. As outras são operações *seguras*, pois não alteram nada.

As requisições descritas anteriormente podem ser interceptadas por um servidor *proxy* (veja a Seção 2.3.1). As respostas de *GET* e *HEAD* podem ser armazenadas em cache por servidores *proxy*.

Conteúdo da mensagem • A mensagem de requisição especifica o nome de um método, o URL de um recurso, a versão do protocolo, alguns cabeçalhos e um corpo de mensagem opcional. A Figura 5.6 mostra o conteúdo de uma mensagem de requisição HTTP cujo método é *GET*. Quando o URL especifica um recurso de dados, o método *GET* não tem corpo de mensagem.

As requisições para servidores *proxies* precisam do URL absoluto, como mostrado na Figura 5.6. As requisições para os servidores de origem (aqueles onde fica o recurso) especificam um nome de caminho e fornecem o nome DNS do servidor no campo de cabeçalho *Host*. Por exemplo,

```
GET /index.html HTTP/1.1
Host: www.dcs.qmul.ac.uk
```

Em geral, os campos de cabeçalho contêm modificadores de requisição e informações do cliente, como as condições do recurso na data de modificação mais recente ou os tipos de conteúdo aceitáveis (por exemplo, texto HTML, áudio ou imagens JPEG). Um campo de autorização pode ser usado para fornecer as credenciais do cliente, na forma de um certificado, definindo seus direitos de acesso a um recurso.

Uma mensagem de resposta possui a versão do protocolo, um código de *status* e um “motivo”, alguns cabeçalhos e um corpo de mensagem opcional, como mostrado na Figura 5.7. O código de *status* e o motivo fornecem um relato sobre o sucesso ou não na execução da requisição: o primeiro é um valor inteiro de três dígitos para interpretação por um programa e o último é uma frase textual que pode ser entendida por uma pessoa. Os campos de ca-

<i>versão de HTTP</i>	<i>código de status</i>	<i>motivo</i>	<i>cabeçalhos</i>	<i>corpo da mensagem</i>
HTTP/1.1	200	OK		dados de recurso

Figura 5.7 Mensagem de resposta HTTP.

beçalho são usados para passar informações adicionais sobre o servidor ou dados relativos ao acesso ao recurso. Por exemplo, se a requisição exige autenticação, o *status* da resposta indica isso e um campo de cabeçalho contém um desafio. Alguns *status* de retorno têm efeitos bastante complexos. Em particular, a resposta de código de *status* 303 diz ao navegador para que examine um URL diferente, o qual é fornecido em um campo de cabeçalho na resposta. Ela é usada em uma resposta de um programa executado por meio de uma requisição *POST*, quando esse programa precisar redirecionar o navegador para um recurso selecionado.

O corpo das mensagens de requisição ou de resposta contém os dados associados ao URL especificado na requisição. O corpo da mensagem possui seus próprios cabeçalhos, especificando informações sobre os dados, como seu comprimento, tipo MIME, conjunto de caracteres, codificação do conteúdo e a data da última modificação. O campo de tipo MIME determina o tipo dos dados, por exemplo *image/jpeg* ou *text/plain*. O campo de codificação do conteúdo especifica o algoritmo de compactação a ser usado.

5.3 Chamada de procedimento remoto

Conforme mencionado no Capítulo 2, o conceito de chamada de procedimento remoto (RPC) representa um importante avanço intelectual na computação distribuída, com o objetivo de tornar a programação de sistemas distribuídos semelhante (se não idêntica) à programação convencional – isto é, obter transparência de distribuição de alto nível. Essa unificação é conseguida de maneira muito simples, estendendo a abstração de chamada de procedimento para os ambientes distribuídos. Em particular, na RPC, os procedimentos em máquinas remotas podem ser chamados como se fossem procedimentos no espaço de endereçamento local. Então, o sistema RPC oculta os aspectos importantes da distribuição, incluindo a codificação e a decodificação de parâmetros e resultados, a passagem de mensagens e a preservação da semântica exigida para a chamada de procedimento. Esse conceito foi apresentado pela primeira vez por Birrell e Nelson [1984] e abriu o caminho para muitos dos desenvolvimentos na programação de sistemas distribuídos utilizados atualmente.

5.3.1 Questões de projeto para RPC

Antes de examinarmos a implementação de sistemas de RPC, vamos ver três questões importantes para se entender esse conceito:

- o estilo de programação promovido pela RPC – programação com interfaces;
- a semântica de chamada associada à RPC;
- o problema da transparência e como ele se relaciona com as chamadas de procedimento remoto.

Programação com interfaces • A maioria das linguagens de programação modernas fornece uma maneira de organizar um programa como um conjunto de módulos que podem se comunicar. A comunicação entre os módulos pode ser feita por meio de chamadas

de procedimento entre eles ou pelo acesso direto às variáveis de outro módulo. Para controlar as possíveis interações entre os módulos, é definida uma *interface* explícita para cada módulo. A interface de um módulo especifica os procedimentos e as variáveis que podem ser acessadas a partir de outros módulos. Os módulos são implementados de forma a ocultar todas as informações sobre eles, exceto o que está disponível por meio de sua interface. Contanto que sua interface permaneça a mesma, a implementação pode ser alterada sem afetar os usuários do módulo.

Interfaces em sistemas distribuídos: em um programa distribuído, os módulos podem ser executados em processos distintos. No modelo cliente-servidor, em particular, cada servidor fornece um conjunto de procedimentos que estão disponíveis para uso pelos clientes. Por exemplo, um servidor de arquivos forneceria procedimentos para ler e escrever em arquivos. O termo *interface de serviço* é usado para se referir à especificação dos procedimentos oferecidos por um servidor, definindo os tipos dos argumentos de cada um dos procedimentos.

Há várias vantagens na programação com interfaces nos sistemas distribuídos, resultantes da importante separação entre interface e implementação:

- Assim como acontece com qualquer forma de programação modular, os programadores só se preocupam com a abstração oferecida pela interface de serviço e não precisam conhecer os detalhes da implementação.
- Extrapolando para sistemas distribuídos (potencialmente heterogêneos), os programadores também não precisam conhecer a linguagem de programação nem a plataforma de base utilizadas para implementar o serviço (um avanço importante no gerenciamento da heterogeneidade em sistemas distribuídos).
- Essa estratégia fornece suporte natural para a evolução de *software*, pois as implementações podem mudar, desde que a interface (a visão externa) permaneça a mesma. Mais corretamente, a interface também pode mudar, contanto que permaneça compatível com a original.

A definição de interfaces de serviço é influenciada pela natureza distribuída da infraestrutura subjacente:

- Não é possível um módulo ser executado em um processo e acessar as variáveis de um módulo em outro processo. Portanto, a interface de serviço não pode especificar acesso direto às variáveis. Note que as interfaces IDL do CORBA podem especificar atributos, o que parece violar essa regra. Entretanto, os atributos não são acessados diretamente, mas por meio de alguns procedimentos de obtenção e atribuição (*getter* e *setter*), adicionados automaticamente na interface.
- Os mecanismos de passagem de parâmetros usados nas chamadas de procedimento local – por exemplo, chamada por valor e chamada por referência – não são convenientes quando o chamador e o procedimento estão em processos diferentes. Em particular, a chamada por referência não é suportada. Em vez disso, a especificação de um procedimento na interface de um módulo em um programa distribuído descreve os parâmetros como sendo de *entrada* ou de *saída* ou, às vezes, ambos. Os parâmetros de *entrada* são passados para o servidor remoto pelo envio dos valores dos argumentos na mensagem de requisição e, então, repassados como argumentos para a operação a ser executada no servidor. Os parâmetros de *saída* são retornados na mensagem de resposta e são usados como resultado da chamada ou para substi-

```
// Arquivo de entrada Person.idl
struct Person {
    string name;
    string place;
    long year;
};

interface PersonList {
    readonly attribute string listname;
    void addPerson(in Person p);
    void getPerson(in string name, out Person p);
    long number();
};
```

Figura 5.8 Exemplo de IDL do CORBA.

tuir os valores das variáveis correspondentes no ambiente de chamada. Quando um parâmetro é usado tanto para entrada como para saída, o valor deve ser transmitido nas mensagens de requisição e de resposta.

- Outra diferença entre módulos locais e remotos é que os endereços de um processo não são válidos em outro processo remoto. Portanto, os endereços não podem ser passados como argumentos nem retornados como resultado de chamadas para módulos remotos.

Essas restrições têm um impacto significativo na especificação de linguagens de definição de interface, conforme discutido a seguir.

Linguagens de definição de interface: um mecanismo de RPC pode ser integrado a uma linguagem de programação em particular, caso inclua uma notação adequada para definir interfaces, permitindo que os parâmetros de entrada e saída sejam mapeados no uso normal de parâmetros da linguagem. Essa estratégia é útil quando todas as partes de um aplicativo distribuído podem ser escritas na mesma linguagem. Ela também é conveniente, pois permite que o programador utilize uma única linguagem, por exemplo, Java, para invocação local e remota.

Entretanto, muitos serviços úteis existentes são escritos em C++ e em outras linguagens. Seria vantajoso permitir que programas escritos em uma variedade de linguagens, incluindo Java, acessassem-nos de forma remota. As *linguagens de definição de interface* (ou IDLs) são projetadas para permitir que procedimentos implementados em diferentes linguagens invoquem uns aos outros. Uma IDL fornece uma notação para definir interfaces, na qual cada um dos parâmetros de uma operação pode ser descrito como sendo de *entrada* ou de *saída*, além de ter seu tipo especificado.

A Figura 5.8 mostra um exemplo simples de IDL do CORBA. A estrutura *Person* é a mesma usada para ilustrar o empacotamento na Seção 4.3.1. A interface chamada *PersonList* especifica os métodos disponíveis para RMI em um objeto remoto que implementa essa interface. Por exemplo, o método *addPerson* especifica seu argumento como *in*, significando que se trata de um argumento de *entrada* (*input*); e o método *getPerson*, que recupera uma instância de *Person* pelo nome, especifica seu segundo argumento como *out*, significando que se trata de um argumento de *saída* (*output*).

Medidas de tolerância a falhas			Semântica de chamada
Reenvio da mensagem de requisição	Filtragem de duplicatas	Reexecução de procedimento ou retransmissão da resposta	
Não	Não aplicável	Não aplicável	Talvez
Sim	Não	Executa o procedimento novamente	Pelo menos uma vez
Sim	Sim	Retransmite a resposta	No máximo uma vez

Figura 5.9 Semânticas de chamada.

O conceito de IDL foi desenvolvido inicialmente para sistemas RPC, mas se aplica igualmente à RMI e também a serviços Web. Nossos estudos de caso incluem:

- XDR da Sun como exemplo de IDL para RPC (na Seção 5.3.3);
- IDL do CORBA como exemplo de IDL para RMI (no Capítulo 8 e também incluído anteriormente);
- a linguagem de descrição de serviços Web (WSDL, Web Services Description Language), que é projetada para uma RPC na Internet que suporte serviços Web (veja a Seção 9.3);
- e buffers de protocolo utilizados no Google para armazenar e trocar muitos tipos de informações estruturadas (veja a Seção 21.4.1).

Semântica de chamada RPC • Os protocolos de requisição-resposta foram discutidos na Seção 5.2, em que mostramos que a operação *doOperation* pode ser implementada de várias maneiras para fornecer diferentes garantias de entrega. As principais escolhas são:

Retentativa de mensagem de requisição: para retransmitir a mensagem de requisição até que uma resposta seja recebida ou que se presuma que o servidor falhou.

Filtragem de duplicatas: quando são usadas retransmissões para eliminar requisições duplicadas no servidor.

Retransmissão de resultados: para manter um histórico das mensagens de respostas a fim de permitir que os resultados perdidos sejam retransmitidos sem uma nova execução das operações no servidor.

Combinações dessas escolhas levam a uma variedade de semânticas possíveis para a confiabilidade das invocações remotas vistas pelo ativador. A Figura 5.9 mostra as escolhas de interesse, com os nomes correspondentes da semântica que produzem. Note que, para chamadas de procedimentos locais, a semântica é *exatamente uma vez*, significando que todo procedimento é executado exatamente uma vez (exceto no caso de falha de processo). As escolhas de semântica de invocação RPC estão definidas a seguir.

Semântica talvez: com a semântica *talvez*, a chamada de procedimento remoto pode ser executada uma vez ou não ser executada. A semântica *talvez* surge quando nenhuma das medidas de tolerância a falhas é aplicada e pode sofrer os seguintes tipos de falha:

- falhas por omissão, se a mensagem de requisição ou de resultado for perdida;
- falhas por colapso, quando o servidor que contém a operação remota falha.

Se a mensagem de resultado não tiver sido recebida após um tempo limite e não houver novas tentativas, não haverá certeza se o procedimento foi executado. Se a mensagem de requisição foi perdida, então o procedimento não foi executado. Por outro lado, o procedimento pode ter sido executado e a mensagem de resultado, perdida. Uma falha por colapso pode ocorrer antes ou depois do procedimento ser executado. Além disso, em um sistema assíncrono, o resultado da execução do procedimento pode chegar após o tempo limite. A semântica *talvez* é útil apenas para aplicações nas quais são aceitáveis chamadas mal-sucedidas ocasionais.

Semântica pelo menos uma vez: com a semântica *pelo menos uma vez*, o invocador recebe um resultado (no caso em que sabe que o procedimento foi executado pelo menos uma vez) ou recebe uma exceção, informando-o de que nenhum resultado foi recebido. A semântica *pelo menos uma vez* pode ser obtida pela retransmissão das mensagens de requisição, o que mascara as falhas por omissão da mensagem de requisição ou de resultado. A semântica *pelo menos uma vez* pode sofrer dos seguintes tipos de falha:

- falhas por colapso, quando o servidor que contém o procedimento remoto falha;
- falhas arbitrárias – nos casos em que a mensagem de requisição é retransmitida, o servidor remoto pode recebê-la e executar o procedimento mais de uma vez, possivelmente causando o armazenamento ou o retorno de valores errados.

A Seção 5.2 definiu uma *operação idempotente* como aquela que pode ser executada repetidamente, com o mesmo efeito de que se tivesse sido executada exatamente uma vez. As operações não idempotentes podem ter o efeito errado se forem executadas mais de uma vez. Por exemplo, uma operação para aumentar um saldo bancário em \$10 deve ser executada apenas uma vez; se ela fosse repetida, o saldo aumentaria e aumentaria! Se as operações em um servidor podem ser projetadas de modo que todos os procedimentos em suas interfaces remotas sejam operações idempotentes, então a semântica *pelo menos uma vez* pode ser aceitável.

Semântica no máximo uma vez: com a semântica *no máximo uma vez*, ou o chamador recebe um resultado – no caso em que o chamador sabe que o procedimento foi executado exatamente uma vez – ou recebe uma exceção informando-o de que nenhum resultado foi recebido – no caso em que o procedimento terá sido executado uma vez ou não terá sido executado. A semântica *no máximo uma vez* pode ser obtida pelo uso de todas as medidas de tolerância a falhas resumidas na Figura 5.9. Como no caso anterior, o emprego de tentativas mascara as falhas por omissão das mensagens de requisição ou de resultado. Esse conjunto de medidas de tolerância a falhas evita falhas arbitrárias garantindo que, para cada RPC, um procedimento nunca seja executado mais de uma vez. A RPC da Sun (discutida na Seção 5.3.3) fornece semântica de chamada *pelo menos uma vez*.

Transparência • Os criadores da RPC, Birrell e Nelson [1984], pretendiam tornar as chamadas de procedimentos remotos o mais parecidas possível com as chamadas de procedimentos locais, sem nenhuma distinção na sintaxe entre uma chamada de procedimento local e um remoto. Todas as chamadas necessárias para procedimentos de empacotamento e trocas de mensagens foram ocultadas do programador que faz a chamada. Embora as mensagens de requisição sejam retransmitidas após um tempo limite, isso é transparente para o chamador, a fim de tornar a semântica das chamadas de procedimento remoto iguais às das chamadas de procedimento local.

Mais precisamente, voltando à terminologia do Capítulo 1, a RPC tenta oferecer pelo menos transparência de localização e de acesso, ocultando o local físico do procedi-

mento (potencialmente remoto) e também acessando procedimentos locais e remotos da mesma maneira. O *middleware* também pode oferecer níveis de transparência adicionais para RPC.

Entretanto, as chamadas de procedimento remoto são mais vulneráveis às falhas do que as locais, pois envolvem uma rede, outro computador e outro processo. Qualquer que seja a semântica escolhida, há sempre a chance de que nenhum resultado seja recebido e, em caso de falha, é impossível distinguir entre falha da rede e do processo servidor remoto. Isso exige que os clientes que estão fazendo chamadas remotas possam se recuperar de tais situações.

A latência de um procedimento remoto é muito maior do que a de um local. Isso sugere que os programas que utilizam chamadas remotas precisam levar esse fator em consideração, talvez minimizando as interações remotas. Os projetistas do Argus [Liskov e Scheifler 1982] sugeriram que deveria ser possível para um chamador cancelar uma chamada de procedimento remoto que esteja demorando demais, de tal maneira que isso não tenha efeito sobre o servidor. Para possibilitar isso, o servidor precisaria restaurar as variáveis para o estado em que estavam antes que o procedimento fosse chamado. Esses problemas serão discutidos no Capítulo 16.

As chamadas de procedimento remoto também exigem um estilo de passagem de parâmetros diferente, conforme discutido anteriormente. Em particular, a RPC não oferece chamada por referência.

Waldo *et al.* [1994] dizem que a diferença entre operações locais e remotas deve ser expressa na interface de serviço, para permitir que os participantes reajam de maneira consistente às possíveis falhas parciais. Outros sistemas foram ainda mais longe, argumentando que a sintaxe de uma chamada remota deve ser diferente da de uma chamada local – no caso do Argus, a linguagem foi ampliada para tornar as operações remotas explícitas para o programador.

A escolha quanto ao fato de a RPC ser transparente também está disponível para os projetistas de IDLs. Por exemplo, em algumas IDLs, uma invocação remota pode lançar uma exceção quando o cliente não consegue se comunicar com um procedimento remoto. Isso exige que o programa cliente trate de tais exceções, permitindo que ele lide com essas falhas. Uma IDL também pode fornecer um recurso para especificar a semântica de chamada de um procedimento. Isso pode ajudar o projetista do serviço – por exemplo, se a semântica *pelo menos uma vez* for escolhida para evitar as sobrecargas da semântica *no máximo uma vez*, as operações deverão ser projetadas para serem idempotentes.

O consenso atual é o de que as chamadas remotas devem se tornar transparentes, no sentido de a sintaxe de uma chamada remota ser a mesma de uma chamada local, mas a diferença entre chamadas locais e remotas deve ser expressa em suas interfaces.

5.3.2 Implementação de RPC

Os componentes de *software* exigidos para implementar RPC estão mostrados na Figura 5.10. O cliente que acessa um serviço inclui um *procedimento stub* para cada procedimento da interface de serviço. O procedimento *stub* se comporta como um procedimento local para o cliente, mas em vez de executar a chamada, ele empacota o identificador de procedimento e os argumentos em uma mensagem de requisição, a qual envia para o servidor por meio de seu módulo de comunicação. Quando a mensagem de resposta chega, ele desempacota os resultados. O processo servidor contém um despachante junto a um procedimento *stub* de servidor e um procedimento de serviço para cada procedimento na interface de serviço. O despachante seleciona um dos procedimentos *stub* de servidor, de

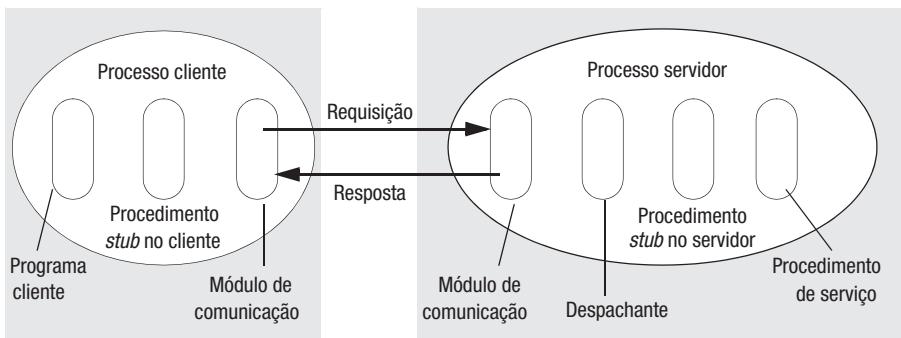


Figura 5.10 Função dos procedimentos *stub* no cliente e no servidor na RPC.

acordo com o identificador de procedimento presente na mensagem de requisição. Então, o procedimento *stub* de servidor desempacota os argumentos presentes na mensagem de requisição, chama o procedimento de serviço correspondente e empacota os valores de retorno para a mensagem de resposta. Os procedimentos de serviço implementam os procedimentos da interface de serviço. Os procedimentos *stub* do cliente e do servidor e o despachante podem ser gerados automaticamente por um compilador de interface, a partir da definição de interface do serviço.

Geralmente, a RPC é implementada sobre um protocolo de requisição-resposta, como o discutido na Seção 5.2. O conteúdo das mensagens de requisição e de resposta é igual ao ilustrado para os protocolos de requisição-resposta na Figura 5.4. A RPC pode ser implementada de modo a ter uma das escolhas de semântica de invocação discutidas na Seção 5.3.1 – *pelo menos uma vez ou no máximo uma vez* são geralmente escolhidas. Para conseguir isso, o módulo de comunicação implementará as escolhas de projeto desejadas, em termos das retransmissões de requisições, tratando de duplicatas e da retransmissão de resultados, como mostrado na Figura 5.9.

5.3.3 Estudo de caso: RPC da Sun

A RFC 1831 [Srinivasan 1995a] descreve a RPC da Sun, que foi projetada para comunicação cliente-servidor no sistema de arquivos de rede NFS (Network File System). A RPC da Sun é também chamada de RPC ONC (Open Network Computing). Ela é fornecida como parte dos vários sistemas operacionais da Sun e em outros, derivados do UNIX, estando também disponível em instalações de NFS. Os implementadores têm a opção de usar chamadas de procedimento remoto sobre UDP ou TCP. Quando a RPC da Sun é usada com UDP, o comprimento das mensagens de requisição e de resposta é limitado – teoricamente, em 64 quilobytes –, mas, na prática, mais frequentemente, em 8 ou 9 quilobytes. A semântica usada é a *pelo menos uma vez*. Existe a opção de usar *broadcast* de RPC.

O sistema RPC da Sun fornece uma linguagem de interface chamada XDR (External Data Representation) e um compilador de interface chamado *rpcgen*, destinado a ser usado com a linguagem de programação C.

Linguagem de definição de interface • A linguagem XDR da Sun, originalmente projetada para especificar representações externas de dados, foi estendida para se tornar uma linguagem de definição de interface. Ela pode ser usada para definir uma interface de serviço para RPC, por meio da especificação de um conjunto de definições de procedi-

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
    int length;
    char buffer[MAX];
};

struct writeargs {
    FileIdentifier f;
    FilePointer position;
    Data data;
};

struct readargs {
    FileIdentifier f;
    FilePointer position;
    Length length;
};

program FILEREADWRITE {
    version VERSION {
        void WRITE (writeargs) = 1; 1
        Data READ (readargs) = 2; 2
        }=2;
    } = 9999;
```

Figura 5.11 Interface de arquivos na XDR da Sun.

mentos, junto a suporte para definições de tipo. A notação é bastante primitiva em comparação com a usada pela IDL do CORBA ou Java. Em particular:

- A maioria das linguagens permite a especificação de nomes de interface, mas a RPC da Sun não permite – em vez disso, são fornecidos um número de programa e um número de versão. Os números de programa podem ser obtidos a partir de uma autoridade central, para permitir que cada programa tenha seu próprio número exclusivo. O número de versão muda quando uma assinatura de procedimento é alterada. Tanto o número de programa como o número de versão são passados na mensagem de requisição para que o cliente e o servidor possam verificar se estão usando a mesma versão.
- A definição de um procedimento especifica uma assinatura e um número. O número do procedimento é usado como identificador nas mensagens de requisição.
- É permitido apenas um parâmetro de entrada. Portanto, os procedimentos que necessitam de vários parâmetros devem incluí-los como componentes de uma única estrutura.
- Os parâmetros de saída de um procedimento são retornados por meio de um único resultado.
- A assinatura de um procedimento consiste no tipo do resultado, no nome do procedimento e no tipo do parâmetro de entrada. O tipo do resultado e do parâmetro de entrada pode especificar um único valor ou uma estrutura contendo vários valores.

Por exemplo, veja a definição de XDR, na Figura 5.11, de uma interface com dois procedimentos para escrever e ler arquivos. O número do programa é 9999 e o número da versão é 2. O procedimento *READ* (linha 2) recebe como parâmetro de entrada uma estrutura com três componentes, especificando um identificador de arquivo, uma posição no arquivo e o número de bytes a serem lidos. Seu resultado é uma estrutura contendo o número de bytes retornados e os dados do arquivo. O procedimento *WRITE* (linha 1) não tem nenhum resultado. Os procedimentos *WRITE* e *READ* recebem os números 1 e 2. O número zero é reservado para um procedimento nulo, que é gerado automaticamente e se destina a testar se um servidor está disponível.

Essa linguagem de definição de interface fornece uma notação para definir constantes, *typedefs*, estruturas, tipos enumerados, uniões e programas. *Typedefs*, estruturas e tipos enumerados utilizam a sintaxe da linguagem C. A partir de uma definição de interface, o compilador de interface *rpcgen* pode ser usado para gerar o seguinte:

- procedimentos *stub* no cliente;
- procedimento *main*, despachante e procedimentos *stub* no servidor;
- procedimentos de empacotamento e desempacotamento da XDR, a serem usados pelo despachante e pelos procedimentos *stub* do cliente e do servidor.

Vinculação (binding) • A RPC da Sun executa em cada computador em um número de porta bem conhecido um serviço de vinculação (*binding*) local chamado *mapeador de porta* (*port mapper*). Cada instância de um mapeador de porta grava o número do programa, o número da versão e o número da porta em uso para cada um dos serviços em execução local. Quando um servidor é iniciado, ele registra seu número de programa, número de versão e número de porta no mapeador de porta. Quando um cliente é iniciado, ele descobre a porta do serviço fazendo uma requisição remota para o mapeador de porta no computador onde o servidor executa, especificando o número de programa e o número de versão.

Quando um serviço tem várias instâncias sendo executadas em diferentes computadores, elas podem usar diferentes números de porta para receber as requisições dos clientes. Se um cliente precisa enviar uma requisição para todas as instâncias de um serviço que usam diferentes números de porta, ele não pode usar *multicast* IP direto para esse propósito. A solução é os clientes fazerem chamadas de procedimento remoto por *multicast* para todos os mapeadores de porta, especificando o número do programa e da versão. Cada mapeador de porta encaminha todas essas chamadas para o programa de serviço local apropriado, se houver um.

Autenticação • As mensagens de requisição-resposta da RPC da Sun fornecem campos adicionais que permitem autenticar as informações a serem passadas entre cliente e servidor. A mensagem de requisição contém as credenciais do usuário que está executando o programa cliente. Por exemplo, na autenticação UNIX, as credenciais incluem o *uid* e o *gid* do usuário. Mecanismos de controle de acesso podem ser construídos sobre informações de autenticação repassadas aos procedimentos do servidor por intermédio de um segundo argumento. O programa servidor é responsável por impor o controle de acesso, decidindo se vai ou não executar cada chamada de procedimento, de acordo com as informações de autenticação. Por exemplo, ao se tratar de um servidor de arquivos NFS, ele pode verificar se o usuário tem direitos suficientes para executar a operação solicitada em determinado arquivo. Vários protocolos de autenticação diferentes podem ser suportados. Isso inclui:

- nenhum;
- estilo UNIX, como descrito anteriormente;

- baseado em uma chave compartilhada para assinar as mensagens RPC;
- Kerberos (veja o Capítulo 11).

Um campo no cabeçalho RPC indica qual estilo está sendo usado.

Uma estratégia de segurança mais genérica está descrita na RFC 2203 [Eisler *et al.* 1997]. Ela proporciona sigilo e integridade das mensagens RPC, assim como autenticação. A estratégia permite que cliente e servidor negoциem um contexto de segurança em que escolhem não usar segurança alguma ou, no caso de ser exigida segurança, garantir integridade das mensagens, garantir privacidade das mensagens ou garantir ambas.

Programas clientes e servidores • Mais material sobre a RPC da Sun está disponível no endereço [www.cdk5.net/rmi] (em inglês). Ele inclui exemplos de programas clientes e servidores correspondentes à interface definida na Figura 5.11.

5.4 Invocação a método remoto

A RMI (Remote Method Invocation – invocação a método remoto) está intimamente relacionada à RPC, mas é estendida para o mundo dos objetos distribuídos. Na RMI, um objeto chamador pode invocar um método em um objeto potencialmente remoto. Assim como na RPC, geralmente os detalhes subjacentes ficam ocultos para o usuário. As características comuns entre a RMI e a RPC são:

- Ambas suportam programação com interfaces, com as vantagens resultantes advindas dessa estratégia (veja a Seção 5.3.1).
- Normalmente, ambas são construídas sobre protocolos de requisição-resposta e podem oferecer diversas semânticas de chamada, como *pelo menos uma vez e no máximo uma vez*.
- Ambas oferecem um nível de transparência semelhante – isto é, as chamadas locais e remotas empregam a mesma sintaxe, mas as interfaces remotas normalmente expõem a natureza distribuída da chamada subjacente, suportando exceções remotas, por exemplo.

As diferenças a seguir levam a uma maior expressividade na programação de aplicações e serviços distribuídos complexos.

- O programador pode usar todo o poder expressivo da programação orientada a objetos no desenvolvimento de *software* de sistemas distribuídos, incluindo o uso de objetos, classes e herança, e também pode empregar metodologias de projeto orientado a objetos relacionadas e ferramentas associadas.
- Complementando o conceito de identidade de objeto dos sistemas orientados a objetos, em um sistema baseado em RMI, todos os objetos têm referências exclusivas (sejam locais ou remotos) e tais referências também podem ser passadas como parâmetros, oferecendo, assim, uma semântica de passagem de parâmetros significativamente mais rica do que na RPC.

A questão da passagem de parâmetros é particularmente importante nos sistemas distribuídos. A RMI permite ao programador passar parâmetros não apenas por valor, como parâmetros de entrada ou saída, mas também por referência de objeto. Passar referências é particularmente atraente se o parâmetro for grande ou complexo. Então, o lado remoto, ao receber uma referência de objeto, pode acessar esse objeto usando invocação a método remoto, em vez de transmitir o valor do objeto pela rede.

O restante desta seção examinará o conceito de invocação a método remoto com mais detalhes, considerando inicialmente os principais problemas envolvidos nos modelos de objeto distribuído, antes de observar os problemas de implementação relacionados à RMI, incluindo a coleta de lixo distribuída.

5.4.1 Questões de projeto para RMI

Conforme mencionado anteriormente, a RMI compartilha os mesmos problemas de projeto da RPC em termos de programação com interfaces, semântica de chamada e nível de transparência. O leitor deve consultar a Seção 5.3.1 para uma discussão sobre esses itens.

O principal problema de projeto se relaciona com o modelo de objeto e, em particular, com a transição de objetos para objetos distribuídos. Descreveremos primeiro o modelo de objeto de imagem única convencional e, depois, consideraremos o modelo de objeto distribuído.

O modelo de objeto • Um programa orientado a objetos, por exemplo em Java ou C++, consiste em um conjunto de objetos interagindo, cada um dos quais composto de um conjunto de dados e um conjunto de métodos. Um objeto se comunica com outros objetos invocando seus métodos, geralmente passando argumentos e recebendo resultados. Os objetos podem encapsular seus dados e o código de seus métodos. Algumas linguagens, como Java e C++, permitem que os programadores definam objetos cujas variáveis de instância podem ser acessadas diretamente. No entanto, para uso em um sistema de objeto distribuído, os dados de um objeto devem ser acessíveis somente por intermédio de seus métodos.

Referências de objeto: os objetos podem ser acessados por meio de referências de objeto. Por exemplo, em Java, uma variável que parece conter um objeto, na verdade, contém uma referência para esse objeto. Para invocar um método em um objeto, são fornecidos a referência do objeto e o nome do método, junto aos argumentos necessários. O objeto cujo método é invocado é chamado de *alvo*, de *destino* e, às vezes, de *receptor*. As referências de objeto são valores de primeira classe, significando que eles podem, por exemplo, ser atribuídos a variáveis, passados como argumentos e retornados como resultados de métodos.

Interfaces: uma interface fornece a definição das assinaturas de um conjunto de métodos (isto é, os tipos de seus argumentos, valores de retorno e exceções), sem especificar sua implementação. Um objeto fornecerá uma interface em particular, caso sua classe contenha código que implemente os métodos dessa interface. Em Java, uma classe pode implementar várias interfaces, e os métodos de uma interface podem ser implementados por qualquer classe. Uma interface também define os tipos que podem ser usados para declarar o tipo de variáveis ou dos parâmetros e valores de retorno dos métodos. Note que as interfaces não têm construtores.

Ações: em um programa orientado a objetos, a ação é iniciada por um objeto invocando um método em outro objeto. Uma invocação pode incluir informações adicionais (argumentos) necessárias para executar o método. O receptor executa o método apropriado e depois retorna o controle para o objeto que fez a invocação, algumas vezes fornecendo um resultado. A ativação de um método pode ter três efeitos:

1. O estado do receptor pode ser alterado.
2. Um novo objeto pode ser instanciado; por exemplo, pelo uso de um construtor em Java ou C++.
3. Outras invocações podem ocorrer nos métodos de outros objetos.

Como uma invocação pode levar a mais invocações a métodos em outros objetos, uma ação é um encadeamento de invocações relacionadas de métodos, cada uma com seu respectivo retorno.

Exceções: os programas podem encontrar muitos tipos de erros e condições inesperadas, de diversos graus de gravidade. Durante a execução de um método, muitos problemas diferentes podem ser descobertos: por exemplo, valores inconsistentes nas variáveis do objeto ou falhas nas tentativas de ler ou gravar arquivos ou soquetes de rede. Quando os programadores precisam inserir testes em seu código para tratar de todos os casos incomuns ou errôneos possíveis, isso diminui a clareza do caso normal. As exceções proporcionam uma maneira clara de tratar com condições de erro, sem complicar o código. Além disso, cada cabeçalho de método lista explicitamente como exceções as condições de erro que pode encontrar, permitindo que os usuários desse método as tratem adequadamente. Pode ser definido um bloco de código para *disparar* uma exceção, quando surgirem condições inesperadas ou erros em particular. Isso significa que o controle passa para outro bloco de código, que *captura* a exceção. O controle não retorna para o lugar onde a exceção foi disparada.

Coleta de lixo (garbage collection): é necessário fornecer uma maneira de liberar o espaço em memória ocupado pelos objetos quando eles não são mais necessários. Uma linguagem (por exemplo, Java) que pode detectar automaticamente quando um objeto não está mais acessível, recupera o espaço em memória e o torna disponível para alocação por outros objetos. Esse processo é chamado de *coleta de lixo (garbage collection)*. Quando uma linguagem não suporta coleta de lixo (por exemplo, C++), o programador tem de se preocupar com a liberação do espaço alocado para os objetos. Isso pode ser uma fonte de erros significativa.

Objetos distribuídos • O estado de um objeto consiste nos valores de suas variáveis de instância. No paradigma baseado em objetos, o estado de um programa é dividido em partes separadas, cada uma associada a um objeto. Como os programas baseados em objetos são logicamente particionados, a distribuição física dos objetos em diferentes processos ou computadores de um sistema distribuído é uma extensão natural (o problema do posicionamento foi discutido na Seção 2.3.1).

Os sistemas de objetos distribuídos podem adotar a arquitetura cliente-servidor. Nesse caso, os objetos são gerenciados pelos servidores, e seus clientes invocam seus métodos usando invocação a método remoto, RMI. Na RMI, a requisição de um cliente para invocar um método de um objeto é enviada em uma mensagem para o servidor que gerencia o objeto. A invocação é realizada pela execução de um método do objeto no servidor e o resultado é retornado para o cliente em outra mensagem. Para permitir encadeamentos de invocações relacionadas, os objetos nos servidores podem se tornar clientes de objetos em outros servidores.

Os objetos distribuídos podem adotar outros modelos arquiteturais. Por exemplo, os objetos podem ser replicados para usufruir as vantagens normais da tolerância a falhas e do melhor desempenho, e os objetos podem ser migrados com o intuito de melhorar seu desempenho e sua disponibilidade.

O fato de ter objetos clientes e servidores em diferentes processos impõe o encapsulamento. Isso significa que o estado de um objeto pode ser acessado somente pelos métodos do objeto, implicando que não é possível a métodos não autorizados agirem livremente sobre o estado. Por exemplo, a possibilidade de RMIs concorrentes, a partir de objetos em diferentes computadores, significa que um objeto pode ser acessado de forma concorrente. Portanto, surge a possibilidade de conflito de acessos. Entretanto, o fato de os dados de um objeto serem acessados somente pelos próprios métodos permite que os objetos forneçam métodos para protegerem-se contra acessos incorretos. Neste caso, por exemplo, eles podem usar primitivas de sincronização, como variáveis de condição, para proteger o acesso a suas variáveis de instância.

Outra vantagem de tratar o estado compartilhado de um programa distribuído como um conjunto de objetos é que um objeto pode ser acessado via RMI ou ser copiado em

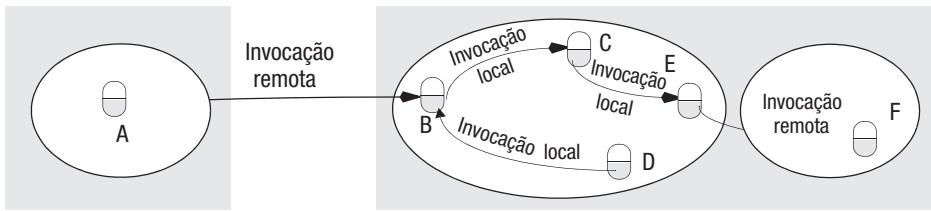


Figura 5.12 Invocação a métodos locais e remotos.

uma cache local e acessado diretamente, desde que a implementação da classe esteja disponível de forma local.

O fato de os objetos serem acessados somente por meio de seus métodos oferece outra vantagem para sistemas heterogêneos, pois diferentes formatos de dados podem ser usados em diferentes instalações – esses formatos não serão notados pelos clientes que utilizam RMI para acessar os métodos dos objetos.

O modelo de objeto distribuído • Esta seção discute as extensões feitas no modelo de objeto para torná-lo aplicável aos objetos distribuídos. Cada processo contém um conjunto de objetos, alguns dos quais podem receber invocações locais e remotas, enquanto os outros objetos podem receber somente invocações locais, como mostrado na Figura 5.12. As invocações a métodos entre objetos em diferentes processos, sejam no mesmo computador ou não, são conhecidas como *invocações a métodos remotos*. As invocações a métodos entre objetos no mesmo processo são invocações a métodos locais.

Referimo-nos aos objetos que podem receber invocações remotas como *objetos remotos*. Na Figura 5.12, B e F são objetos remotos. Todos os objetos podem receber invocações locais de outros objetos que contenham referências a eles. Por exemplo, o objeto C deve ter uma referência ao objeto E para poder invocar um de seus métodos. Os dois conceitos fundamentais a seguir estão no centro do modelo de objeto distribuído:

Referência de objeto remoto: outros objetos podem invocar os métodos de um objeto remoto se tiverem acesso a sua *referência de objeto remoto*. Por exemplo, na Figura 5.12, uma referência de objeto remoto de B deve estar disponível para A.

Interface remota: todo objeto remoto tem uma interface remota especificando qual de seus métodos pode ser invocado de forma remota. Por exemplo, os objetos B e F devem ter interfaces remotas.

Os parágrafos a seguir discutem as referências de objeto remoto, as interfaces remotas e outros aspectos do modelo de objeto distribuído.

Referências de objeto remoto: a noção de referência de objeto é estendida para permitir que qualquer objeto que possa receber uma RMI tenha uma referência de objeto remoto. Uma referência de objeto remoto é um identificador que pode ser usado por todo um sistema distribuído para se referir a um objeto remoto único em particular. Sua representação, que geralmente é diferente da representação de referências de objeto local, foi discutida na Seção 4.3.4. As referências de objeto remoto são análogas às locais, pois:

1. O objeto remoto que vai receber uma invocação a método remoto é especificado pelo *invocador* como uma referência de objeto remoto.
2. As referências de objeto remoto podem ser passadas como argumentos e resultados de invocações a métodos remotos.

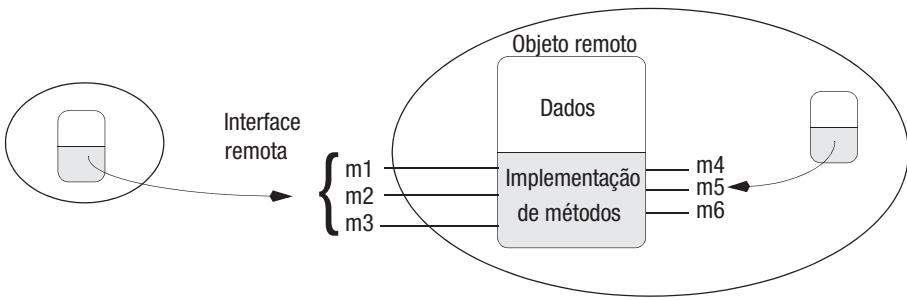


Figura 5.13 Um objeto remoto e sua interface remota.

Interfaces remotas: a classe de um objeto remoto implementa os métodos de sua interface remota; por exemplo, como métodos de instância públicos em Java. Objetos em outros processos só podem invocar os métodos pertencentes à interface remota, como mostrado na Figura 5.13. Os objetos locais podem invocar os métodos da interface remota, assim como outros métodos implementados por um objeto remoto. Note que as interfaces remotas, assim como todas as interfaces, não têm construtores.

O sistema CORBA fornece uma linguagem de definição de interface (IDL) que é usada para definir interfaces remotas. Veja, na Figura 5.8, um exemplo de interface remota definida na IDL do CORBA. As classes de objetos remotos e os programas clientes podem ser implementados em qualquer linguagem, como C++, Java ou Python, para a qual esteja disponível um compilador de IDL. Os clientes CORBA não precisam usar a mesma linguagem que o objeto remoto para invocar seus métodos de forma remota.

Na RMI Java, as interfaces remotas são definidas da mesma forma que qualquer outra interface Java. Elas adquirem a capacidade de ser interfaces remotas estendendo uma interface como *Remote*. Tanto a IDL do CORBA (Capítulo 8) como a linguagem Java suportam herança múltipla de interfaces, isto é, uma interface pode estender uma ou mais interfaces.

Ações em um sistema de objeto distribuído • Assim como no caso não distribuído, uma ação é iniciada por uma invocação a método, a qual pode resultar em mais invocações a métodos em outros objetos. Porém, no caso distribuído, os objetos envolvidos em um encadeamento de invocações relacionadas podem estar localizados em diferentes processos ou em diferentes computadores. Quando uma invocação cruza o limite de um processo ou computador, a RMI é usada e a referência remota do objeto deve estar disponível para o invocador. Na Figura 5.12, o objeto A precisa conter uma referência de objeto remoto para o objeto B. As referências de objeto remoto podem ser obtidas como resultado de invocações a métodos remotos. Por exemplo, na Figura 5.12, o objeto A poderia obter uma referência remota para o objeto F a partir do objeto B.

Quando uma ação levar à instanciação de um novo objeto, esse objeto normalmente ficará dentro do processo em que a instanciação é feita; por exemplo, onde o construtor foi usado. Se o objeto recentemente instanciado tiver uma interface remota, ele será um objeto remoto com uma referência de objeto remoto.

Os aplicativos distribuídos podem fornecer objetos remotos com métodos para instanciar objetos que podem ser acessados por uma RMI, fornecendo assim, efetivamente, o efeito da instanciação remota de objetos. Suponha, por exemplo, que o objeto L da Figura 5.14 contivesse um método para criar objetos remotos e que as invocações remotas de C e K pudessem levar à instanciação dos objetos M e N, respectivamente.

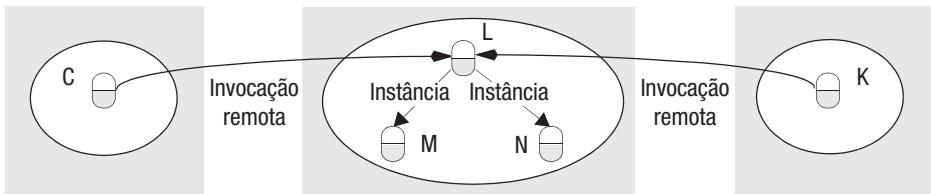


Figura 5.14 Instanciação de objetos remotos.

Coleta de lixo em um sistema de objeto distribuído: se uma linguagem, por exemplo Java, suporta coleta de lixo, então qualquer sistema RMI associado deve permitir a coleta de lixo de objetos remotos. A coleta de lixo distribuída geralmente é obtida pela cooperação entre o coletor de lixo local existente e um módulo adicionado que realiza uma forma de coleta de lixo distribuída, normalmente baseada em contagem de referência. A Seção 5.4.3 descreverá esse esquema em detalhes. Se a coleta de lixo não estiver disponível, os objetos remotos que não são mais necessários deverão ser excluídos.

Exceções: uma invocação remota pode falhar por motivos relacionados ao fato de o objeto invocado estar em um processo ou computador diferente do invocador. Por exemplo, o processo que contém o objeto remoto pode ter falhado ou estar ocupado demais para responder, ou a mensagem de invocação ou de resultado pode ter se perdido. Portanto, a invocação a método remoto deve ser capaz de lançar exceções, como expiração de tempo limite (*timeout*), causados pelo envio de mensagens, assim como as ocorridas durante a execução do método invocado. Exemplos destas últimas são uma tentativa de ler além do final de um arquivo ou de acessar um arquivo sem as permissões corretas.

A IDL do CORBA fornece uma notação para especificar exceções em nível de aplicativo e o sistema subjacente gera exceções padrão quando ocorrem erros devido à distribuição de mensagens. Os programas clientes CORBA precisam ser capazes de tratar as exceções. Por exemplo, um programa cliente em C++ usará os mecanismos de exceção do C++.

5.4.2 Implementação de RMI

Vários objetos e módulos separados estão envolvidos na realização de uma invocação a método remoto. Eles aparecem na Figura 5.15, na qual um objeto em nível de aplicativo A invoca um método em um objeto remoto B, para o qual mantém uma referência de objeto remoto. Esta seção discute as funções de cada um dos componentes mostrados nessa figura, tratando primeiro da comunicação e dos módulos de referência remota e, depois, do *software RMI* neles executado.

Em seguida, exploraremos os seguintes tópicos relacionados: a geração de *proxies*, a associação de nomes às suas referências de objeto remoto, a invocação e a passividade de objetos e a localização de objetos a partir de suas referências de objeto remoto.

Módulo de comunicação • Dois módulos de comunicação cooperam para executar o protocolo de requisição-resposta, o qual transmite mensagens de *requisição-resposta* entre o cliente e o servidor. O conteúdo das mensagens de *requisição-resposta* aparece na Figura 5.4. O módulo de comunicação usa apenas os três primeiros elementos, os quais especificam o tipo de mensagem, sua *requestId* e a referência remota do objeto a ser invocado. O identificador da operação (*OperationId*) e todo o empacotamento e desempacotamento são de responsabilidade do *software RMI*, como discutido a seguir. Juntos, os módulos

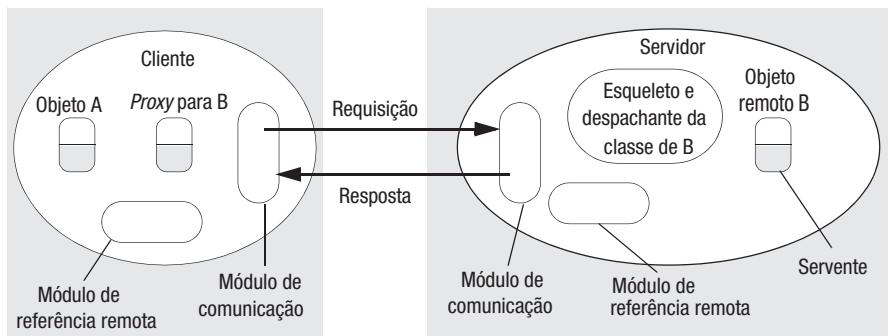


Figura 5.15 A função do *proxy* e do *esqueleto* na invocação a método remoto.

de comunicação são responsáveis por fornecer uma semântica de invocação específica, por exemplo, *no máximo uma vez*.

O módulo de comunicação no servidor seleciona o despachante para a classe do objeto a ser invocado, passando sua referência local, obtida a partir do módulo de referência remota, através do identificador do objeto remoto dado pela mensagem de requisição. A função do despachante será discutida no *software RMI*, a seguir.

Módulo de referência remota • O módulo de referência remota é responsável pela transformação entre referências de objeto local e remoto e pela criação de referências de objeto remoto. Para executar sua funcionalidade, o módulo de referência remota de cada processo contém uma *tabela de objetos remotos* para registrar a correspondência entre as referências de objeto local desse processo e as referências de objeto remoto (que abrangem todo o sistema). A tabela inclui:

- Uma entrada para todos os objetos remotos mantidos pelo processo. Por exemplo, na Figura 5.15, o objeto remoto B será registrado na tabela do servidor.
- Uma entrada para cada *proxy* local. Por exemplo, na Figura 5.15, o *proxy* de B será registrado na tabela do cliente.

A função de um *proxy* é discutida na subseção sobre *software RMI*. As ações do módulo de referência remota são as seguintes:

- Quando um objeto remoto precisa ser passado como argumento ou resultado pela primeira vez, o módulo de referência remota cria uma referência de objeto remoto, a qual adiciona em sua tabela.
- Quando uma referência de objeto remoto chega em uma mensagem de requisição ou resposta, é solicitada ao módulo de referência remota a correspondente referência de objeto local, a qual pode se referir a um *proxy* ou a um objeto remoto. No caso da referência de objeto remoto não estar na tabela, o *software RMI* cria um novo *proxy* e pede ao módulo de referência remota para adicioná-lo na tabela.

Esse módulo é chamado pelos componentes do *software RMI* quando estão empacotando e desempacotando referências de objeto remoto. Por exemplo, quando chega uma mensagem de requisição, a tabela é usada para descobrir qual objeto local deve ser invocado.

Serventes • Um *servente* é uma instância de uma classe que fornece o corpo de um objeto remoto. É o servente que trata as requisições remotas repassadas pelo esqueleto cor-

respondente. Os serventes são partes integrantes do processo servidor. Eles são criados quando os objetos remotos são instanciados e permanecem em uso até não serem mais necessários, sendo finalmente excluídos.

O software RMI • É uma camada de *software – middleware* – entre os objetos do aplicativo e os módulos de comunicação e de referência remota. As funções dos seus componentes, mostrados na Figura 5.15, são as seguintes:

Proxy: a função de um *proxy* é tornar a invocação a método remoto transparente para os clientes, comportando-se como um objeto local para o invocador; mas, em vez de executar uma invocação local, ele a encaminha em uma mensagem para um objeto remoto. Ele oculta os detalhes da referência de objeto remoto, do empacotamento de argumentos, do desempacotamento dos resultados e do envio e recepção de mensagens do cliente. Existe um *proxy* para cada objeto remoto a que um processo faz uma referência de objeto remoto. A classe de um *proxy* implementa os métodos da interface remota do objeto remoto que ele representa. Isso garante que as invocações a métodos remotos sejam apropriadas para o objeto remoto em questão. Entretanto, o *proxy* implementa os métodos de uma forma diferente. Cada método do *proxy* empacota uma referência para o objeto alvo, o *OperationId* e seus argumentos em uma mensagem de *requisição* e a envia para o objeto alvo. A seguir, espera pela mensagem de *resposta*; quando a recebe, desempacota e retorna os resultados para o invocador.

Despachante: um servidor tem um despachante e um esqueleto para cada classe que representa um objeto remoto. Em nosso exemplo, o servidor tem um despachante e um esqueleto para a classe do objeto remoto B. O despachante recebe a mensagem de *requisição* do módulo de comunicação e utiliza o *OperationId* para selecionar o método apropriado no esqueleto, repassando a mensagem de *requisição*. O despachante e o *proxy* usam o mesmo *OperationId* para os métodos da interface remota.

Esqueleto: a classe de um objeto remoto tem um *esqueleto*, que implementa os métodos da interface remota, mas de uma forma diferente dos métodos implementados no servente que personifica o objeto remoto. Um método de esqueleto desempacota os argumentos na mensagem de *requisição* e invoca o método correspondente no servente. Ele espera que a invocação termine e, em seguida, empacota o resultado, junto às exceções, em uma mensagem de *resposta* que é enviada para o método do *proxy* que fez a requisição.

As referências de objeto remoto são empacotadas na forma mostrada na Figura 4.13, que inclui informações sobre a interface remota do objeto remoto; por exemplo, o nome da interface remota ou da classe do objeto remoto. Essas informações permitem que a classe do *proxy* seja determinada para que, quando necessário, um novo *proxy* possa ser criado. Por exemplo, o nome da classe do *proxy* pode ser gerado anexando “*_proxy*” ao nome da interface remota.

Geração das classes para proxies, despachantes e esqueletos • As classes para *proxy*, despachante e esqueleto usados na RMI são geradas automaticamente por um compilador de interface. Por exemplo, na implementação Orbix do CORBA, as interfaces dos objetos remotos são definidas na IDL do CORBA e o compilador de interface pode ser usado para gerar as classes para *proxies*, despachantes e esqueletos em C++ ou em Java [www.iona.com]. Para a RMI Java, o conjunto de métodos oferecidos por um objeto

remoto é definido como uma interface Java implementada dentro da classe do objeto remoto. O compilador da RMI Java gera as classes de *proxy*, despachante e esqueleto a partir da classe do objeto remoto.

Ativação dinâmica: uma alternativa aos proxies • O *proxy* que acabamos de descrever é estático, no sentido de que sua classe é gerada a partir de uma definição de interface e, depois, compilada no código do cliente. Às vezes, isso não é prático: suponha que um programa cliente receba uma referência remota para um objeto cuja interface remota não estava disponível no momento da compilação. Neste caso, ele precisa de outra maneira para invocar o objeto remoto: isso é chamado de *invocação dinâmica* e dá ao cliente acesso a uma representação genérica de uma invocação remota, como o método *DoOperation* usado no Exercício 5.18, que está disponível como parte da infraestrutura de RMI (veja a Seção 5.4.1). O cliente fornecerá a referência de objeto remoto, o nome do método e os argumentos para *DoOperation* e depois esperará para receber os resultados.

Note que, embora a referência de objeto remoto inclua informações sobre a interface do objeto remoto, como seu nome, isso não é suficiente – os nomes dos métodos e os tipos dos argumentos são necessários para se fazer uma invocação dinâmica. O CORBA fornece suas informações por meio de um componente chamado *Interface Repository* (repositório de interfaces), que será descrito no Capítulo 8.

A interface de invocação dinâmica não é tão conveniente para usar como *proxy*, mas é útil em aplicações em que algumas das interfaces dos objetos remotos não podem ser previstas no momento do projeto. Um exemplo de tal aplicação é o quadro branco compartilhado usado para ilustrar a RMI Java (Seção 5.5), o CORBA (Capítulo 8) e os serviços Web (Seção 9.2.3). Resumindo: o aplicativo de quadro branco compartilhado exibe muitos tipos diferentes de figuras, como círculos, retângulos e linhas, mas precisa mostrar novas figuras que não foram previstas quando o cliente foi compilado. Um cliente que utilize invocação dinâmica é capaz de resolver esse problema. Na Seção 5.5, veremos que o *download* dinâmico de classes em clientes é uma alternativa à invocação dinâmica. Isso está disponível na RMI Java.

Esqueletos dinâmicos: a partir do exemplo anterior, fica claro que um servidor também precisará conter objetos remotos cujas interfaces não eram conhecidas no momento da compilação. Por exemplo, um cliente pode fornecer um novo tipo de figura para que o servidor de quadro branco compartilhado armazene. Um servidor com esqueletos dinâmicos resolveria essa situação. Vamos deixar para descrever os esqueletos dinâmicos no capítulo sobre CORBA (Capítulo 8). Entretanto, conforme veremos na Seção 5.5, a RMI Java resolve esse problema usando um despachante genérico e o *download* dinâmico de classes no servidor.

Programas clientes e servidores • O programa servidor contém as classes para os despachantes e esqueletos, junto às implementações das classes de todos os serventes que suporta. Além disso, o programa servidor contém uma seção de *inicialização* (por exemplo, em um método *main* em Java ou C++). A seção de *inicialização* é responsável por criar e inicializar pelo menos um dos serventes que fazem parte do servidor. Mais serventes podem ser criados em resposta às requisições dos clientes. A seção de *inicialização* também pode registrar alguns de seus serventes com um vinculador (*binder*) (veja o próximo parágrafo); porém, geralmente, é registrado apenas um servente, o qual pode ser usado para acessar os restantes.

O programa cliente conterá as classes dos *proxies* de todos os objetos remotos que ativará. Ele pode usar um vinculador para pesquisar referências de objeto remoto.

Métodos de fábrica: mencionamos anteriormente que as interfaces de objeto remoto não incluem construtores. Isso significa que os serventes não podem ser criados por invocação remota em construtores. Os serventes são criados pela *inicialização* ou por métodos de uma interface remota destinados a esse propósito. O termo *método de fábrica* é também usado para se referir a um método que cria serventes, e um *objeto de fábrica* é um objeto com métodos de fábrica. Todo objeto remoto que precise criar novos objetos remotos a pedido dos clientes deve fornecer métodos em sua interface remota para esse propósito. Tais métodos são chamados de métodos de fábrica, embora na verdade sejam apenas métodos normais.

O vinculador (binder) • Os programas clientes geralmente exigem uma maneira de obter uma referência de objeto remoto para pelo menos um dos objetos remotos mantidos por um servidor. Por exemplo, na Figura 5.12, o objeto A exige uma referência de objeto remoto para o objeto B. Em um sistema distribuído, um *vinculador (brinder)* é um serviço separado que mantém uma tabela contendo mapeamentos dos nomes textuais para referências de objeto remoto. Ele é usado pelos servidores para registrar seus objetos remotos pelo nome e pelos clientes para pesquisá-los. O Capítulo 8 traz uma discussão sobre o serviço de atribuição de nomes (Naming Service) do CORBA. O vinculador Java, RMI-registry, será discutido brevemente no estudo de caso sobre a RMI Java, na Seção 5.5.

Threads no servidor • Quando um objeto executa uma invocação remota, ele pode acarretar execução em mais invocações a métodos em outros objetos remotos, os quais podem levar algum tempo para retornar. Para evitar que a execução de uma invocação remota atrase a execução de outra, geralmente os servidores criam uma *thread* para cada invocação remota. Quando isso acontece, o projetista da implementação de um objeto remoto deve levar em conta os efeitos das execuções concorrentes sobre seu estado.

Invocação de objetos remotos • Algumas aplicações exigem que as informações sejam válidas por longos períodos de tempo. Entretanto, não é prático que os objetos que representam tais informações sejam mantidos por períodos ilimitados em processos que estejam em execução, particularmente porque eles não estão necessariamente em uso o tempo todo. Para evitar o potencial desperdício de recursos, devido à execução de todos os servidores que gerenciam objetos remotos, os servidores podem ser iniciados somente quando forem necessários para os clientes. Isso é análogo ao que acontece no conjunto padrão de serviços TCP, como o FTP, que são iniciados de acordo com a demanda, por um serviço chamado *Inetd*. Os processos que iniciam processos servidores para conter objetos remotos são chamados de *ativadores* pelos motivos a seguir.

Um objeto remoto é dito *ativo* quando está disponível para invocação dentro de um processo em execução, e é chamado de *passivo* se não estiver ativo no momento da invocação, mas puder se tornar. Um objeto passivo consiste em duas partes:

1. na implementação de seus métodos;
2. em seu estado na forma empacotada.

A *ativação* consiste na criação de um objeto ativo a partir do objeto passivo correspondente pela criação de uma nova instância de sua classe e pela inicialização de suas variáveis de instância a partir do estado armazenado. Os objetos passivos podem ser ativados por demanda; por exemplo, quando eles precisam ser invocados por outros objetos.

Um *ativador* é responsável por:

- Registrar os objetos passivos que estão disponíveis para ativação, o que envolve registrar os nomes dos servidores com os URLs ou nomes de arquivo dos objetos passivos correspondentes.

- Iniciar processos servidores, identificá-los e ativar objetos remotos neles.
- Controlar a localização dos servidores de objetos remotos que já tenha ativado.

A RMI Java fornece a capacidade de tornar objetos remotos *passíveis de ativação* [[java.sun.com IX](#)]. Quando um objeto passível de ativação é invocado, se já não estiver correntemente ativo, ele se tornará ativo a partir de seu estado empacotado e, depois, será invocado. A RMI Java emprega um ativador para cada computador servidor.

O estudo de caso do CORBA, no Capítulo 8, descreverá o repositório de implementação – uma forma simplificada de ativador para disparar serviços contendo objetos em um estado inicial.

Repositório de objetos persistentes • Um objeto que mantém seu estado entre invocações é chamado de *objeto persistente*. Geralmente, os objetos persistentes são gerenciados por repositórios de objetos persistentes, que guardam seus estados em uma forma empacotada no disco. Exemplos incluem o serviço de estado persistente do CORBA (veja o Capítulo 8), *Java Data Objects* [[java.sun.com VIII](#)] e *Persistent Java* [Jordan 1996, [java.sun.com IV](#)].

Em geral, um repositório de objetos persistentes gerencia uma grande quantidade de objetos, os quais são armazenados em disco ou em um banco de dados até serem necessários. Eles serão ativados quando seus métodos forem invocados por outros objetos. A ativação geralmente é projetada para ser transparente – isto é, o invocador não deve saber se um objeto já está na memória principal ou se precisa ser ativado antes de seu método ser invocado. Os objetos persistentes que não são mais necessários na memória principal podem ser postos em estado passivo. Na maioria dos casos, os objetos são salvos no repositório de objetos persistentes sempre que atingirem um estado consistente, para fornecer um grau de tolerância a falhas. O repositório de objetos persistentes precisa de uma estratégia para decidir quando deve tornar passivo os objetos. Por exemplo, ele pode fazer isso em resposta a um pedido feito pelo programa que ativou os objetos, no final de uma transação ou quando o programa termina. Geralmente, os repositórios de objetos persistentes tentam otimizar esse procedimento salvando apenas os objetos que foram modificados desde a última vez em que foram salvos.

De modo geral, os repositórios de objetos persistentes permitem que conjuntos de objetos persistentes relacionados tenham nomes legíveis por seres humanos, como nomes de caminho ou URLs. Na prática, cada um desses nomes é associado à raiz de um conjunto de objetos persistentes.

Existem duas abordagens para decidir se um objeto é persistente ou não:

- O repositório de objetos persistentes mantém raízes persistentes e todo objeto atingido a partir de uma raiz persistente é definido como persistente. Esta estratégia é usada por *Persistent Java*, *Java Data Objects* e por *PerDiS* [Ferreira et al. 2000]. Elas utilizam um coletor de lixo para descartar os objetos que não podem mais ser atingidos a partir das raízes persistentes.
- O repositório de objetos persistentes fornece algumas classes bases para a obtenção de persistência – os objetos persistentes pertencem a suas subclasses. Por exemplo, em *Arjuna* [Parrington et al. 1995], os objetos persistentes são baseados nas classes C++ que fornecem transações e recuperação. Os objetos que não são mais necessários devem ser explicitamente excluídos.

Alguns repositórios de objetos persistentes, por exemplo *PerDiS* e *Khazana* [Carter et al. 1998] permitem que os objetos sejam ativados em caches locais aos usuários, em vez de serem ativados nos servidores. Neste caso, um protocolo de consistência de cache é exigido. Mais detalhes sobre os modelos de consistência podem ser encontrados no *site*

que acompanha o livro, no capítulo da 4^a edição sobre memória compartilhada distribuída [www.cdk5.net/dsm] (em inglês).

Localização de objetos • A Seção 4.3.4 descreveu uma forma de referência de objeto remoto que continha o endereço IP e o número de porta do processo que criou o objeto remoto como uma maneira de garantir a exclusividade. Essa forma de referência de objeto remoto também pode ser usada como endereço de um objeto remoto, desde que ele permaneça no mesmo processo enquanto existir. Contudo, eventualmente, alguns objetos remotos poderão fazer parte de processos diferentes, possivelmente em computadores distintos. Neste caso, uma referência de objeto remoto não pode atuar como endereço. Os clientes que fazem invocações precisam tanto de uma referência de objeto remoto como de endereço para enviar as invocações.

Um serviço de localização ajuda os clientes a encontrarem objetos remotos a partir de suas referências de objeto remoto. Ele utiliza um banco de dados que faz o mapeamento das referências de objeto remoto para suas prováveis localizações correntes – as localizações são prováveis porque um objeto pode ter migrado novamente, desde a última vez que foi encontrado. Por exemplo, o sistema *Clouds* [Dasgupta *et al.* 1991] e o sistema *Emerald* [Jul *et al.* 1988] usavam um esquema de cache/broadcast no qual um membro de um serviço de localização em cada computador continha uma pequena cache de mapeamentos de referência de objeto remoto para localização. Se uma referência de objeto remoto estivesse na cache, a invocação era feita nesse endereço e falhava se o objeto tivesse mudado de lugar. Para localizar um objeto que tinha mudado de lugar, ou cuja localização não fosse a cache, o sistema fazia uma requisição em broadcast. Esse esquema pode ser aprimorado pelo uso de ponteiros de previsão de localização, os quais contêm sugestões sobre a nova localização de um objeto. Outro exemplo é o serviço de resolução de nomes, mencionado na Seção 9.1, usado para transformar o URN de um recurso em seu URL corrente.

5.4.3 Coleta de lixo distribuída

O objetivo de um coletor de lixo distribuído é garantir que, se uma referência local ou remota para um objeto ainda for mantida em algum lugar, em um conjunto de objetos distribuídos, então o próprio objeto continuará a existir; assim que não houver mais referência para ele, esse objeto será coletado e a memória utilizada por ele será recuperada.

Descrevemos o algoritmo de coleta de lixo distribuída Java, que é semelhante ao descrito por Birrell *et al.* [1995]. Ele é baseado na contagem de referência. Quando uma referência de objeto remoto for feita em um processo, um *proxy* será criado e existirá enquanto for necessário. O processo onde o objeto reside (seu servidor) deve ser informado desse *proxy* cliente. Então, posteriormente, quando o *proxy* deixar de existir no cliente, o servidor deverá ser novamente informado. O coletor de lixo distribuído trabalha em cooperação com os coletores de lixo locais, como segue:

- Cada processo servidor mantém um conjunto de nomes dos processos que contêm referências de objeto remoto para cada um de seus objetos remotos; por exemplo, *B.holders* é o conjunto de processos clientes (máquinas virtuais) que têm *proxies* para o objeto *B*. (Na Figura 5.15, esse conjunto incluirá o processo cliente ilustrado.) Esse conjunto pode ser mantido em uma coluna adicional na tabela de objetos remotos.
- Quando um cliente *C* recebe, pela primeira vez, uma referência remota para um objeto remoto particular *B*, ele faz uma invocação à *addRef(B)* no servidor desse objeto remoto e depois cria um *proxy*; o servidor adiciona *C* a *B.holders*.

- Quando o coletor de lixo de um cliente C percebe que um *proxy* para o objeto remoto B não é mais necessário, ele faz uma invocação à $\text{removeRef}(B)$ no servidor correspondente e exclui o *proxy*; o servidor remove C de $B.holders$.
- Quando $B.holders$ estiver vazio, o coletor de lixo local do servidor recuperará o espaço ocupado por B , a não ser que existam clientes ($holders$) locais.

Esse algoritmo é feito para ser executado por meio de comunicação do tipo requisição-resposta, com uma semântica de invocação *no máximo uma vez* entre os módulos de referência remota presentes nos processos cliente e servidor – ele não exige nenhum sincronismo global. Note que invocações normais à coleta de lixo não afetam cada RMI; elas ocorrem quando os *proxies* são criados e excluídos.

Existe a possibilidade de que um cliente possa fazer uma invocação $\text{removeRef}(B)$ praticamente ao mesmo tempo em que outro cliente faz uma invocação $\text{addRef}(B)$. Se a invocação removeRef chegar primeiro e $B.holders$ estiver vazio, o objeto remoto B poderá ser excluído antes da chegada da invocação addRef . Para evitar essa situação, se o conjunto $B.holders$ estiver vazio no momento em que uma referência de objeto remoto for transmitida, uma entrada temporária é adicionada até a chegada de addRef .

O algoritmo de coleta de lixo distribuída Java tolera falhas de comunicação usando a seguinte estratégia: as operações addRef e removeRef são idempotentes. No caso de uma chamada $\text{addRef}(B)$ retornar uma exceção (significando que o método foi executado uma vez ou não foi executado), o cliente não criará o *proxy*, mas fará uma chamada $\text{removeRef}(B)$. O efeito de removeRef estará correto, tenha addRef obtido êxito ou não. O caso em que removeRef falha é tratado por *leasing*, conforme descrito a seguir.

O algoritmo de coleta de lixo distribuída Java pode tolerar a falha de processos clientes. Para conseguir isso, os servidores *cedem* seus objetos para os clientes por um tempo limitado. O período de arrendamento (*leasing*) começa quando o cliente faz uma invocação a addRef no servidor e termina quando o tempo tiver expirado ou quando o cliente fizer uma invocação a removeRef no servidor. As informações armazenadas pelo servidor, relativas a cada *leasing*, contêm o identificador da máquina virtual do cliente e o período de *leasing*. Os clientes são responsáveis por pedir ao servidor para que renove seus *leasings* antes que expirem.

Arrendamento (*leasing*) no Jini • O sistema distribuído Jini inclui uma especificação para *arrendamento* [Arnold *et al.* 1999] que pode ser usada nas situações em que um objeto oferece um recurso para outro objeto como, por exemplo, quando objetos remotos oferecem referências para outros objetos. Os objetos que oferecem tais recursos correm o risco de terem que mantê-los mesmo quando os usuários não estiverem mais interessados ou quando seus programas tiverem terminado. Para evitar a necessidade de protocolos complicados para descobrir se os usuários ainda estão interessados no recurso, eles são oferecidos por tempo limitado. A concessão do uso de um recurso por um período de tempo é chamada de *arrendamento* (*leasing*). O objeto que está oferecendo o recurso o manterá até o momento em que o arrendamento expirar. Os usuários do recurso são responsáveis, quando necessário, por solicitar a renovação do período de arrendamento.

O período de um arrendamento pode ser negociado entre o cedente e o receptor, embora isso não aconteça com os arrendamentos usados na RMI Java. Um objeto que representa um arrendamento implementa a interface *Lease*. Ela contém informações sobre o período de arrendamento e os métodos que permitem sua renovação ou seu cancelamento. O cedente retorna uma instância de *Lease* quando fornece um recurso para outro objeto.

5.5 Estudo de caso: RMI Java

A RMI Java estende o modelo de objeto Java para dar suporte para objetos distribuídos na linguagem Java. Em particular, ela permite que os objetos invoquem métodos em objetos remotos usando a mesma sintaxe das invocações locais. Além disso, a verificação de tipo se aplica igualmente às invocações remotas e às locais. Entretanto, um objeto que faz uma invocação remota sabe que seu destino é remoto, pois precisa lidar com exceções *RemoteException*; e o desenvolvedor de um objeto remoto sabe que ele é remoto porque precisa implementar a interface *Remote*. Embora o modelo de objeto distribuído seja integrado na linguagem Java de maneira natural, a semântica da passagem de parâmetros difere, pois o invocador e o alvo (destino) são remotos entre si.

A programação de aplicativos distribuídos na RMI Java é relativamente simples, pois se trata de um sistema desenvolvido com base em apenas uma linguagem – as interfaces remotas são definidas na linguagem Java. Se for usado um sistema que emprega várias linguagens em seu desenvolvimento, como o CORBA, o programador precisará aprender uma IDL e entender como ela faz o mapeamento em cada linguagem de implementação. Entretanto, mesmo quando é usada apenas uma linguagem de programação, o programador de um objeto remoto deve considerar seu comportamento em um ambiente concorrente.

No restante desta introdução, daremos um exemplo de interface remota e, com base nele, discutiremos a semântica da passagem de parâmetros. Finalmente, discutiremos o *download* de classes e o vinculador. A segunda seção deste estudo de caso discutirá como se faz a construção de programas cliente e servidor com base no exemplo de interface remota. A terceira seção abordará o projeto e a implementação da RMI Java. Para detalhes completos sobre a RMI Java, consulte o exercício dirigido sobre invocação remota [[java.sun.com I](#)].

Neste estudo de caso e no estudo de caso do CORBA, no Capítulo 8, assim como na discussão sobre serviços Web, no Capítulo 9, usamos um *quadro branco compartilhado* como exemplo. Trata-se de um programa distribuído que permite a um grupo de usuários compartilhar uma vista comum de uma superfície de desenho contendo objetos gráficos, como retângulos, linhas e círculos, cada um dos quais desenhado por um dos usuários. O servidor mantém o estado corrente de um desenho, fornecendo uma operação para os clientes informarem-no sobre a figura mais recente que um de seus usuários desenhou e mantendo um registro de todas as figuras que tiver recebido. O servidor também fornece operações que permitem aos clientes recuperarem as figuras mais recentes desenhadas por outros usuários, fazendo uma consulta sequencial no servidor. O servidor tem um número de versão (um valor inteiro), que é incrementado sempre que uma nova figura chega. O servidor fornece operações que permitem aos clientes perguntarem seu número de versão e o número de versão de cada figura, para que eles possam evitar a busca de figuras que já possuem.

Interfaces remotas na RMI Java • As interfaces remotas são definidas pela ampliação de uma interface chamada *Remote*, fornecida no pacote *java.rmi*. Os métodos devem disparar a exceção *RemoteException*, mas exceções específicas do aplicativo também podem ser disparadas. A Figura 5.16 mostra um exemplo de duas interfaces remotas chamadas *Shape* e *ShapeList*. Neste exemplo, *GraphicalObject* é uma classe que contém o estado de um objeto gráfico – por exemplo, seu tipo, sua posição, retângulo envoltório, cor da linha e cor de preenchimento – e fornece operações para acessar e atualizar seu estado. A classe *GraphicalObject* deve implementar a interface *Serializable*. Considere primeiramente a interface *Shape*: o método *getVersion* retorna um valor inteiro, enquanto

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject getAllState() throws RemoteException;
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

1
2

Figura 5.16 Interfaces remotas *Shape* e *ShapeList*.

o método *getAllState* retorna uma instância da classe *GraphicalObject*. Agora, considere a interface *ShapeList*: seu método *newShape* passa como argumento uma instância de *GraphicalObject*, mas retorna como resultado um objeto com uma interface remota (isto é, um objeto remoto). Um ponto importante a se notar é que tanto objetos locais como remotos podem aparecer como argumentos e resultados em uma interface remota. Estes últimos são sempre denotados pelo nome de suas interfaces remotas. No próximo parágrafo, discutiremos como os objetos locais e os objetos remotos são passados como argumentos e resultados.

Passagem de parâmetros e resultados • Na RMI Java, supõe-se que os parâmetros de um método são parâmetros de *entrada* e o resultado de um método é um único parâmetro de *saída*. A Seção 4.3.2 descreveu a serialização Java, que é usada para empacotar argumentos e resultados na RMI Java. Qualquer objeto que seja serializável – isto é, que implemente a interface *Serializable* – pode ser passado como argumento ou ser resultado na RMI Java. Todos os tipos primitivos e objetos remotos são serializáveis. As classes de argumentos e valores de resultado são carregadas por *download* no destino pelo sistema RMI, quando necessário.

Passagem de objetos remotos: quando o tipo de um parâmetro ou valor de resultado é definido como uma interface remota, o argumento ou resultado correspondente é sempre passado como uma referência de objeto remoto. Por exemplo, na Figura 5.16, linha 2, o valor de retorno do método *newShape* é definido como *Shape* – uma interface remota. Quando uma referência de objeto remoto é recebida, ela pode ser usada para fazer chamadas RMI no objeto remoto a que se refere.

Passagem de objetos não remotos: todos os objetos não remotos serializáveis são copiados e passados por valor. Por exemplo, na Figura 5.16 (linhas 2 e 1), o argumento de *newShape* e o valor de retorno de *getAllState* são de tipo *GraphicalObject*, que é serializável e passado por valor. Quando um objeto é passado por valor, um novo objeto é criado no processo destino. Os métodos desse novo objeto podem ser invocados de forma local, possivelmente fazendo seu estado ser diferente do estado do objeto original no processo do remetente.

Assim, em nosso exemplo, o cliente usa o método *newShape* para passar uma instância de *GraphicalObject* para o servidor; o servidor cria um objeto remoto de tipo *Shape*, contendo o estado de *GraphicalObject*, e retorna uma referência de objeto remoto para ele.

void rebind (String name, Remote obj)

Este método é usado por um servidor para registrar o identificador de um objeto remoto pelo nome, conforme mostrado na Figura 5.18, linha 3.

void bind (String name, Remote obj)

Este método pode ser usado como alternativa por um servidor para registrar um objeto remoto pelo nome, mas se o nome já estiver vinculado a uma referência de objeto remoto, será disparada uma exceção.

void unbind (String name, Remote obj)

Este método remove vínculos.

Remote lookup(String name)

Este método é usado pelos clientes para procurar um objeto remoto pelo nome, conforme mostrado na Figura 5.20, linha 1. Retorna uma referência de objeto remoto.

String [] list()

Este método retorna um vetor de objetos *String* contendo os nomes vinculados no registro.

Figura 5.17 A classe *Naming* de *RMIregistry* Java.

Os argumentos e valores de retorno em uma invocação remota são serializados em um fluxo de bytes, usando o método descrito na Seção 4.3.2, com as seguintes modificações:

1. Quando um objeto que implementa a interface *Remote* é serializado, ele é substituído por sua referência de objeto remoto, a qual contém o nome de sua classe (do objeto remoto).
2. Quando um objeto é serializado, suas informações de classe são anotadas com a localização da classe (como um URL), permitindo que a classe seja carregada por *download* pelo destino.

Download de classes • A linguagem Java é projetada para permitir que as classes sejam carregadas, por meio de *download*, de uma máquina virtual para outra. Isso é particularmente relevante para objetos distribuídos que se comunicam por meio de invocação remota. Vimos que os objetos não remotos são passados por valor e os objetos remotos são passados por referência, como argumentos e resultados das RMIs. Se o destino ainda não possuir a classe de um objeto passado por valor, seu código será carregado por *download* automaticamente. Analogamente, se o destino de uma referência de objeto remoto ainda não possuir a classe de um *proxy*, seu código será carregado por *download* automaticamente. Isso tem duas vantagens:

1. Não há necessidade de cada usuário manter o mesmo conjunto de classes em seu ambiente de trabalho.
2. Os programas clientes e servidores podem fazer uso transparente de instâncias de novas classes quando elas forem adicionadas.

Como exemplo, considere o programa de quadro branco e suponha que sua implementação inicial de *GraphicalObject* não admite texto. Então, um cliente com um objeto textual pode implementar uma subclasse de *GraphicalObject*, que lida com texto, e passar uma instância para o servidor como argumento do método *newShape*. Depois disso, outros clientes poderão recuperar a instância usando o método *getAppState*. O código da nova classe será carregado por *download* automaticamente, do primeiro cliente para o servidor e depois, conforme for necessário, para outros clientes.

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            ShapeList aShapeList = new ShapeListServant();
            ShapeList stub =
                (ShapeList) UnicastRemoteObject.exportObject(aShapeList,0);
            Naming.rebind("//bruno.ShapeList", stub );
            System.out.println("ShapeList server ready");
        }catch(Exception e){
            System.out.println("ShapeList server main " + e.getMessage());
        }
    }
}
```

Figura 5.18 Classe ShapeListServer Java com o método *main*.

RMIregistry • O RMIregistry é o vinculador da RMI Java. Uma instância de RMIregistry normalmente deve ser executada em cada computador servidor que contenha objetos remotos. Ele mantém uma tabela mapeando nomes textuais no estilo dos URLs, em referências para objetos remotos contidos nesse computador. Ele é acessado por métodos da classe *Naming*, cujos métodos recebem como argumento um *string* formatado como um URL, da forma:

//*nomeComputador*:*porta*/*nomeObjeto*

onde *nomeComputador* e *porta* se referem à localização do RMIregistry. Se eles forem omitidos, serão presumidos como sendo o computador local e a porta padrão. Sua interface oferece os métodos mostrados na Figura 5.17, na qual as exceções não estão listadas – todos os métodos podem disparar a exceção *RemoteException*.

Usado dessa maneira, os clientes devem direcionar suas consultas de *pesquisa* para computadores específicos. Como alternativa, é possível configurar um serviço de vinculação em nível de sistema. Para se conseguir isso, é necessário executar uma instância do RMIregistry no ambiente de rede e, então, usar a classe *LocateRegistry*, que está em *java.rmi.registry*, para descobrir esse registro. Mais especificamente, essa classe contém um método *getRegistry*, que retorna um objeto de tipo *Registry* representando o serviço de vinculação remoto:

public static Registry getRegistry() throws RemoteException

Depois disso, é necessário fazer uma chamada para *rebind* nesse objeto *Registry* retornado, para estabelecer uma conexão com o registro RMI remoto.

5.5.1 Construção de programas cliente e servidor

Esta seção esboça as etapas necessárias para produzir programas cliente e servidor que utilizam as interfaces *Remote Shape* e *ShapeList*, mostradas na Figura 5.16. O programa servidor é uma versão simplificada do servidor de quadro branco que implementa as duas interfaces *Shape* e *ShapeList*. Descreveremos um programa cliente de consulta sequencial simples e, depois, apresentaremos a técnica de *callback*, que pode ser usada para evi-

```

import java.util.Vector;
public class ShapeListServant implements ShapeList {
    private Vector theList; // contém a lista de elementos Shapes
    private int version;
    public ShapeListServant() {...}
    public Shape newShape(GraphicalObject g) { 1
        version++;
        Shape s = new ShapeServant(g, version);
        theList.addElement(s);
        return s;
    }
    public Vector allShapes() {...}
    public int getVersion() { ... }
}

```

Figura 5.19 A classe *ShapeListServant* Java que implementa a interface *ShapeList*.

tar a necessidade de fazer a consulta sequencial no servidor. Versões completas das classes ilustradas nesta seção estão disponíveis na página do livro em [www.grupoa.com.br] ou em [www.cdk5.net/rmi].

Programa servidor • O programa é um servidor de quadro branco: ele representa cada figura como um objeto remoto instanciado por um servente que implementa a interface *Shape* e contém o estado de um objeto gráfico, assim como seu número de versão; ele representa sua coleção de figuras por meio de outro servente, que implementa a interface *ShapeList* e contém uma coleção de figuras em um *Vector*.

O servidor consiste em um método *main* e uma classe servente para implementar cada uma de suas interfaces remotas. O método *main* da classe servidora está mostrado na Figura 5.18, com as principais etapas contidas nas linhas marcadas de 1 a 4:

- Na linha 1, o servidor cria uma instância de *ShapeListServant*.
- As linhas 2 e 3 utilizam o método *exportObject* (definido em *UnicastRemoteObject*) para tornar esse objeto disponível para o *runtime* da RMI, tornando-o, com isso, disponível para receber invocações. O segundo parâmetro de *exportObject* especifica a porta TCP a ser usada para receber invocações. É uma prática normal configurar isso como zero, significando que uma porta anônima será usada (uma gerada pelo *runtime* da RMI). Usar *UnicastRemoteObject* garante que o objeto resultante durará somente enquanto o processo no qual é criado existir (uma alternativa é torná-lo um objeto *Activable*; isto é, que dura além da instância do servidor).
- Finalmente, a linha 4 vincula o objeto remoto a um nome no *RMIRegistry*. Note que o valor vinculado ao nome é uma referência de objeto remoto e seu tipo é o tipo de sua interface remota – *ShapeList*.

As duas classes serventes são *ShapeListServant*, que implementa a interface *ShapeList*, e *ShapeServant*, que implementa a interface *Shape*. A Figura 5.19 apresenta um esboço da classe *ShapeListServant*.

A implementação dos métodos da interface remota em uma classe servente é extremamente simples, pois ela pode ser feita sem nenhuma preocupação com os detalhes da comunicação. Considere o método de *newShape* na Figura 5.19 (linha 1), que poderia

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
    public static void main(String args[ ]){
        System.setSecurityManager(new RMISecurityManager( ));
        ShapeList aShapeList = null;
        try{
            aShapeList = (ShapeList) Naming.lookup("//bruno.ShapeList");
            Vector sList = aShapeList.allShapes();
            1
            2
        } catch(RemoteException e) {System.out.println(e.getMessage( ));
        }catch(Exception e) {System.out.println("Client: " + e.getMessage( ));}
    }
}
```

Figura 5.20 Cliente Java para *ShapeList*.

ser chamado de método de fábrica, pois ele permite que o cliente solicite a criação de um servente. Ele usa o construtor de *ShapeServant*, o qual cria um novo servente contendo o objeto *GraphicalObject* e o número de versão passados como argumentos. O tipo do valor de retorno de *newShape* é *Shape* – a interface implementada pelo novo servente. Antes de retornar, o método *newShape* adiciona a nova figura em seu vetor, que contém a lista de figuras (linha 2).

O método *main* de um servidor precisa criar um gerenciador de segurança para permitir que a linguagem Java aplique a proteção apropriada ao servidor RMI. É fornecido um gerenciador de segurança padrão, chamado *RMISecurityManager*. Ele protege os recursos locais para garantir que as classes carregadas a partir de *sites* remotos não possam ter qualquer efeito sobre recursos como, por exemplo, arquivos; contudo, ele difere do gerenciador de segurança Java padrão, pois permite que o programa forneça seu próprio carregador de classe e use reflexão. Se um servidor RMI não configurar nenhum gerenciador de segurança, os *proxies* e as classes só poderão ser carregados a partir do caminho de classe local. O objetivo é proteger o programa do código que é carregado por *download* como resultado das invocações a métodos remotos.

Programa cliente • Um cliente simplificado para o servidor *ShapeList* está ilustrado na Figura 5.20. Todo programa cliente precisa ser iniciado usando um vinculador para pesquisar uma referência de objeto remoto. Nossa cliente configura um gerenciador de segurança e, depois, pesquisa uma referência de objeto remoto usando a operação *lookup* do RMIREgistry (linha 1). Tendo obtido uma referência de objeto remoto inicial, o cliente continua, enviando RMIs para esse objeto remoto ou para outros objetos descobertos durante sua execução, de acordo com as necessidades de seu aplicativo. Em nosso exemplo, o cliente invoca o método *allShapes* no objeto remoto (linha 2) e recebe um vetor de referências de objeto remoto para todas as figuras correntemente armazenadas no servidor. Se o cliente implementasse uma tela para o quadro branco, ele usaria o método *getAllState* do servidor na interface *Shape* para recuperar cada um dos objetos gráficos do vetor e os exibir na janela. Sempre que o usuário terminar o desenho de um objeto gráfico, ele invocará o método *newShape* no servidor, passando o novo objeto gráfico como argumento. O cliente manterá um registro do número de versão mais recente no servidor e, de tempos em tempos, invocará *getVersion* no servidor para descobrir se figuras novas foram adicionadas por outros usuários. Se assim for, ele as recuperará e exibirá.

Callbacks • A ideia geral por trás das *callbacks* é que, em vez de os clientes fazerem consultas no servidor para descobrir se ocorreu algum evento, o servidor informa os clientes quando o evento ocorrer. O termo *callback* é usado para referenciar a ação de um servidor ao notificar os clientes sobre um evento. As *callbacks* podem ser implementadas na RMI como segue:

- O cliente cria um objeto remoto que implementa uma interface que contém um método para o servidor chamar. Nos referimos a isso como *objeto de callback*.
- O servidor fornece uma operação que permite aos clientes lhe informar as referências de objeto remoto de seus objetos de *callback*. O servidor as registra em uma lista.
- Quando ocorre um evento, o servidor chama os clientes interessados. Por exemplo, o servidor de quadro branco chamaria seus clientes quando um objeto gráfico fosse adicionado.

O uso de *callbacks* evita a necessidade de um cliente fazer consultas sistemáticas ao servidor para verificar seus objetos de interesse e suas consequentes desvantagens:

- O desempenho do servidor pode ser degradado pelas constantes consultas.
- Os clientes podem não notificar os usuários sobre as atualizações.

Entretanto, as *callbacks* têm seus próprios problemas: primeiro, o servidor precisa ter listas atualizadas dos objetos de *callback* dos clientes. No entanto, os clientes podem não informar o servidor antes de terminarem, deixando o servidor com listas incorretas. A técnica de arrendamento, discutida na Seção 5.4.3, pode ser usada para superar esse problema. O segundo problema associado a *callbacks* é que o servidor precisa fazer uma série de RMIs síncronas nos objetos de *callback* da lista. Consulte o Capítulo 6 para algumas ideias sobre como resolver o segundo problema.

Ilustramos o uso de *callbacks* no contexto do aplicativo de quadro branco. A interface *WhiteboardCallback* poderia ser definida como:

```
public interface WhiteboardCallback implements Remote {
    void callback(int version) throws RemoteException;
};
```

Essa interface é implementada como um objeto remoto pelo cliente, permitindo que o servidor envie ao cliente um número de versão quando um novo objeto é adicionado. Contudo, para o servidor fazer isso, é necessário que o cliente informe o seu objeto de *callback*. Para tornar isso possível, a interface *ShapeList* exige métodos adicionais, como *register* e *deregister*, definidos como segue:

```
int register(WhiteboardCallback callback) throws RemoteException;
void deregister(int callbackId) throws RemoteException;
```

Após o cliente ter obtido uma referência para o objeto remoto com a interface *ShapeList* (por exemplo, na Figura 5.20, linha 1) e criado uma instância de objeto de *callback*, ele utiliza o método *register* de *ShapeList* para registrar no servidor o seu interesse em receber *callbacks*. O método *register* retorna um valor inteiro (uma *callbackId*) que serve como identificador desse registro. Quando o cliente tiver terminado, ele deve chamar *deregister* para informar o servidor que não quer mais as *callbacks*. O servidor é responsável por manter uma lista dos clientes interessados e por notificar todos eles sempre que seu número de versão aumentar.

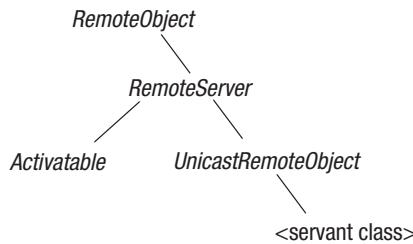


Figura 5.21 Classes que suportam a RMI Java.

5.5.2 Projeto e implementação da RMI Java

O sistema RMI Java original usava todos os componentes mostrados na Figura 5.15. No entanto, no Java 1.2, os recursos de reflexão foram usados para fazer um despachante genérico e para evitar a necessidade de esqueletos. Antes do J2SE 5.0, os *proxies* clientes eram gerados por um compilador, chamado de *rmic*, a partir das classes de servidor compiladas (e não das definições das interfaces remotas). Contudo, essa etapa não é mais necessária, com as versões recentes do J2SE, que contêm suporte para a geração dinâmica de classes *stub* em tempo de execução.

Uso de reflexão • A reflexão é usada para passar informações, nas mensagens de requisição, sobre o método a ser invocado. Isso é obtido por meio da classe *Method* no pacote de reflexão. Cada instância de *Method* representa as características de um método em particular, incluindo sua classe, os tipos de seus argumentos, o valor de retorno e as exceções. A característica mais interessante dessa classe é que uma instância de *Method* pode ser invocada em um objeto de uma classe, por intermédio de seu método *invoke*. O método *invoke* exige dois argumentos: o primeiro especifica o objeto que vai receber a invocação e o segundo é um vetor de *Object* contendo os argumentos. O resultado é retornado como tipo *Object*.

Voltando ao uso da classe *Method* na RMI: o *proxy* precisa empacotar informações sobre um método e seus argumentos na mensagem de requisição. Para o método, ele empacota um objeto da classe *Method*. Ele coloca os argumentos em um vetor de elementos *Object* e, depois, empacota esse vetor. O despachante desempacota o objeto *Method* e seus argumentos no vetor de elementos *Object* da mensagem de *pedido*. Como sempre, a referência de objeto remoto do destino terá sido desempacotada e a referência de objeto local correspondente, obtida do módulo de referência remota. Então, o despachante chama o método *invoke* do objeto *Method*, fornecendo o destino e o vetor de valores de argumento. Quando o método tiver sido executado, o despachante empacotará o resultado ou as exceções na mensagem de *resposta*. Assim, o despachante é genérico – isto é, o mesmo despachante pode ser usado por todas as classes de objeto remoto e nenhum esqueleto é exigido.

Classes Java que suportam RMI • A Figura 5.21 mostra a estrutura de herança das classes que suportam servidores Java RMI. A única classe que o programador necessita conhecer é *UnicastRemoteObject*, que toda classe servente precisa estender. A classe *UnicastRemoteObject* estende uma classe abstrata chamada *RemoteServer*, a qual fornece versões abstratas dos métodos exigidos pelos servidores remotos. *UnicastRemoteObject* foi o primeiro exemplo de *RemoteServer* a ser fornecido. Outro método interessante, chamado *Activatable*, está disponível para fornecer objetos passíveis de ativação. Há também classes para fornecer suporte à replicação de objetos. A classe *RemoteServer* é uma subclasse de *RemoteObject* que tem uma variável de instância contendo a referência de objeto remoto e fornece os seguintes métodos:

equals: este método compara referências de objeto remoto;

toString: este método fornece o conteúdo da referência de objeto remoto como um *String*;

readObject, *writeObject*: estes métodos desserializam/serializam objetos remotos.

Além disso, o operador *instanceOf* pode ser usado para testar objetos remotos.

5.6 Resumo

Este capítulo discutiu três paradigmas da programação distribuída – protocolos de requisição-resposta, chamadas de procedimento remoto e invocação a método remoto. Todos esses paradigmas fornecem mecanismos para entidades distribuídas independentes (processos, objetos, componentes ou serviços) se comunicarem diretamente.

Os programas de requisição-resposta fornecem suporte leve e mínimo para a computação cliente-servidor. Tais protocolos são frequentemente usados em ambientes onde as sobrecargas de comunicação devem ser minimizadas – por exemplo, em sistemas incorporados. Sua função mais comum é dar suporte para RPC ou RMI, conforme discutido a seguir.

A estratégia de chamada de procedimento remoto foi um avanço significativo nos sistemas distribuídos, fornecendo suporte de nível mais alto para os programadores, por estender o conceito de chamada de procedimento para operar em um ambiente de rede. Isso oferece importantes níveis de transparéncia nos sistemas distribuídos. Contudo, devido a suas diferentes características de falha e de desempenho e à possibilidade de acesso concorrente aos servidores, não é necessariamente uma boa ideia fazer as chamadas de procedimento remoto serem exatamente iguais às chamadas locais. As chamadas de procedimento remoto fornecem diversas semânticas de invocação, desde invocações *talvez* até a semântica *no máximo uma vez*.

O modelo de objeto distribuído é uma ampliação do modelo de objeto local usado nas linguagens de programação baseadas em objetos. Os objetos encapsulados formam componentes úteis em um sistema distribuído, pois o encapsulamento os tornam inteiramente responsáveis por gerenciar seus próprios estados, e as invocações a métodos locais podem ser estendidas para invocações remotas. Cada objeto em um sistema distribuído tem uma referência de objeto remoto (um identificador globalmente exclusivo) e uma interface remota que especifica quais de suas operações podem ser invocadas de forma remota.

As implementações de *middleware* da RMI fornecem componentes (incluindo *proxies*, esqueletos e despachantes) que ocultam aos programadores do cliente e do servidor os detalhes do empacotamento, da passagem de mensagem e da localização de objetos remotos. Esses componentes podem ser gerados por um compilador de interface. A RMI Java estende a invocação local em remota usando a mesma sintaxe, mas as interfaces remotas devem ser especificadas estendendo uma interface chamada *Remote* e fazendo cada método disparar uma exceção *RemoteException*. Isso garante que os programadores saibam quando fazem invocações remotas ou implementam objetos remotos, permitindo a eles tratar de erros ou projetar objetos convenientes para acesso concorrente.

Exercícios

- 5.1 Defina uma classe cujas instâncias representem as mensagens de requisição-resposta, conforme ilustrado na Figura 5.4. A classe deve fornecer dois construtores, um para mensagens de requisição e o outro para mensagens de resposta, mostrando como o identificador de requi-

sição é atribuído. Ela também deve fornecer um método para empacotar a si mesma em um vetor de bytes e desempacotar um vetor de bytes em uma instância. *página 188*

- 5.2 Programe cada uma das três operações do protocolo de requisição-resposta da Figura 5.3 usando comunicação UDP, mas sem adicionar quaisquer medidas de tolerância a falhas. Você deve usar as classes que definiu no capítulo anterior para referências de objeto remoto (Exercício 4.13) e acima para mensagens de requisição-resposta (Exercício 5.1). *página 187*
- 5.3 Forneça um esboço da implementação de servidor, mostrando como as operações *getRequest* e *sendReply* são usadas por um servidor que cria uma nova *thread* para executar cada requisição do cliente. Indique como o servidor copiará o *requestId* da mensagem de requisição na mensagem de resposta e como obterá o endereço IP e a porta do cliente. *página 187*
- 5.4 Defina uma nova versão do método *doOperation* que configure um tempo limite para a espera da mensagem de resposta. Após a expiração do tempo limite, ele retransmite a mensagem de requisição *n* vezes. Se ainda não houver nenhuma resposta, ele informará o chamador. *página 188*
- 5.5 Descreva um cenário no qual um cliente poderia receber uma resposta de uma chamada anterior. *página 187*
- 5.6 Descreva as maneiras pelas quais o protocolo de requisição-resposta mascara a heterogeneidade dos sistemas operacionais e das redes de computador. *página 187*
- 5.7 Verifique se as seguintes operações são *idempotentes*:
- i) pressionar o botão “subir” (elevador);
 - ii) escrever dados em um arquivo;
 - iii) anexar dados em um arquivo.
- É uma condição necessária para a idempotência o fato de a operação não estar associada a nenhum estado? *página 190*
- 5.8 Explique as escolhas de projeto relevantes para minimizar o volume de dados de resposta mantidos em um servidor. Compare os requisitos de armazenamento quando os protocolos RR e RRA são usados. *página 191*
- 5.9 Suponha que o protocolo RRA esteja em uso. Por quanto tempo os servidores devem manter dados de resposta não confirmados? Os servidores devem enviar a resposta repetidamente, em uma tentativa de receber uma confirmação? *página 191*
- 5.10 Por que o número de mensagens trocadas em um protocolo poderia ser mais significativo para o desempenho do que o volume total de dados enviados? Projete uma variante do protocolo RRA na qual a confirmação vá “de carona” (*piggyback*) – isto é, seja transmitida na mesma mensagem – na próxima requisição, onde apropriado e, caso contrário, seja enviada como uma mensagem separada. (Dica: use um temporizador extra no cliente.) *página 191*
- 5.11 Uma interface *Election* fornece dois métodos remotos:

vote: este método possui dois parâmetros por meio dos quais o cliente fornece o nome de um candidato (um *string*) e o “número do votante” (um valor inteiro usado para garantir que cada usuário vote apenas uma vez). Os números dos votantes são alocados esparsamente a partir do intervalo de inteiros para torná-los difíceis de adivinhar.

result: este método possui dois parâmetros com os quais o servidor fornece para o cliente o nome de um candidato e o número de votos desse candidato.

Quais dos parâmetros desses dois métodos são de *entrada* e quais são parâmetros de *saiida*? *página 195*

- 5.12 Discuta a semântica de invocação que pode ser obtida quando o protocolo de requisição-resposta é implementado sobre uma conexão TCP/IP, a qual garante que os dados são distribuídos na ordem enviada, sem perda nem duplicação. Leve em conta todas as condições que causam a perda da conexão.

Seção 4.2.4 e página 198

- 5.13 Defina a interface do serviço *Election* na IDL CORBA e na RMI Java. Note que a IDL CORBA fornece o tipo *long* para inteiros de 32 bits. Compare os métodos nas duas linguagens, para especificar argumentos de *entrada* e *saída*.

Figuras 5.8 e 5.16

- 5.14 O serviço *Election* deve garantir que um voto seja registrado quando o usuário achar que depositou o voto.

Discuta o efeito da semântica *talvez* no serviço *Election*.

A semântica *pelo menos uma vez* seria aceitável para o serviço *Election* ou você recomendaria a semântica *no máximo uma vez*?

página 199

- 5.15 Um protocolo de requisição-resposta é implementado em um serviço de comunicação com falhas por omissão para fornecer semântica de invocação *pelo menos uma vez*. No primeiro caso, o desenvolvedor presume um sistema assíncrono distribuído. No segundo caso, o desenvolvedor presume que o tempo máximo para a comunicação e a execução de um método remoto é *T*. De que maneira esta última suposição simplifica a implementação?

página 198

- 5.16 Esboce uma implementação para o serviço *Election* que garanta que seus registros permaneçam consistentes quando ele é acessado simultaneamente por vários clientes.

página 199

- 5.17 Suponha que o serviço *Election* seja implementado em RMI e deva garantir que todos os votos sejam armazenados com segurança, mesmo quando o processo servidor falha. Explique como isso pode ser conseguido no esboço de implementação de sua resposta para o Exercício 5.16.

páginas 213, 214

- 5.18 Mostre como se usa reflexão Java para construir a classe *proxy* cliente para a interface *Election*. Forneça os detalhes da implementação de um dos métodos dessa classe, o qual deve chamar o método *doOperation* com a seguinte assinatura:

`byte[] doOperation (RemoteObjectRef o, Method m, byte[] arguments);`

Dica: uma variável de instância da classe *proxy* deve conter uma referência de objeto remoto (veja o Exercício 4.13).

Figura 5.3, página 224

- 5.19 Mostre como se gera uma classe *proxy* cliente usando uma linguagem como C++, que não suporta reflexão, por exemplo, a partir da definição de interface CORBA dada em sua resposta para o Exercício 5.13. Forneça os detalhes da implementação de um dos métodos dessa classe, o qual deve chamar o método *doOperation* definido na Figura 5.3.

página 211

- 5.20 Explique como se faz para usar reflexão Java para construir um despachante genérico. Forneça o código Java de um despachante cuja assinatura seja:

`public void dispatch(Object target, Method aMethod, byte[] args)`

Os argumentos fornecem o objeto de destino, o método a ser invocado e os argumentos desse método, em um vetor de bytes.

página 224

- 5.21 O Exercício 5.18 exigia que o cliente convertesse argumentos *Object* em um vetor de bytes antes de ativar *doOperation*, e o Exercício 5.20 exigia que o despachante convertesse um vetor de bytes em um vetor de elementos *Object*, antes de invocar o método. Discuta a implementação de uma nova versão de *doOperation* com a seguinte assinatura:

`Object[] doOperation (RemoteObjectRef o, Method m, Object[] arguments);`

que usa as classes *ObjectOutputStream* e *ObjectInputStream* para comunicar as mensagens de requisição-resposta entre cliente e servidor por meio de uma conexão TCP. Como essas alterações afetariam o projeto do despachante? *Seção 4.3.2 e página 224*

- 5.22 Um cliente faz invocações a método remoto a um servidor. O cliente demora 5 milissegundos para computar os argumentos de cada requisição, e o servidor demora 10 milissegundos para processar cada requisição. O tempo de processamento do sistema operacional local para cada operação de envio ou recepção é de 0,5 milissegundos, e o tempo que a rede leva para transmitir cada mensagem de requisição ou resposta é de 3 milissegundos. O empacotamento ou desempacotamento demora 0,5 milissegundos por mensagem.

Calcule o tempo que leva para o cliente gerar e retornar duas requisições:

- (i) se ele tiver só uma *thread*;
- (ii) se ele tiver duas *threads* que podem fazer requisições concorrentes em um único processador.

Você pode ignorar os tempos de troca de contexto. Há necessidade de invocação assíncrona se os processos cliente e servidor forem programados com múltiplas *threads*? *página 213*

- 5.23 Projete uma tabela de objetos remotos que possa suportar coleta de lixo distribuída, assim como fazer a transformação entre referências de objeto local e remota. Dê um exemplo envolvendo vários objetos remotos e *proxies* em diversos *sites* para ilustrar o uso da tabela. Mostre as alterações na tabela quando uma invocação faz um novo *proxy* ser criado. Em seguida, mostre as alterações na tabela quando um dos *proxies* se torna inatingível. *página 215*
- 5.24 Uma versão mais simples do algoritmo de coleta de lixo distribuída, descrito na Seção 5.4.3, apenas invoca *addRef* no *site* onde está um objeto remoto, quando um *proxy* é criado, e *removeRef*, quando um *proxy* é excluído. Esboce todos os efeitos possíveis das falhas de comunicação e de processos no algoritmo. Sugira como superar todos esses efeitos, mas sem usar arrendamentos. *página 215*

6

Comunicação Indireta

- 6.1 Introdução
- 6.2 Comunicação em grupo
- 6.3 Sistemas publicar-assinar
- 6.4 Filas de mensagem
- 6.5 Estratégias de memória compartilhada
- 6.6 Resumo

Este capítulo conclui nosso estudo dos paradigmas de comunicação, examinando a comunicação indireta; ele complementa nossos estudos sobre comunicação entre processos e invocação remota dos Capítulos 4 e 5, respectivamente. A essência da comunicação indireta é se comunicar por meio de um intermediário e, assim, não ter qualquer acoplamento direto entre o remetente e um ou mais destinatários. Os conceitos importantes de desacoplamento em relação ao espaço e ao tempo também são apresentados.

O capítulo examina uma variedade de técnicas de comunicação indireta:

- comunicação em grupo, na qual a comunicação é feita por meio de uma abstração de grupo, sem que o remetente saiba a identidade dos destinatários;
- sistemas publicar-assinar (*publish-subscriber*), uma família de estratégias que compartilham a característica comum de disseminar eventos para vários destinatários por meio de um intermediário;
- sistemas de fila de mensagens, por meio dos quais as mensagens são direcionadas para a conhecida abstração de fila, com os destinatários extraíndo mensagens dessas filas;
- estratégias baseadas em memória compartilhada, incluindo memória compartilhada distribuída e estratégias de espaço de tuplas, as quais apresentam uma abstração de memória compartilhada global para os programadores.

Estudos de caso são usados ao longo de todo o capítulo para ilustrar os principais conceitos apresentados.

6.1 Introdução

Este capítulo conclui nosso estudo sobre paradigmas de comunicação, examinando a comunicação indireta, complementando os estudos da comunicação entre processos e da invocação remota dos Capítulos 4 e 5, respectivamente. A indireção é um conceito fundamental na ciência da computação, e sua ubiquidade e importância são muito bem capturadas na citação a seguir, que surgiu do Projeto Titan da Universidade de Cambridge e é atribuída a Roger Needham, Maurice Wilkes e David Wheeler:

Todos os problemas na ciência da computação podem ser resolvidos por outro nível de indireção.

Em termos de sistemas distribuídos, o conceito de indireção é cada vez mais aplicado aos paradigmas de comunicação:

A *comunicação indireta* é definida como a comunicação entre entidades de um sistema distribuído por meio de um intermediário, sem nenhum acoplamento direto entre o remetente e o destinatário (ou destinatários). A natureza precisa do intermediário varia de uma estratégia para outra, conforme será visto no restante deste capítulo. Além disso, a natureza precisa do acoplamento varia significativamente entre os sistemas e, novamente, isso vai ser salientado no texto a seguir. Observe a forma plural opcional associada a destinatário; isso significa que muitos paradigmas da comunicação indireta suportam explicitamente a comunicação de um para muitos.

As técnicas consideradas nos Capítulos 4 e 5 são baseadas em um acoplamento direto entre um remetente e um destinatário, e isso leva a certa rigidez no sistema, em termos de lidar com alterações. Para ilustrar isso, considere uma interação cliente-servidor simples. Por causa do acoplamento direto, é mais difícil substituir um servidor por outro alternativo que ofereça funcionalidade equivalente. Da mesma forma, se o servidor falha, isso afeta diretamente o cliente, que precisa lidar com a falha explicitamente. Em contraste, a comunicação indireta evita esse acoplamento direto e, assim, herda propriedades interessantes. A literatura se refere a duas propriedades importantes que resultam do uso de um intermediário:

Desacoplamento espacial, no qual o remetente não sabe ou não precisa saber a identidade do destinatário (ou destinatários) e vice-versa. Por causa do desacoplamento espacial, o desenvolvedor de sistema tem muitos graus de liberdade para lidar com alterações: os participantes (remetentes ou destinatários) podem ser substituídos, atualizados, duplicados ou migrados.

Desacoplamento temporal, no qual o remetente e o destinatário (ou destinatários) podem ter tempos de vida independentes. Em outras palavras, o remetente e o destinatário (ou destinatários) não precisam existir ao mesmo tempo para se comunicar. Isso tem vantagens importantes, por exemplo, em ambientes mais voláteis, onde remetentes e destinatários podem ir e vir.

Por esses motivos, a comunicação indireta é frequentemente usada em sistemas distribuídos em que são previstas alterações – por exemplo, em ambientes móveis, onde os usuários podem se conectar e desconectar rapidamente da rede global – e devem ser gerenciadas para fornecer serviços mais confiáveis. A comunicação indireta também é muito usada para disseminação de eventos em sistemas distribuídos, em que os destinatários podem ser desconhecidos e estar propensos à mudança – por exemplo, no gerenciamento da disseminação de eventos (*feeds*) em sistemas financeiros, conforme apresentado no Capítulo 1. A comunicação indireta também é explorada em partes importantes da infraestrutura do Google, conforme veremos no estudo de caso do Capítulo 21.

	<i>Acoplamento temporal</i>	<i>Desacoplamento temporal</i>
<i>Acoplamento espacial</i>	<p><i>Propriedades:</i> comunicação direcionada para determinado destinatário (ou destinatários); o destinatário (ou destinatários) deve existir nesse momento no tempo.</p> <p><i>Exemplos:</i> passagem de mensagens, invocação remota (consulte os Capítulos 4 e 5).</p>	<p><i>Propriedades:</i> comunicação direcionada para determinado destinatário (ou destinatários); o remetente (ou remetentes) e o destinatário (ou destinatários) podem ter tempos de vida independentes.</p> <p><i>Exemplos:</i> consulte o Exercício 6.3.</p>
<i>Desacoplamento espacial</i>	<p><i>Propriedades:</i> o remetente não precisa conhecer a identidade do destinatário (ou destinatários); o destinatário (ou destinatários) deve existir nesse momento no tempo.</p> <p><i>Exemplos:</i> <i>multicast IP</i> (consulte o Capítulo 4).</p>	<p><i>Propriedades:</i> o remetente não precisa conhecer a identidade do destinatário (ou destinatários); o remetente (ou remetentes) e o destinatário (ou destinatários) podem ter tempos de vida independentes.</p> <p><i>Exemplos:</i> a maioria dos paradigmas de comunicação indireta abordados neste capítulo.</p>

Figura 6.1 Acoplamento espacial e temporal em sistemas distribuídos.

A discussão anterior mostra as vantagens associadas à comunicação indireta. A principal desvantagem é que, inevitavelmente, vai haver uma sobrecarga no desempenho, introduzida pelo nível de indireção acrescentado. De fato, a citação anterior sobre indireção é frequentemente acompanhada da seguinte, atribuída a Jim Gray:

Não há um problema de desempenho que não possa ser resolvido pela eliminação de um nível de indireção.

Além disso, os sistemas desenvolvidos usando comunicação indireta podem ser mais difíceis de gerenciar, precisamente por causa da falta de qualquer acoplamento (espacial ou temporal) direto.

Um exame mais detido do desacoplamento espacial e temporal • Pode-se supor que a indireção significa desacoplamento espacial e temporal, mas nem sempre isso acontece. A relação precisa está resumida na Figura 6.1.

A partir dessa tabela, fica claro que a maioria das técnicas consideradas neste livro é acoplada no tempo e no espaço ou, na verdade, desacoplada nas duas dimensões. O quadro superior esquerdo representa os paradigmas de comunicação apresentados nos Capítulos 4 e 5, em que a comunicação é direta, sem nenhum desacoplamento espacial ou temporal. Por exemplo, a passagem de mensagens é direcionada para uma entidade específica e exige que o destinatário esteja presente no momento do envio da mensagem (mas veja no Exercício 6.2 uma dimensão adicional, introduzida pela resolução de nomes DNS). Os diversos paradigmas de invocação remota também são acoplados, tanto no espaço como no tempo. O quadro inferior direito representa os principais paradigmas de comunicação indireta que exibem essas duas propriedades. Um pequeno número de paradigmas de comunicação fica fora dessas duas áreas:

- O *multicast IP*, apresentado no Capítulo 4, é desacoplado no espaço, mas acoplado no tempo. É desacoplado no espaço porque as mensagens são direcionadas para o grupo *multicast* e não para qualquer destinatário específico. É acoplado no tempo, pois todos os destinatários devem existir no momento do envio da mensagem *multicast*. Algumas implementações de comunicação em grupo, e mesmo de sistemas publicar-assinar, também caem nessa categoria (consulte a Seção 6.6). Esse

exemplo ilustra a importância da *persistência* no canal de comunicação para se obter desacoplamento temporal – isto é, o paradigma de comunicação deve armazenar as mensagens para que elas possam ser entregues quando o destinatário (ou destinatários) estiver pronto para receber. O *multicast IP* não suporta esse nível de persistência.

- O caso em que a comunicação é acoplada no espaço, mas desacoplada no tempo, é mais sutil. O acoplamento espacial implica que o remetente conhece a identidade de um destinatário (ou destinatários) específico, mas o desacoplamento temporal implica que o destinatário (ou destinatários) não precisa existir no momento do envio. Os Exercícios 6.3 e 6.4 convidam o leitor a considerar se esse paradigma existe ou poderia ser construído.

Voltando à nossa definição, tratamos como indiretos *todos* os paradigmas que envolvem um intermediário e reconhecemos que o nível preciso de acoplamento vai variar de um sistema para outro. Vamos rever as propriedades dos diferentes paradigmas de comunicação indireta na Seção 6.6, quando tivermos estudado as características exatas da estratégia.

A relação com a comunicação assíncrona • Note que, para se entender essa área, é importante distinguir entre comunicação assíncrona (conforme definimos no Capítulo 4) e desacoplamento temporal. Na comunicação assíncrona, um remetente envia uma mensagem e, então, continua (sem bloquear); assim, não há necessidade de esperar o destinatário para se comunicar. O desacoplamento temporal adiciona a dimensão extra de que o remetente e o destinatário (ou destinatários) podem ter existências independentes; por exemplo, o destinatário pode não existir no momento em que a comunicação é iniciada. Eugster *et al.* também reconhecem a importante distinção entre comunicação assíncrona (desacoplamento em relação ao sincronismo) e desacoplamento temporal [2003].

Muitas das técnicas examinadas neste capítulo são desacopladas no tempo e assíncronas, mas algumas, como as operações *MessageDispatcher* e *RpcDispatcher* no JGroups, discutidas na Seção 6.2.3, oferecem um serviço síncrono sobre comunicação indireta.

O restante do capítulo examina exemplos específicos de comunicação indireta, começando com a comunicação em grupo na Seção 6.2. Em seguida, a Seção 6.3 examina os fundamentos dos sistemas publicar-assinar, e a Seção 6.4 aborda a estratégia contrastante oferecida pelas filas de mensagem. Depois disso, a Seção 6.5 considera estratégias baseadas em abstrações de memória compartilhada, especificamente a memória compartilhada distribuída e as estratégias baseadas em espaço de tuplas.

6.2 Comunicação em grupo

A comunicação em grupo fornece nosso primeiro exemplo de paradigma de comunicação indireta. A *comunicação em grupo* oferece um serviço por meio do qual uma mensagem é enviada para um grupo e, então, entregue a todos os membros do grupo. Nessa ação, o remetente não conhece a identidade dos destinatários. A comunicação em grupo representa uma abstração em relação à comunicação por *multicast* e pode ser implementada sobre *multicast IP* ou sobre uma rede de sobreposição equivalente, melhorando significativamente o gerenciamento de participantes do grupo e a detecção de falhas e fornecendo garantias de confiabilidade e ordenação. Com as garantias reforçadas, a comunicação em grupo está para o *multicast IP* assim como o TCP está para o serviço ponto a ponto em IP.

A comunicação em grupo é um importante bloco de construção para sistemas distribuídos e, particularmente, para sistemas distribuídos confiáveis, com as principais áreas de aplicação incluindo:

- a disseminação confiável de informações para números potencialmente grandes de clientes, incluindo o setor financeiro, no qual as instituições exigem acesso preciso e atualizado a uma ampla variedade de fontes de informação;
- suporte para aplicativos colaborativos, em que, novamente, os eventos precisam ser disseminados para vários usuários a fim de preservar uma visão comum – por exemplo, em jogos multiusuários (discutidos no Capítulo 1);
- suporte para diversas estratégias de tolerância a falhas, incluindo a atualização coerente de dados replicados (conforme discutido em detalhes no Capítulo 18) ou a implementação de servidores (replicados) altamente disponíveis;
- suporte para monitoramento e gerenciamento de sistemas, incluindo, por exemplo, estratégias de balanceamento de carga.

Estudaremos a comunicação em grupo com mais detalhes a seguir, examinando o modelo de programação oferecido e os problemas de implementação associados. Veremos o *toolkit JGroups* como um estudo de caso de serviço de comunicação em grupo.

6.2.1 O modelo de programação

Na comunicação em grupo, o conceito central é o de um *grupo* com *atribuições de membros* associadas por meio das quais os processos podem *ingressar* no grupo ou *sair* dele. Os processos podem enviar uma mensagem para esse grupo e, então, ela é propagada para todos os seus membros, com certas garantias em termos de confiabilidade e ordenação. Assim, a comunicação em grupo implementa comunicação por *multicast*, na qual uma mensagem é enviada para todos os membros do grupo por meio de uma única operação. A comunicação para *todos* os processos no sistema, em contraste com o envio para um subgrupo deles, é conhecida como *broadcast*, enquanto a comunicação para um único processo é conhecida como *unicast*.

A característica fundamental da comunicação em grupo é que um processo executa somente uma operação de *multicast* para enviar uma mensagem para cada processo de um grupo de processos (em Java, essa operação é *aGroup.send(aMessage)*), em vez de executar várias operações de envio para processos individuais.

O uso de uma única operação de *multicast*, em vez de várias operações de envio, significa muito mais do que uma conveniência para o programador: isso permite que a implementação seja eficiente na utilização de largura de banda. É possível enviar a mensagem apenas uma vez por qualquer enlace de comunicação, enviando-a por uma árvore de distribuição, e é possível usar suporte do *hardware* de rede para *multicast* onde isso estiver disponível. A implementação também pode minimizar o tempo total gasto para entregar a mensagem para todos os destinos, em vez de transmiti-la separadamente e em série.

Para perceber essas vantagens, compare a utilização de largura de banda e o tempo de transmissão total gasto ao enviar a mesma mensagem de um computador em Londres para dois computadores na mesma Ethernet em Palo Alto, (a) por meio de dois envios UDP separados e (b) por meio de uma única operação de *multicast IP*. No primeiro caso, duas cópias das mensagens são enviadas independentemente e a segunda é atrasada pela primeira. No último caso, um conjunto de roteadores habilitados para *multicast* encaminha uma única cópia da mensagem de Londres para um roteador na rede local de destino na Califórnia.

Então, esse roteador utiliza *multicast* por *hardware* (fornecido pela Ethernet) para entregar a mensagem nos dois destinos ao mesmo tempo, em vez de enviá-la duas vezes.

O uso de uma única operação de *multicast* também é importante em termos de garantias de entrega. Se um processo executa várias operações de envio independentes para processos individuais, não há como a implementação dar garantias que afetem o grupo de processos como um todo. Se o remetente falha na metade do envio, alguns membros do grupo podem receber a mensagem, enquanto outros, não. Além disso, a ordem relativa de duas mensagens entregues para quaisquer dois membros do grupo é indefinida. Contudo, a comunicação em grupo tem o potencial de oferecer diversas garantias em termos de confiabilidade e ordenação, conforme discutido na Seção 6.2.2, a seguir.

A comunicação em grupo tem sido o tema de muitos projetos de pesquisa, incluindo o V-system [Cheriton e Zwaenepoel 1985], Chorus [Rozier *et al.* 1988], Amoeba [Kaashoek *et al.* 1989, Kaashoek e Tanenbaum 1991], Trans/Total [Melliar-Smith *et al.* 1990], Delta-4 [Powell 1991], Isis [Birman 1993], Horus [van Renesse *et al.* 1996], Totem [Moser *et al.* 1996] e Transis [Dolev e Malki 1996] – e citaremos outros trabalhos dignos de nota no decorrer deste capítulo e, de fato, ao longo de todo o livro (e especialmente nos Capítulos 15 e 18).

Grupos de processos e grupos de objetos • A maioria dos trabalhos sobre serviços de grupo se concentra no conceito de *grupos de processos*; isto é, grupos em que as entidades que se comunicam são processos. Esses serviços são de nível relativamente baixo, pois:

- As mensagens são entregues para processos e nenhum outro suporte para entrega é fornecido.
- Normalmente, as mensagens são vetores de byte não estruturados, sem suporte para empacotamento de tipos de dados complexos (conforme o fornecido, por exemplo, em RPC ou RMI – consulte o Capítulo 5).

Portanto, o nível de serviço fornecido pelos grupos de processos é semelhante ao dos soquetes, conforme discutido no Capítulo 4. Em contraste, os *grupos de objetos* fornecem uma estratégia de nível mais alto para a computação em grupo. Um grupo de objetos é um conjunto de objetos (normalmente instâncias da mesma classe) que processam o mesmo conjunto de invocações concorrentemente, com cada um retornando respostas. Os objetos clientes não precisam estar cientes da replicação. Eles ativam operações em um único objeto local, o qual atua como *proxy* para o grupo. O *proxy* usa um sistema de comunicação em grupo para enviar as invocações para os membros do grupo de objetos. Os parâmetros e resultados do objeto são empacotados como na RMI e as chamadas associadas são entregues automaticamente para os objetos/métodos de destino corretos.

O Electra [Maffeis 1995] é um sistema compatível com CORBA que suporta grupos de objetos. Um grupo do Electra pode fazer interface com qualquer aplicativo compatível com CORBA. O Electra foi construído originalmente sobre o sistema de comunicação em grupo Horus para gerenciar a participação como membro do grupo e para fazer *multicast* das invocações. No “modo transparente”, o *proxy* local retorna a primeira resposta disponível para um objeto cliente. No “modo não transparente”, o objeto cliente pode acessar todas as respostas retornadas pelos membros do grupo. O Electra usa uma extensão da interface Object Request Broker padrão do CORBA, com funções para criar e destruir grupos de objeto e gerenciar sua participação como membro. O Eternal [Moser *et al.* 1998] e o Object Group Service [Guerraoui *et al.* 1998] também fornecem suporte compatível com CORBA para grupos de objeto.

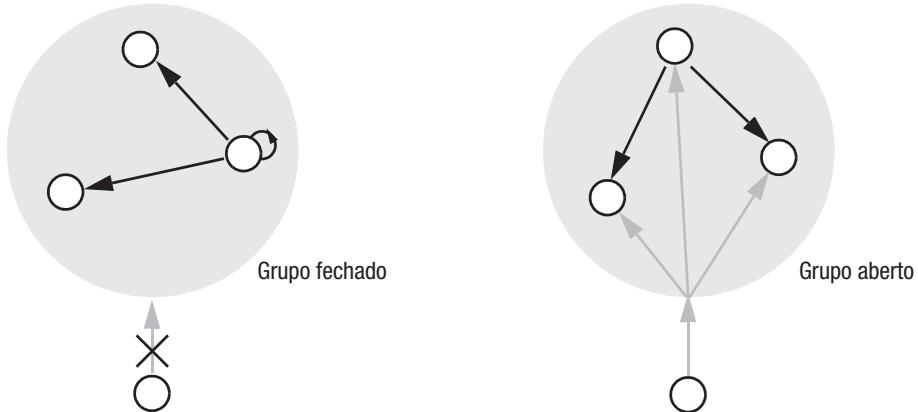


Figura 6.2 Grupos abertos e fechados.

Contudo, apesar do potencial dos grupos de objetos os grupos de processos ainda dominam em termos de utilização. Por exemplo, o popular *toolkit JGroups*, discutido na Seção 6.2.3, é uma estratégia clássica de grupo de processos.

Outras distinções importantes • Uma ampla diversidade de serviços de comunicação em grupo foi desenvolvida e eles variam nas suposições que fazem:

Grupos fechados e abertos: diz-se que um grupo é *fechado* se somente membros do grupo podem enviar mensagem para ele (Figura 6.2). Um processo em um grupo fechado entrega para si mesmo qualquer mensagem que envie para o grupo. Um grupo é *aberto* se processos de fora do grupo podem enviar mensagens para ele. (As categorias “aberto” e “fechado” também se aplicam, com significados similares, às listas de correio.) Os grupos fechados de processos são úteis, por exemplo, para servidores em cooperação enviarem, uns para os outros, mensagens que somente eles devem receber. Os grupos abertos são úteis, por exemplo, para entregar eventos para grupos de processos interessados.

Grupos sobrepostos e não sobrepostos: nos grupos *sobrepostos*, as entidades (processos ou objetos) podem ser membros de vários grupos, e os grupos *não sobrepostos* implicam que a participação como membro não deve se sobrepor (isto é, qualquer processo pertence, no máximo, a um grupo). Note que, em sistemas reais, pode-se esperar que a participação como membro do grupo se sobreponha.

Sistemas síncronos e assíncronos: há o requisito de considerar a comunicação em grupo nos dois ambientes.

Essas distinções podem ter um impacto significativo nos algoritmos de *multicast*. Por exemplo, alguns algoritmos presumem que os grupos são fechados. O mesmo efeito de abertura pode ser obtido com um grupo fechado, escolhendo-se um membro do grupo e enviando a ele uma mensagem (um para um) para que faça *multicast* para seu grupo. Rodrigues *et al.* [1998] discutem o *multicast* para grupos abertos. Os problemas relacionados aos grupos abertos e fechados aparecem no Capítulo 15, quando forem considerados os algoritmos de confiabilidade e ordenação. Esse capítulo também considera o impacto dos grupos sobrepostos e se o sistema é síncrono ou assíncrono em tais protocolos.

6.2.2 Problemas de implementação

Voltemos agora nossa atenção aos problemas de implementação dos serviços de comunicação em grupo, discutindo as propriedades do serviço de *multicast* em termos de confiabilidade e ordenação e também a importante função do gerenciamento da participação como membro do grupo em ambientes dinâmicos, em que os processos podem ingressar e sair ou falhar a qualquer momento.

Confiabilidade e ordenação em multicast • Na comunicação em grupo, todos os membros de um grupo devem receber cópias das mensagens enviadas para o grupo, geralmente com garantias de entrega. As garantias incluem acordo sobre o conjunto de mensagens que todo processo do grupo deve receber e sobre a ordem de entrega para os membros do grupo.

Os sistemas de comunicação em grupo são extremamente sofisticados. Mesmo o *multicast IP*, que oferece garantias de entrega mínimas, exige um grande esforço de engenharia.

Até aqui, discutimos confiabilidade e ordenação em termos bastante gerais. Agora, vamos ver mais detalhes sobre o que essas propriedades significam.

A confiabilidade na comunicação de um para um foi definida na Seção 2.4.2 em termos de duas propriedades: integridade (a mensagem recebida é a mesma que foi enviada e nenhuma mensagem é entregue duas vezes) e validade (qualquer mensagem enviada é entregue). A interpretação de *multicast confiável* complementa essas propriedades, com a *integridade* definida da mesma maneira, em termos de entregar a mensagem corretamente no máximo uma vez, e a *validade* garantindo que uma mensagem enviada vai ser entregue. Para estender a semântica, a fim de abranger a entrega para vários destinatários, uma terceira propriedade é adicionada – trata-se do *acordo*, o qual diz que, se a mensagem é entregue para um processo, então ela é entregue para todos os processos do grupo.

Assim como as garantias de confiabilidade, a comunicação em grupo exige garantias extras em termos da ordem relativa das mensagens entregues para vários destinos. A ordem não é garantida pelas primitivas de comunicação entre processos. Por exemplo, se o *multicast* é implementado por uma série de mensagens de um para um, elas podem ficar sujeitas a atrasos arbitrários. Problemas semelhantes podem ocorrer se for usado *multicast IP*. Para levar isso em conta, os serviços de comunicação em grupo oferecem *multicast ordenado*, com a opção de uma ou mais das propriedades a seguir (também com a possibilidade de soluções híbridas):

Ordem FIFO: a ordem FIFO (first-in-first-out – o primeiro a entrar é o primeiro a sair) – também referida como ordem de origem – preocupa-se em preservar a ordem da perspectiva de um processo remetente, no sentido de que, se um processo enviar uma mensagem antes de outro, então ela vai ser entregue nessa ordem em todos os processos do grupo.

Ordem causal: a ordem causal leva em conta as relações causais entre as mensagens, no sentido de que, se uma mensagem *acontece antes* de outra no sistema distribuído, essa assim chamada relação causal vai ser preservada na entrega das mensagens associadas em todos os processos (consulte o Capítulo 14 para ver uma discussão detalhada sobre o significado de “acontecer antes”).

Ordem total: na ordem total, se uma mensagem for entregue antes de outra em um processo, então a mesma ordem vai ser preservada em todos os processos.

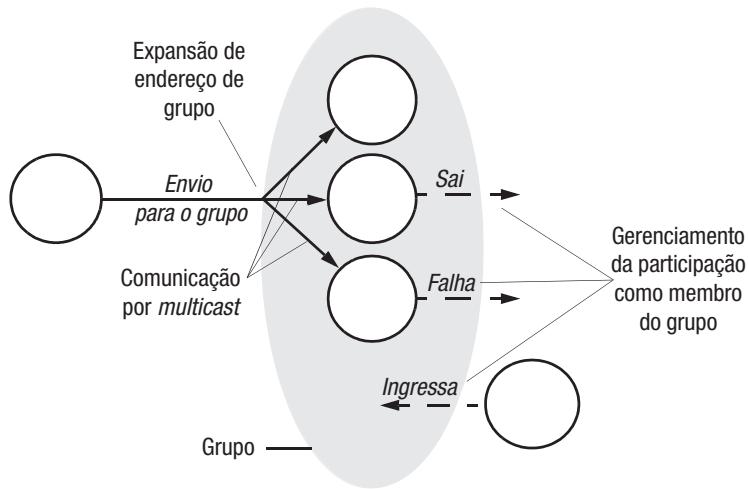


Figura 6.3 O papel do gerenciamento da participação como membro do grupo.

A confiabilidade e a ordenação são exemplos de coordenação e acordo em sistemas distribuídos e, assim, mais considerações sobre isso serão deixadas para o Capítulo 15, que aborda exclusivamente esse assunto. Em particular, o Capítulo 15 fornece definições mais completas de integridade, validade, acordo e diversas propriedades de ordenação, examinando também em detalhes os algoritmos para se fazer *multicast* ordenado e confiável.

Gerenciamento da participação no grupo • Os principais elementos do gerenciamento da comunicação em grupo estão resumidos na Figura 6.3, a qual mostra um grupo aberto. Esse diagrama ilustra o importante papel do gerenciamento da participação no grupo na manutenção de uma *visão* precisa da participação como membro atual, dado que as entidades podem ingressar, sair ou mesmo falhar. Um serviço de participação como membro de um grupo tem quatro tarefas principais:

Fornecer uma interface para mudanças de participação como membro do grupo: o serviço de participação no grupo fornece operações para criar e destruir grupos de processos e para adicionar ou retirar um processo de um grupo. Na maioria dos sistemas, um único processo pode pertencer a vários grupos ao mesmo tempo (grupos sobrepostos, conforme definido anteriormente). Isso vale para o *multicast* IP, por exemplo.

Detecção de falha: esse serviço monitora os membros do grupo não somente para o caso de falha por colapso, mas também para o caso de se tornarem inacessíveis devido a uma falha de comunicação. O detector marca os processos como *Suspeitos* ou *Não suspeitos*. O serviço usa o detector de falha para chegar a uma decisão sobre a participação como membro do grupo: ele exclui um processo como membro se houver suspeita de que ele falhou ou se tornou inacessível.

Notificar os membros sobre mudanças de participação no grupo: o serviço notifica os membros do grupo quando um processo é adicionado ou quando um processo é excluído (devido a uma falha ou quando é deliberadamente retirado do grupo).

Realizar expansão de endereço de grupo: quando um processo envia uma mensagem ao grupo, ele o faz através de um identificador de grupo, em vez de uma lista dos processos no grupo. O serviço de gerenciamento de participação de membros expande o identificador para os membros que pertencem ao grupo. O serviço pode coordenar a entrega mesmo com mudanças na participação de membros no grupo. Isto é, ele pode decidir coerentemente onde vai entregar determinada mensagem, mesmo que a participação como membro tenha mudado durante a entrega.

Note que o *multicast IP* é um caso frágil de serviço de participação como membro do grupo, com algumas dessas propriedades, mas não todas. Ele permite que processos ingressem ou saiam de grupos dinamicamente e realiza a expansão de endereço, de modo que os remetentes só precisam fornecer um endereço de *multicast IP* como destino para uma mensagem. No entanto, o *multicast IP* em si não fornece aos membros do grupo informações sobre a participação como membro atual e a entrega por *multicast* não é coordenada com alterações na participação. Obter essas propriedades é complexo e exige o que é conhecida como *comunicação em grupo com visão síncrona*. Mais considerações sobre esse importante problema serão deixadas para o Capítulo 18, que discute a manutenção de visões de grupo e como fazer comunicação em grupo com visão síncrona no contexto do suporte para replicação em sistemas distribuídos.

Em geral, a necessidade de manter a participação como membro do grupo tem um impacto significativo sobre a utilidade das estratégias baseadas em grupo. Em particular, a comunicação em grupo é mais eficiente em sistemas de pequena escala e estáticos e não funciona tão bem em ambientes de escala maior ou em ambientes com alto grau de volatilidade. Isso pode ser atribuído à necessidade de uma forma de suposição de sincronia. Ganesh *et al.* [2003] apresentam uma estratégia mais probabilística para a participação como membro de grupo, projetada para operar em ambientes mais dinâmicos e de maior escala, usando um protocolo de fofoca (*gossip*) subjacente (consulte a Seção 10.5.3). Os pesquisadores também desenvolveram protocolos de participação como membro de grupo especificamente para redes *ad-hoc* e ambientes móveis [Prakash e Baldoni 1998, Roman *et al.* 2001, Liu *et al.* 2005].

6.2.3 Estudo de caso: o toolkit JGroups

JGroups é um *toolkit* (kit de ferramentas) para comunicação em grupo confiável, escrito em Java. O kit faz parte da linhagem de ferramentas de comunicação em grupo desenvolvidas na Cornell University, complementando os conceitos fundamentais desenvolvidos no ISIS [Birman 1993], Horus [van Renesse *et al.* 1996] e Ensemble [van Renesse *et al.* 1998]. Agora, o kit é mantido e desenvolvido pela comunidade de código-fonte aberto JGroups [www.jgroups.org], a qual faz parte da comunidade de *middleware* JBoss, conforme discutido no Capítulo 8 [www.jboss.org].

O JGroups suporta grupos de processos nos quais os processos podem ingressar e sair de um grupo, enviar uma mensagem para todos os membros do grupo, ou mesmo para um único membro, e receber mensagens do grupo. O *toolkit* suporta uma variedade de garantias de confiabilidade e ordenação (as quais estão discutidas com mais detalhes a seguir) e também oferece um serviço de participação como membro de grupo.

A arquitetura do JGroups aparece na Figura 6.4, a qual mostra os principais componentes da implementação do JGroups:

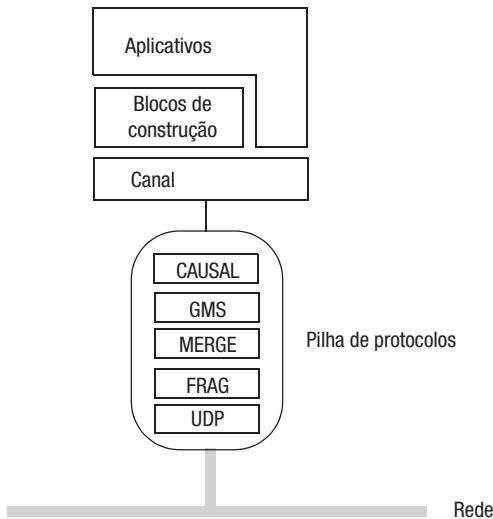


Figura 6.4 A arquitetura do JGroups.

- Os *canais* representam a interface mais primitiva para desenvolvedores de aplicativo, oferecendo as funções básicas de ingresso, saída, envio e recebimento;
 - Os *blocos de construção* oferecem abstrações de nível mais alto, complementando o serviço subjacente oferecido pelos canais;
 - A *pilha de protocolos* fornece o protocolo de comunicação subjacente, construído como uma pilha de camadas de protocolo compostas.
- Examinaros cada um deles a seguir.

Canais • Um processo interage com um grupo por meio de um objeto *canal*, o qual atua como um tratador para um grupo. Quando criado, ele está desconectado, mas uma operação *connect* subsequente vincula esse tratador a um grupo nomeado em particular; se o grupo nomeado não existe, ele é criado implicitamente no momento da primeira conexão. Para sair do grupo, o processo executa a operação *disconnect* correspondente. Também é fornecida uma operação *close* para tornar o canal inutilizável. Note que um canal só pode estar conectado a um grupo por vez; se um processo quer se conectar em dois ou mais grupos, deve criar vários canais. Quando conectado, um processo pode enviar ou receber por meio de um canal. As mensagens são enviadas por *multicast* confiável, com a semântica precisa definida pela pilha de protocolo implantada (conforme discutido a seguir).

Diversas outras operações são definidas nos canais, mais notadamente para retornar informações de gerenciamento associadas ao canal. Por exemplo, *getView* retorna a visão atual definida em termos da lista de membros atual, enquanto *getState* retorna o estado do aplicativo histórico associado ao grupo (isso pode ser usado, por exemplo, por um novo membro do grupo para atualizar-se em relação aos eventos anteriores).

Note que o termo *canal* não deve ser confundido com *publicar-assinar baseada em canal*, conforme apresentado na Seção 6.3.1. No JGroups, canal é sinônimo de instância de um grupo, conforme definido na Seção 6.2.1.

```
import org.jgroups.JChannel;
public class FireAlarmJG {
    public void raise() {
        try {
            JChannel channel = new JChannel();
            channel.connect("AlarmChannel");
            Message msg = new Message(null, null, "Fire!");
            channel.send(msg);
        }
        catch(Exception e) {
        }
    }
}
```

Figura 6.5 Classe Java *FireAlarmJG*.

Ilustramos melhor o uso de canais por meio de um exemplo simples, um serviço por meio do qual um alarme de incêndio inteligente pode enviar uma mensagem “Fire!” (Fogo!) por *multicast* para quaisquer destinatários registrados. O código do alarme de incêndio está mostrado na Figura 6.5.

Quando um alarme é disparado, o primeiro passo é criar uma nova instância de *JChannel* (a classe que representa canais no JGroups) e, então, conectar em um grupo chamado *AlarmChannel*. Se essa for a primeira conexão, então o grupo vai ser criado nesse estágio (improvável neste exemplo, ou o alarme não vai ser muito eficiente). O construtor de uma mensagem recebe três parâmetros, o destino, a origem e a área de dados. Neste caso, o destino é *null*, o que especifica que a mensagem deve ser enviada para todos os membros (se um endereço é especificado, ela é enviada somente para esse endereço). A origem também é *null*; isso não precisa ser fornecido no JGroups, pois vai ser incluído automaticamente. A área de dados é um vetor de bytes não estruturado que é entregue a todos os membros do grupo por meio do método *send*. O código para criar uma nova instância da classe *FireAlarmJG* e, então, disparar um alarme, seria:

```
FireAlarmJG alarm = new FireAlarmJG();
alarm.raise();
```

O código correspondente para a extremidade do destinatário tem estrutura semelhante e aparece na Figura 6.6. Neste caso, contudo, um método *receive* é chamado após a conexão. Esse método recebe apenas um parâmetro, um tempo limite. Se isso for configurado como zero, como neste caso, a mensagem de recebimento vai ser bloqueada até que uma mensagem seja recebida. Note que, no JGroups, as mensagens recebidas são colocadas no *buffer*, e *receive* retorna o elemento superior no *buffer*. Se nenhuma mensagem estiver presente, *receive* bloqueia, esperando a próxima mensagem. Rigorosamente falando, *receive* pode retornar uma variedade de tipos de objeto – por exemplo, notificação de uma alteração na participação como membro ou de uma suspeita de falha de um membro do grupo (daí, a conversão para *Message* acima).

Determinado destinatário deve incluir o código a seguir para esperar um alarme:

```
FireAlarmConsumerJG alarmCall = new FireAlarmConsumerJG();
String msg = alarmCall.await();
System.out.println("Alarm received: " + msg);
```

```

import org.jgroups.JChannel;
public class FireAlarmConsumerJG {
    public String await() {
        try {
            JChannel channel = new JChannel();
            channel.connect("AlarmChannel");
            Message msg = (Message) channel.receive(0);
            return (String) msg.GetObject();
        }
        catch(Exception e) {
            return null;
        }
    }
}

```

Figura 6.6 Classe Java *FireAlarmConsumerJG*.

Blocos de construção • Blocos de construção são abstrações de nível mais alto sobre a classe de canal discutida anteriormente. Os canais têm nível semelhante aos soquetes. Os blocos de construção são análogos aos paradigmas de comunicação mais avançados, discutidos no Capítulo 5, oferecendo suporte para padrões de comunicação que ocorrem comumente (mas, neste caso, destinados à comunicação por *multicast*). Exemplos de blocos de construção no JGroups são:

- *MessageDispatcher*: é o mais intuitivo dos blocos de construção oferecidos no JGroups. Na comunicação em grupo, frequentemente é útil para um remetente enviar uma mensagem para um grupo e, então, esperar por algumas ou por todas as respostas. *MessageDispatcher* suporta isso fornecendo um método *castMessage* que envia uma mensagem para um grupo e bloqueia até que um número especificado de respostas seja recebido (por exemplo, até que um número *n* especificado, a maioria ou todas as mensagens sejam recebidas).
- *RpcDispatcher*: recebe um método específico (junto a parâmetros opcionais e resultados) e, então, chama esse método em todos os objetos associados a um grupo. Assim como acontece com *MessageDispatcher*, o chamador pode bloquear, esperando por algumas ou por todas as respostas.
- *NotificationBus*: é uma implementação de barramento de evento distribuído na qual um evento é qualquer objeto Java que possa ser serializado. Essa classe é frequentemente usada para implementar consistência em caches replicadas.

A pilha de protocolos • O JGroups segue as arquiteturas oferecidas pelo Horus e pelo Ensemble, construindo pilhas de protocolo a partir de camadas de protocolos (inicialmente referidos como microprotocolos na literatura [van Renesse *et al.* 1996, 1998]). Nessa estratégia, um protocolo é uma pilha bidirecional de camadas de protocolo, com cada camada implementando os dois métodos a seguir:

```

public Object up (Event evt);
public Object down (Event evt);

```

Portanto, o processamento de protocolo ocorre pela passagem de eventos para cima e para baixo na pilha. No JGroups, os eventos podem ser mensagens recebidas ou enviadas

ou eventos de gerenciamento relacionados, por exemplo, às mudanças de visão. Cada camada pode realizar processamento arbitrário na mensagem, incluindo modificar seu conteúdo, adicionar um cabeçalho ou mesmo eliminar ou reordenar a mensagem.

Para ilustrarmos melhor o conceito, vamos examinar a pilha de protocolos da Figura 6.4. Isso mostra um protocolo que consiste em cinco camadas:

- A camada referida como UDP é a camada de transporte mais comum no JGroups. Note que, apesar do nome, ela não é totalmente equivalente ao protocolo UDP; em vez disso, a camada utiliza *multicast* IP para enviar mensagens a todos os membros de um grupo e datagramas UDP especificamente para comunicação ponto a ponto. Portanto, essa camada presume que *multicast* IP está disponível. Se não estiver disponível, a camada pode ser configurada para enviar uma série de mensagens *unicast* para os membros, contando com outra camada para descoberta da participação como membro (em particular, uma camada conhecida como *PING*). Para sistemas de escala maior, operando em redes remotas, uma camada TCP pode ser preferível (usando o protocolo TCP para enviar mensagens *unicast* e, novamente, contando com *PING* para descoberta de participação como membro).
- FRAG implementa a transformação de mensagens em pacotes e pode ser configurada em termos do tamanho máximo da mensagem (8.192 bytes, por padrão).
- MERGE é um protocolo que lida com o particionamento inesperado de rede e a subsequente mesclagem de subgrupos após a partição. Uma série de camadas de mesclagem alternativas está disponível, variando desde as simples até as que lidam, por exemplo, com transferência de estado.
- GMS implementa um protocolo de participação como membro de grupo para manter visões coerentes da participação no grupo (consulte o Capítulo 18 para mais detalhes sobre os algoritmos para gerenciamento da participação como membro de grupo).
- CAUSAL implementa ordenação causal, apresentada na Seção 6.2.2 (e discutida com mais detalhes no Capítulo 15).

Uma ampla variedade de outras camadas de protocolo está disponível, incluindo protocolos para ordenação FIFO e total, para descoberta de participação como membro e detecção de falha, para cifragem de mensagens e para implementar estratégias de controle de fluxo (consulte o site do JGroups para ver os detalhes [www.jgroups.org]). Note que, como todas as camadas implementam a mesma interface, elas podem ser combinadas em qualquer ordem, embora muitas das pilhas de protocolo resultantes não façam sentido. Todos os membros de um grupo devem compartilhar a mesma pilha de protocolos.

6.3 Sistemas publicar-assinar

Voltamos agora nossa atenção para a área dos *sistemas publicar-assinar* [Baldoni e Virgillito 2005], também referidos como *sistemas baseados em eventos distribuídos* [Muhl *et al.* 2006]. Essas são as mais amplamente utilizadas de todas as técnicas de comunicação indireta discutidas neste capítulo. O Capítulo 1 já destacou que muitas classes de sistema se preocupam fundamentalmente com a comunicação e com o processamento de eventos (por exemplo, sistemas de negócios financeiros). Mais especificamente, embora muitos sistemas sejam naturalmente mapeados no padrão requisição-resposta ou em um padrão de invocação remota de interação, conforme discutido no Capítulo 5, muitos não

são, sendo mais naturalmente modelados pelo estilo de programação mais descolgado e reativo oferecido pelos eventos.

Um sistema baseado em publicar-assinar é um sistema em que *publicadores* divulgam eventos estruturados para um serviço de evento e *assinantes* expressam interesse em eventos específicos por meio de *assinaturas*, as quais podem ser padrões arbitrários sobre os eventos estruturados. Por exemplo, um assinante poderia expressar interesse em todos os eventos relacionados a este livro sobre sistemas distribuídos, como a disponibilidade de uma nova edição ou atualizações no seu *site*. A tarefa do sistema publicar-assinar é combinar as assinaturas com os eventos publicados e garantir a entrega correta de *notificações de evento*. Determinado evento vai ser entregue possivelmente para muitos assinantes e, assim, o publicar-assinar é, fundamentalmente, um paradigma de comunicação de um para muitos.

Aplicações de sistemas publicar-assinar • Os sistemas publicar-assinar são usados em uma grande variedade de domínios de aplicação, particularmente aqueles relacionados à disseminação de eventos em larga escala. Exemplos incluem:

- sistemas de informação financeira;
- outras áreas com divulgação ao vivo de dados em tempo real (incluindo *feeds RSS*);
- suporte para trabalho cooperativo, em que vários participantes precisam ser informados sobre eventos de interesse compartilhado;
- suporte para computação ubíqua, incluindo o gerenciamento de eventos provenientes de infraestrutura ubíqua (por exemplo, eventos de localização);
- um grande conjunto de aplicativos de monitoramento, incluindo monitoramento de rede na Internet.

O publicar-assinar também é um componente importante da infraestrutura do Google, incluindo, por exemplo, a disseminação de eventos relacionados aos anúncios, como os *ad clicks*, para as partes interessadas (consulte o Capítulo 21).

Para ilustrarmos melhor o conceito, consideramos um sistema simples de sala de negociação como exemplo da classe mais ampla dos sistemas de informação financeira.

Sistema de sala de negociações: considere um sistema simples de sala de negociações cuja tarefa é permitir que negociantes usando computadores vejam as informações mais recentes sobre os preços de mercado da mercadoria que comercializam. O preço de mercado de uma mercadoria nomeada é representado por um objeto associado. As informações chegam à sala de negociação a partir de várias fontes externas diferentes, na forma de atualizações de alguns ou de todos os objetos que representam as mercadorias, e são coletadas por processos que chamamos de *provedores de informação*. Normalmente, os negociantes estão interessados somente em suas mercadorias especializadas. Um sistema de sala de negociações poderia ser modelado por processos com duas tarefas diferentes:

- Um processo provedor de informação recebe continuamente novas informações de negócio de uma única fonte externa. Cada uma das atualizações é considerada um evento. O provedor de informação publica esses eventos no sistema publicar-assinar para entrega a todos os negociantes que tenham expressado interesse na mercadoria correspondente. Haverá um processo provedor de informação separado para cada fonte externa.
- Um processo negociante cria uma assinatura representando cada mercadoria nomeada que o usuário pede para ser exibida. Essa assinatura expressa o interesse em eventos relacionados a determinada mercadoria no provedor de informação relev-

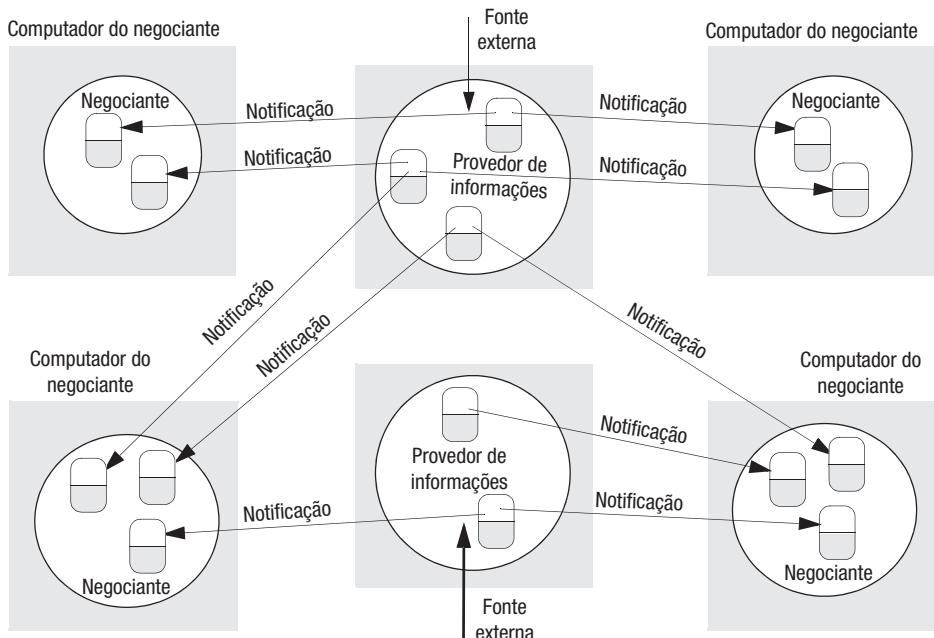


Figura 6.7 Sistema de sala de negociações.

vante. Então, ela recebe todas as informações enviadas em notificações e as mostra para o usuário. A comunicação de notificações está ilustrada na Figura 6.7.

Características dos sistemas publicar-assinar • Os sistemas publicar-assinar têm duas características principais:

Heterogeneidade: quando notificações de evento são usadas como meio de comunicação, pode-se fazer com que componentes de um sistema distribuído que não foram projetados para operação conjunta funcionem juntos. Basta os objetos que geram eventos publicarem os tipos de eventos que oferecem e outros objetos assinarem os padrões de eventos e fornecerem uma interface para receber e lidar com as notificações resultantes. Por exemplo, Bates *et al.* [1996] descrevem como os sistemas publicar-assinar podem ser usados para conectar componentes heterogêneos na Internet. Eles descrevem um sistema no qual os aplicativos podem reconhecer as localizações e atividades dos usuários, como o uso de computadores, impressoras ou livros etiquetados eletronicamente. Eles imaginam seu uso futuro no contexto de uma rede doméstica suportando comandos como: “se as crianças chegarem, ligue o aquecimento central”.

Assíncronos: as notificações são enviadas de forma assíncrona, pelos publicadores que geram eventos, a todos os assinantes que expressaram interesse neles, para evitar que os publicadores precisem estar sincronizados com os assinantes – os publicadores e os assinantes precisam estar desacoplados. O Mushroom [Kindberg *et al.* 1996] é um sistema publicar-assinar baseado em objetos projetado para suportar trabalho colaborativo, no qual a interface exibe objetos que representam usuários e objetos informação, como documentos e blocos de anotações dentro de espaços

de trabalho compartilhados, chamados de *lugares da rede*. O estado de cada lugar é replicado nos computadores dos usuários que estão nesse lugar. Eventos são usados para descrever mudanças nos objetos e no foco de interesse de um usuário. Por exemplo, um evento poderia especificar que um usuário em particular entrou ou saiu de um lugar ou executou determinada ação em um objeto. Cada réplica de qualquer objeto, para o qual tipos de eventos específicos são relevantes, expressa interesse neles por meio de uma assinatura e recebe notificações quando elas ocorrem. No entanto, os assinantes de eventos são desacoplados dos objetos que recebem eventos, pois diferentes usuários estão ativos em diferentes momentos.

Além disso, uma variedade de diferentes *garantias de entrega* pode ser fornecida para notificações – a escolha deve depender dos requisitos das aplicações. Por exemplo, se *multicast IP* for usado para enviar notificações para um grupo de destinatários, o modelo de falha vai estar relacionado àqueles descritos para *multicast IP* na Seção 4.4.1 e não vai garantir que qualquer destinatário específico receba uma mensagem de notificação em particular. Isso é adequado para algumas aplicações – por exemplo, para comunicar o estado mais recente de um jogador em um jogo da Internet –, pois é provável que a próxima atualização passe.

No entanto, outras aplicações têm requisitos mais fortes. Considere o aplicativo de sala de negociações: para sermos justos com os negociantes interessados em uma mercadoria em particular, exigimos que todos os negociantes da mesma mercadoria recebam as mesmas informações. Isso significa que deve ser usado um protocolo de *multicast* confiável.

No sistema Mushroom mencionado anteriormente, as notificações sobre a mudança do estado do objeto são entregues de forma confiável para um servidor, cuja responsabilidade é manter cópias atualizadas dos objetos. Contudo, as notificações também podem ser enviadas para réplicas do objeto nos computadores dos usuários, por meio de *multicast* não confiável; no caso destes últimos perderem notificações, eles podem recuperar o estado de um objeto a partir do servidor. Quando a aplicação exigir, as notificações podem ser ordenadas e enviadas de forma confiável para réplicas do objeto.

Algumas aplicações têm requisitos de tempo real. Isso inclui eventos em um reator nuclear ou em um monitor de pacientes de um hospital. É possível projetar protocolos de *multicast* que forneçam garantias de tempo real, assim como confiabilidade e ordenação em um sistema, e que satisfaçam as propriedades de um sistema distribuído síncrono.

Discutiremos os sistemas publicar-assinar com mais detalhes nas seções a seguir, considerando o modelo de programação oferecido por eles, antes de examinarmos alguns dos principais desafios da implementação, particularmente os relacionados à disseminação em larga escala de eventos na Internet.

6.3.1 O modelo de programação

O modelo de programação em sistemas publicar-assinar é baseado em um pequeno conjunto de operações, mostrado na Figura 6.8. Os publicadores dissemeliam um evento *e* por meio de uma operação *publish(e)* e os assinantes expressam interesse em um conjunto de eventos por meio de assinaturas. Em particular, eles conseguem isso por meio de uma operação *subscribe(f)*, onde *f* se refere a um filtro – isto é, a um padrão definido sobre o conjunto de todos os eventos possíveis. A expressividade dos filtros (*e*, portanto, das assinaturas) é determinada pelo modelo de assinatura, discutido com mais detalhes a seguir. Posteriormente, os assinantes podem revogar esse interesse por meio de uma operação *unsubscribe(f)* correspondente. Quando chegam eventos para um assinante, eles são entregues usando uma operação *notify(e)*.

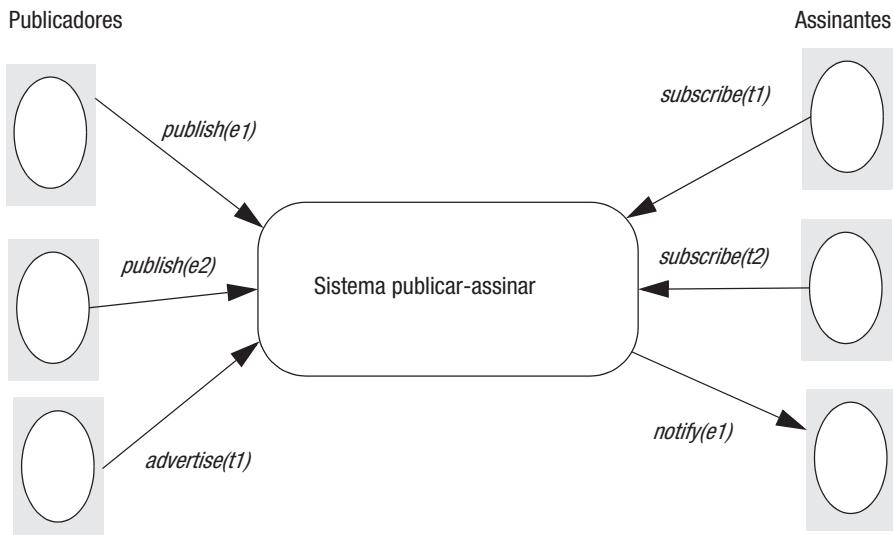


Figura 6.8 O paradigma publicar-assinar.

Alguns sistemas complementam o conjunto de operações anteriores, introduzindo o conceito de anúncios. Com os anúncios, os publicadores têm a opção de declarar a natureza de futuros eventos por meio de uma operação *advertise(f)*. Os anúncios são definidos em termos dos tipos de eventos de interesse (que assumem a mesma forma dos filtros). Em outras palavras, os assinantes declararam seu interesse em termos de assinaturas e, opcionalmente, os publicadores declararam os estilos de eventos que vão gerar por meio de anúncios. Os anúncios podem ser revogados com uma chamada de *unadvertise(f)*.

Conforme mencionado anteriormente, a expressividade dos sistemas publicar-assinar é determinada pelo modelo de assinatura (filtro), com vários esquemas definidos e considerados aqui em ordem crescente de sofisticação:

Baseado em canal: nesta estratégia, os publicadores divulgam eventos para canais nomeados e, então, os assinantes se inscrevem em um deles para receber todos os eventos enviados para esse canal. Esse é um esquema bastante primitivo e o único que define um canal físico; todos os outros esquemas empregam alguma forma de filtragem no conteúdo de um evento, conforme veremos a seguir. Embora simples, esse esquema tem sido usado com sucesso no Event Service do CORBA (consulte o Capítulo 8).

Baseado em tópico (também chamado de *baseado em assunto*): nesta estratégia, supomos que cada notificação é expressa em termos de vários campos, com um deles denotando o tópico. Então, as assinaturas são definidas em termos do tópico de interesse. Esta estratégia é equivalente às estratégias baseadas em canal, sendo que a diferença é que os tópicos são definidos implicitamente no caso dos canais, mas declarados explicitamente como um dos campos nas estratégias baseadas em tópico. A expressividade das estratégias baseadas em tópico também pode ser melhorada pela introdução de uma organização hierárquica de tópicos. Por exemplo, consideraremos um sistema publicar-assinar para este livro. As assinaturas poderiam ser definidas em termos de *comunicação_indireta* ou *comunicação_indireta/publicar-assinar*. Os assinantes que expressem interesse no primeiro receberão todos os eventos relacio-

nados a este capítulo, enquanto, no segundo caso, os assinantes podem expressar interesse no tópico mais específico da publicar-assinar.

Baseado em conteúdo: as estratégias baseadas em conteúdo são uma generalização das baseadas em tópico, permitindo a expressão de assinaturas sobre diversos campos em uma notificação de evento. Mais especificamente, um filtro baseado em conteúdo é uma consulta definida em termos de composições de restrições sobre os valores de atributos de evento. Por exemplo, um assinante poderia expressar interesse nos eventos relacionados ao tópico dos sistemas publicar-assinar, onde o sistema em questão é o “Event Service do CORBA” e onde o autor é “Tim Kindberg” ou “Gordon Blair”. A sofisticação das linguagens de consulta associadas varia de um sistema para outro, mas em geral essa estratégia é significativamente mais expressiva do que as estratégias baseadas em canal ou em tópico, porém com significativos novos desafios em relação à implementação (discutidos a seguir).

Baseado em tipo: essas estratégias estão intrinsecamente ligadas às estratégias baseadas em objeto, em que os objetos têm um tipo específico. Nas estratégias baseadas em tipo, as assinaturas são definidas em termos de tipos de eventos e a combinação é definida em termos de tipos ou subtipos do filtro dado. Esta estratégia pode expressar diversos filtros, desde filtragem mais grossa, baseada em nomes de tipo globais, até consultas mais refinadas, definindo atributos e métodos de determinado objeto. Esses filtros refinados têm expressividade semelhante às estratégias baseadas em conteúdo. As vantagens das estratégias baseadas em tipo é que elas podem ser elegantemente integradas às linguagens de programação e podem verificar a exatidão do tipo das assinaturas, eliminando alguns tipos de erros de assinatura.

Assim como essas categorias clássicas, vários sistemas comerciais são baseados na assinatura direta em *objetos de interesse*. Esses sistemas são semelhantes às estratégias baseadas em tipo no sentido de serem intrinsecamente ligados às estratégias baseadas em objeto, embora difiram por enfocar mudanças de estado dos objetos de interesse e não os predicados associados ao tipo dos objetos. Elas permitem que um objeto reaja a uma mudança ocorrida em outro objeto. As notificações de eventos são assíncronas e determinadas por seus destinatários. Em particular, em aplicativos interativos, as ações executadas pelo usuário em objetos – por exemplo, manipular um botão com o mouse ou digitar texto em uma caixa de texto com o teclado – são vistas como eventos que causam mudanças nos objetos que mantêm o estado do aplicativo. Os objetos responsáveis por exibir uma visão do estado atual são notificados quando o estado muda.

Rosenblum e Wolf [1997] descrevem uma arquitetura geral para esse estilo de sistema publicar-assinar. O principal componente em sua arquitetura é um serviço de evento que mantém um banco de dados de notificações de evento e de interesses dos assinantes. O serviço de evento é notificado sobre os eventos que ocorrem nos objetos de interesse. Os assinantes informam o serviço de evento a respeito dos tipos dos eventos em que estão interessados. Quando ocorre um evento em um objeto de interesse, uma mensagem contendo a notificação é enviada diretamente para os assinantes desse tipo de evento.

A especificação de evento distribuído Jini, descrita por Arnold *et al.* [1999], é um bom exemplo dessa estratégia, sendo que um estudo de caso sobre Jini, junto a mais informações sobre esse estilo de estratégia, pode ser encontrado no *site* que acompanha o livro [www.cdk5.net/rmi] (em inglês). Note, contudo, que o Jini é um exemplo relativamente primitivo de sistema distribuído baseado em evento que permite conectividade direta entre produtores e consumidores de eventos (comprometendo, assim, o desacoplamento temporal e espacial).

Várias estratégias mais experimentais também estão sendo investigadas. Por exemplo, alguns pesquisadores estão considerando a maior expressividade de *contexto* [Frey e Roman 2007, Meier e Cahill 2010]. Contexto e reconhecimento de contexto são conceitos importantes na computação móvel e ubíqua. O contexto está definido no Capítulo 19 como um aspecto das circunstâncias físicas relevantes para o comportamento do sistema. Um exemplo intuitivo de contexto é a localização, e esses sistemas têm o potencial de os usuários assinarem eventos associados a determinado local – por exemplo, quaisquer mensagens de emergência associadas ao prédio onde um usuário esteja localizado. Cilia *et al.* [2004] também introduziram modelos de assinatura *baseados em conceito*, segundo os quais os filtros são expressos em termos da semântica e da sintaxe dos eventos. Mais especificamente, os itens de dados têm um contexto semântico associado que captura o significado desses itens, permitindo a interpretação e a possível transformação em diferentes formatos de dados, tratando, assim, da heterogeneidade.

Para alguns tipos de aplicação, como o sistema de negócios financeiros descrito no Capítulo 1, não basta as assinaturas expressarem consultas sobre eventos individuais. Em vez disso, há necessidade de sistemas mais complexos que possam reconhecer padrões de evento complexos. Por exemplo, o Capítulo 1 apresentou o exemplo da compra e venda de ações com base na observação de sequências temporais de eventos relacionados aos preços das ações, demonstrando a necessidade de *processamento de evento complexo* (ou detecção de evento composto, como também é chamado). O processamento de evento complexo permite a especificação de padrões de eventos conforme ocorrem no ambiente distribuído – por exemplo, “informe-me se os níveis de água subirem pelo menos 20% em pelo menos três lugares no River Eden e se os modelos de simulação também estiverem relatando risco de enchente”. Outro exemplo de padrão de evento surgiu no Capítulo 1, a respeito da detecção de mudanças de preço de ação em determinado período de tempo. Em geral, os padrões podem ser lógicos, temporais ou espaciais. Para mais informações sobre processamento de evento complexo, consulte Muhl *et al.* [2006].

6.3.2 Problemas de implementação

A partir da descrição anterior, a tarefa de um sistema publicar-assinar é clara: garantir que os eventos sejam entregues eficientemente para todos os assinantes que tenham definido filtros que correspondam ao evento. Além disso, pode haver requisitos adicionais em termos de segurança, escalabilidade, tratamento de falha, concorrência e qualidade de serviço. Isso torna a implementação de sistemas publicar-assinar bastante complexa, e essa tem sido uma área de intensa investigação na comunidade de pesquisa. Consideraremos os principais problemas de implementação a seguir, examinando as implementações centralizadas *versus* as distribuídas, antes de considerarmos a arquitetura de sistema global exigida para implementar sistemas publicar-assinar (particularmente, implementações distribuídas de estratégias baseadas em conteúdo). Concluiremos a seção resumindo o espaço de projeto dos sistemas publicar-assinar, com indicadores associados para a literatura.

Implementações centralizadas *versus* distribuídas • Foram identificadas várias arquiteturas para a implementação de sistemas publicar-assinar. A estratégia mais simples é centralizar a implementação em um único nó, com um servidor nesse nó atuando como intermediário de evento. Então, os publicadores divulgam eventos (e, opcionalmente, enviam anúncios) para esse intermediário, e os assinantes enviam assinaturas para ele e recebem notificações em resposta. Assim, a interação com o intermediário é por meio de uma série de mensagens ponto a ponto; isso pode ser implementado usando-se passagem de mensagens ou invocação remota.

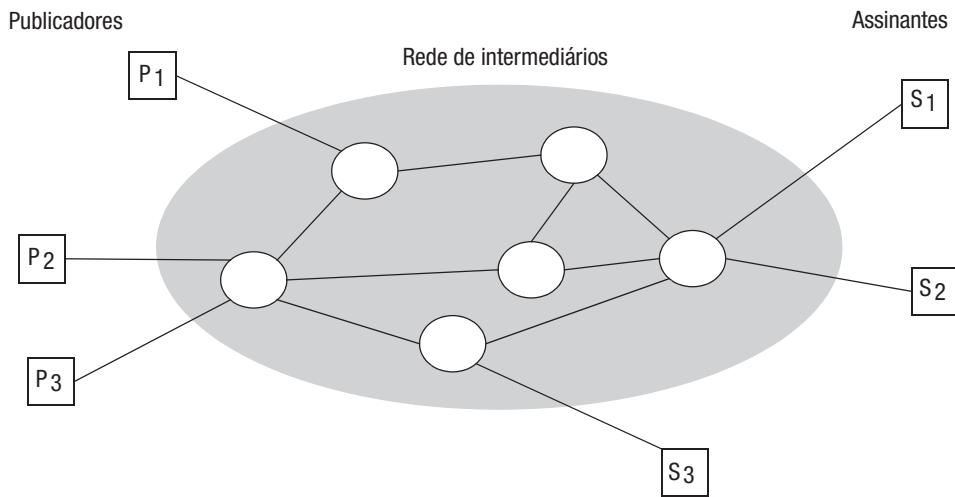


Figura 6.9 A rede de intermediários.

Essa estratégia é simples de ser implementada, mas o projeto não mostra flexibilidade e escalabilidade, pois o intermediário centralizado representa um ponto único de falha e é um gargalo para o desempenho. Consequentemente, também estão disponíveis implementações distribuídas de sistemas publicar-assinar. Nesses esquemas, o intermediário centralizado é substituído por uma *rede de intermediários*, que coopera para oferecer a funcionalidade desejada, conforme ilustrado na Figura 6.9. Essas estratégias têm o potencial de sobreviver à falha do nó e têm-se mostrado capazes de funcionar bem em implantações na Internet.

Levando isso um passo adiante, é possível ter uma implementação totalmente *peer-to-peer* de um sistema publicar-assinar. Essa é uma estratégia de implementação muito popular nos sistemas recentes, e não há distinção entre publicadores, assinantes e intermediários; todos os nós atuam como intermediários, implementando cooperativamente a funcionalidade de roteamento de evento exigida (conforme discutido a seguir).

Arquitetura de sistemas global • Conforme mencionado anteriormente, a implementação de esquemas centralizados é relativamente simples, com o serviço central mantendo um repositório de assinaturas e fazendo a correspondência das notificações de evento com esse conjunto de assinaturas. Da mesma forma, a implementação de esquemas baseados em canal ou em tópico é relativamente simples. Por exemplo, uma implementação distribuída pode ser obtida pelo mapeamento de canais ou tópicos nos grupos associados (conforme definido na Seção 6.2) e, então, usando os recursos de comunicação por *multicast* subjacentes para entregar os eventos às partes interessadas (usando variantes confiáveis e ordenadas, conforme for apropriado). A implementação distribuída das estratégias baseadas em conteúdo (ou, por extrapolação, baseadas em tipo) é mais complexa e merece mais considerações. As diversas escolhas de arquitetura dessas estratégias estão mostradas na Figura 6.10 (adaptada de Baldoni e Virgillito [2005]).

Na camada inferior, os sistemas publicar-assinar fazem uso de diversos serviços de comunicação entre processos, como TCP/IP, *multicast IP* (onde estiver disponível) ou serviços mais especializados, conforme os oferecidos, por exemplo, pelas redes sem fio. A parte principal da arquitetura é fornecida pela camada de roteamento de eventos

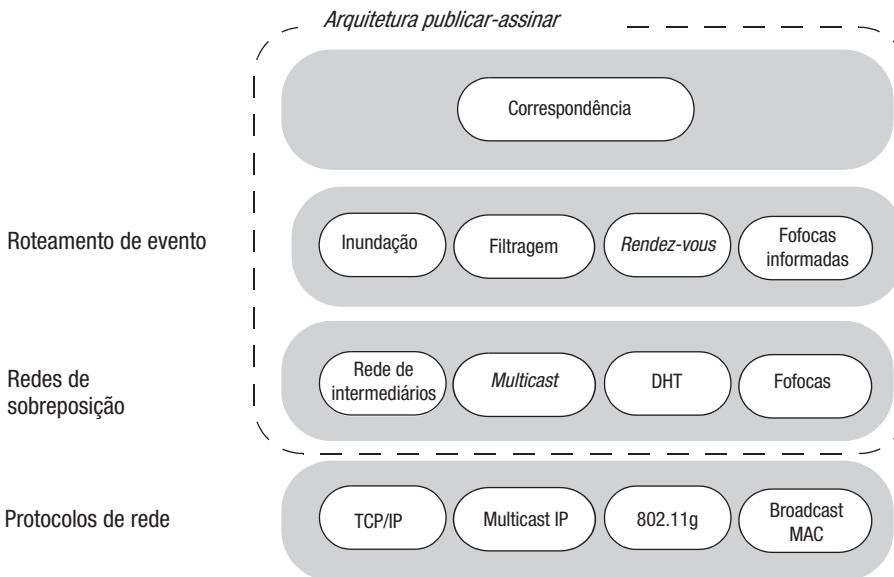


Figura 6.10 A arquitetura de sistemas publicar-assinar.

suportada pela infraestrutura de sobreposição de rede. O roteamento de eventos executa a tarefa de garantir que as notificações de evento sejam roteadas da forma mais eficiente possível para os assinantes apropriados, enquanto a infraestrutura de sobreposição suporta isso configurando redes de intermediários ou estruturas *peer-to-peer* adequadas. Para estratégias baseadas em conteúdo, esse problema é referido como *roteamento baseado em conteúdo* (*CBR, Content-Based Routing*), com o objetivo de explorar as informações do conteúdo para rotear os eventos eficientemente para seus destinos. A camada superior implementa a correspondência – isto é, garante que os eventos correspondam a determinada assinatura. Embora isso possa ser implementado como uma camada separada, frequentemente a correspondência é rebaixada para os mecanismos de roteamento de evento, conforme ficará claro em breve.

Dentro dessa arquitetura global, há uma grande variedade de estratégias de implementação. Examinamos um conjunto selecionado de implementações para ilustrar os princípios gerais por trás do roteamento baseado em conteúdo:

Inundação: a estratégia mais simples é baseada na *inundação*; ou seja, enviar uma notificação de evento para todos os nós da rede e, então, realizar a correspondência apropriada na extremidade assinante. Como alternativa, a inundação pode ser usada para enviar assinaturas de volta para todos os publicadores possíveis, com a correspondência feita na extremidade publicadora e os eventos correspondentes enviados diretamente para os assinantes relevantes usando comunicação ponto a ponto. A inundação pode ser implementada usando-se um recurso de *broadcast* ou *multicast* subjacente. Como alternativa, os intermediários podem ser organizados em um grafo acíclico, no qual cada um encaminha as notificações de evento recebidas para todos os seus vizinhos (efetivamente fornecendo uma sobreposição de *multicast*, conforme discutido na Seção 4.5.1). Essa estratégia tem a vantagem da

```

upon receive publish(event e) from node x           1
    matchlist := match(e, subscriptions)            2
    send notify(e) to matchlist;                  3
    fwplist := match(e, routing);                 4
    send publish(e) to fwplist - x;               5
upon receive subscribe(subscription s) from node x   6
    if x is client then                         7
        add x to subscriptions;                   8
    else add(x, s) to routing;                  9
    send subscribe(s) to neighbours - x;          10

```

Figura 6.11 Roteamento baseado em filtragem.

simplicidade, mas pode resultar em muito tráfego de rede desnecessário. Assim, os esquemas alternativos descritos a seguir tentam otimizar o número de mensagens trocadas por meio da consideração do conteúdo.

Filtragem: um princípio que serve de base para muitas estratégias é aplicar *filtragem* na rede de intermediários. Isso é chamado de *roteamento baseado em filtragem*. Os intermediários encaminham as notificações pela rede somente onde há um caminho para um assinante válido. Isso é obtido pela propagação, pela rede, de informações de assinatura para os publicadores em potencial, seguido do armazenamento do estado associado em cada intermediário. Mais especificamente, cada nó deve manter uma *lista de vizinhos* contendo uma relação de todos os vizinhos conectados à rede de intermediários, uma lista de assinaturas contendo uma relação de todos os assinantes conectados diretamente e servidos por esse nó e uma tabela de roteamento. Fundamentalmente, essa tabela de roteamento mantém uma lista de vizinhos e assinaturas válidas para esse caminho.

Essa estratégia também exige uma implementação de correspondência em cada nó na rede de intermediários: em particular, uma função *match* recebe determinada notificação de evento e uma lista de nós, junto às assinaturas associadas, e retorna um conjunto de nós em que a notificação corresponde à assinatura. O algoritmo específico dessa estratégia de filtragem está na Figura 6.11 (extraído de Baldoni e Virgillito [2005]). Quando um intermediário recebe uma requisição de publicação de determinado nó, deve passar essa notificação para todos os nós conectados em que haja uma assinatura correspondente e decidir para onde vai propagar esse evento pela rede de intermediários. As linhas 2 e 3 atingem o primeiro objetivo, correspondendo o evento com a lista de assinaturas e, então, encaminhando o evento para todos os nós com assinaturas correspondentes (a *matchlist*). Então, as linhas 4 e 5 usam novamente a função *match*, desta vez correspondendo o evento com a tabela de roteamento e encaminhando somente para os caminhos que levam a uma assinatura (a *fwplist*). Os intermediários também precisam lidar com eventos de assinatura recebidos. Se o evento de assinatura é a entrada de um novo assinante diretamente conectado ao intermediário, então essa assinatura deve ser inserida na tabela de assinaturas (linhas 7 e 8). Caso contrário, se for proveniente de outro intermediário, o que recebe o evento identifica um caminho para essa assinatura e, assim, adiciona uma entrada apropriada na tabela de roteamento (linha 9). Nos dois casos, esse evento de assinatura é, então, passado para todos os vizinhos, com exceção do nó originário (linha 10).

```
upon receive publish(event e) from node x at node i
    rvlist := EN(e);
    if i in rvlist then begin
        matchlist <- match(e, subscriptions);
        send notify(e) to matchlist;
    end
    send publish(e) to rvlist - i;
upon receive subscribe(subscription s) from node x at node i
    rvlist := SN(s);
    if i in rvlist then
        add s to subscriptions;
    else
        send subscribe(s) to rvlist - i;
```

Figura 6.12 Roteamento baseado em *rendez-vous*.

Anúncios: a estratégia baseada em filtragem pura descrita anteriormente pode gerar muito tráfego, devido à propagação de assinaturas, com as assinaturas basicamente usando uma estratégia de inundação de volta para todos os possíveis publicadores. Nos sistemas com *anúncios*, essa carga pode ser reduzida por se propagar os anúncios para os assinantes de maneira semelhante (na verdade, simétrica) à propagação de assinaturas. Existem compromissos interessantes entre as duas estratégias e alguns sistemas adotam ambas em conjunto [Carzaniga *et al.* 2001].

Rendez-vous: outra estratégia para controlar a propagação de assinaturas (e obter um balanceamento de carga natural) é o *rendez-vous*. Para se entender essa estratégia, é necessário ver o conjunto de todos os eventos possíveis como um espaço de eventos e repartir a responsabilidade por esse espaço de eventos entre o conjunto de intermediários na rede. Em particular, essa estratégia define nós de *rendez-vous*, que são nós intermediários responsáveis por determinado subconjunto do espaço de eventos. Para se obter isso, um algoritmo de *roteamento baseado em rendez-vous* deve definir duas funções. Primeiro, $SN(s)$ recebe determinada assinatura, s , e retorna um ou mais nós de *rendez-vous*, os quais assumem a responsabilidade por essa assinatura. Cada nó de *rendez-vous* mantém uma lista de assinaturas, como na estratégia de filtragem anterior, e encaminha todos os eventos correspondentes para o conjunto de nós assinantes. Segundo, quando um evento e é publicado, a função $EN(e)$ também retorna um ou mais nós de *rendez-vous*, desta vez responsável por corresponder e às assinaturas do sistema. Note que tanto $SN(s)$ como $EN(e)$ retornam mais de um nó, para o caso de haver preocupação com a confiabilidade. Note também que essa estratégia só funciona se a intersecção de $EN(e)$ e $SN(s)$ não é vazia para determinado e correspondente a s (conhecido como regra de intersecção de mapeamento, conforme definido por Baldoni e Virgillito [2005]). O código correspondente do roteamento baseado em *rendez-vous* está mostrado na Figura 6.12 (novamente extraído de Baldoni e Virgillito [2005]). Desta vez, deixamos a interpretação do algoritmo como exercício para o leitor (veja o Exercício 6.11).

Uma interpretação interessante do roteamento baseado em *rendez-vous* é mapear o espaço de eventos em uma *tabela de hashing distribuída* (*DHT*, *Distributed Hash Table*). As tabelas de *hashing* distribuídas foram apresentadas brevemente na Seção 4.5.1 e serão

Sistemas (leituras recomendadas)	Modelo de assinatura	Modelo de distribuição	Roteamento de eventos
CORBA Event Service (Capítulo 8)	Baseado em canal	Centralizado	–
TIB Rendezvous [Oki et al. 1993]	Baseado em tópicos	Distribuído	Filtragem
Scribe [Castro et al. 2002b]	Baseado em tópicos	Peer-to-peer (DHT)	Rendez-vous
TERA [Baldoni et al. 2007]	Baseado em tópicos	Peer-to-peer	Fofoca informada
Siena [Carzaniga et al. 2001]	Baseado em conteúdo	Distribuído	Filtragem
Gryphon [www.research.ibm.com]	Baseado em conteúdo	Distribuído	Filtragem
Hermes [Pietzuch e Bacon 2002]	Baseado em tópico e em conteúdo	Distribuído	Rendez-vous e filtragem
MEDYM [Cao e Singh 2005]	Baseado em conteúdo	Distribuído	Inundação
Meghdoot [Gupta et al. 2004]	Baseado em conteúdo	Peer-to-peer	Rendez-vous
Structure-less CBR [Baldoni et al. 2005]	Baseado em conteúdo	Peer-to-peer	Fofoca informada

Figura 6.13 Exemplos de sistemas publicar-assinar.

examinadas com mais detalhes no Capítulo 10. Uma tabela de *hashing* distribuída é um estilo de rede de sobreposição que distribui uma tabela de *hashing* por um conjunto de nós em uma rede *peer-to-peer*. A principal observação em relação ao roteamento baseado em *rendez-vous* é que a função de *hashing* pode ser usada para mapear tanto eventos como assinaturas em um nó de *rendez-vous* correspondente, para o gerenciamento dessas assinaturas.

É possível empregar outras estratégias de *middleware peer-to-peer* para servir de base para o roteamento de eventos em sistemas publicar-assinar. Na verdade, essa é uma área de pesquisa muito ativa, com muitas propostas originais e interessantes surgindo, particularmente para sistemas de escala muito grande [Carzaniga et al. 2001]. Uma estratégia específica é adotar um protocolo de fofocas como uma maneira de suportar roteamento de eventos. As estratégias baseadas em *fofocas* são um mecanismo popular para obter *multicast* (incluindo o confiável), conforme discutido na Seção 18.4.1. Elas operam por meio de nós na rede, trocando eventos (ou dados) periódica e probabilisticamente com os nós vizinhos. Por meio dessa estratégia, é possível propagar eventos eficientemente na rede, sem a estrutura imposta pelas outras estratégias. Uma estratégia de fofocas pura é, na verdade, uma estratégia alternativa para implementar inundação, conforme descrito anteriormente. Contudo, é possível levar em conta as informações locais e, em particular, o conteúdo, para se obter o que é chamado de *fofoca informada*. Essas estratégias podem ser particularmente atraentes em ambientes altamente dinâmicos, em que a rotatividade de rede ou nó pode ser alta [Baldoni et al. 2005].

6.3.3 Exemplos de sistemas publicar-assinar

Concluímos esta seção listando alguns exemplos importantes de sistemas publicar-assinar, fornecendo referências de leitura recomendada (consulte a Figura 6.13.). Essa figura também mostra o espaço de projeto para sistemas publicar-assinar, ilustrando como diferentes projetos podem resultar das decisões tomadas sobre os modelos de assinatura e distribuição e, especialmente, sobre a estratégia de roteamento de evento subjacente. Note que o roteamento de evento não é exigido por esquemas centralizados, daí a entrada em branco na tabela.

6.4 Filas de mensagem

As *filas de mensagem* (ou, mais precisamente, filas de mensagem distribuídas) representam uma categoria importante de sistemas de comunicação indireta. Enquanto os grupos e os sistemas publicar-assinar fornecem um estilo de comunicação de um para muitos, as filas de mensagem fornecem um serviço *ponto a ponto*, usando o conceito de fila de mensagens como uma indireção, obtendo, assim, as propriedades desejadas do desacoplamento temporal e espacial. Elas são ponto a ponto no sentido de que o remetente coloca a mensagem em uma fila e isso, então, é removido por um único processo. As filas de mensagem também são referidas como *middleware* orientado a mensagens. Esse é um tipo importante de *middleware* comercial, com as principais implementações incluindo o Websphere MQ, da IBM; o MSMQ, da Microsoft, e o Streams Advanced Queuing (AQ), da Oracle. O principal uso desses produtos é na obtenção de integração de aplicativo empresarial (ou EAI, Enterprise Application Integration) – isto é, a integração entre aplicativos dentro de determinada empresa –, um objetivo atingido pelo fraco acoplamento inerente às filas de mensagem. Eles também são amplamente usados como a base de *sistemas de processamento de transações comerciais*, em razão de seu suporte intrínseco para transações, discutido com mais detalhes na Seção 6.4.1, a seguir.

Examinemos as filas de mensagem com mais detalhes, considerando o modelo de programação oferecido pelos sistemas de enfileiramento de mensagens antes de tratarmos dos problemas de implementação. Então, a seção termina apresentando o serviço JMS (Java Messaging Service) como um exemplo de especificação de *middleware* que suporta filas de mensagem (e também sistemas publicar-assinar).

6.4.1 O modelo de programação

O modelo de programação oferecido pelas filas de mensagem é muito simples. Ele oferece uma estratégia para comunicação em sistemas distribuídos por meio de filas. Em particular, os processos produtores podem *enviar* mensagens para uma fila específica e outros processos (consumidores) podem receber mensagens dessa fila. Geralmente, são suportados três estilos de recepção:

- uma *recepção com bloqueio*, que bloqueará até que uma mensagem apropriada esteja disponível;
- uma *recepção sem bloqueio* (uma operação de consulta), que vai verificar o status da fila e retornar uma mensagem, se estiver disponível; caso contrário, retornará uma indicação de não disponível;
- uma operação de *notificação*, que vai emitir uma notificação de evento quando uma mensagem estiver disponível na fila associada.

Essa estratégia global é ilustrada na Figura 6.14.

Vários processos podem enviar mensagens para a mesma fila e, do mesmo modo, vários destinatários podem remover mensagens de uma fila. A política de enfileiramento normalmente é FIFO (first-in-first-out – primeiro a entrar, primeiro a sair), mas a maioria das implementações de fila de mensagens também suporta o conceito de prioridade, com as mensagens de prioridade mais alta entregues primeiro. Os processos consumidores também podem *selecionar* mensagens da fila com base nas propriedades de uma mensagem. Em mais detalhes, uma mensagem consiste em um *destino* (isto é, um identificador exclusivo designando a fila de destino), *metadados* associados à mensagem, incluindo campos como a prioridade da mensagem e o modo de entrega, e o *corpo* da mensagem. O corpo

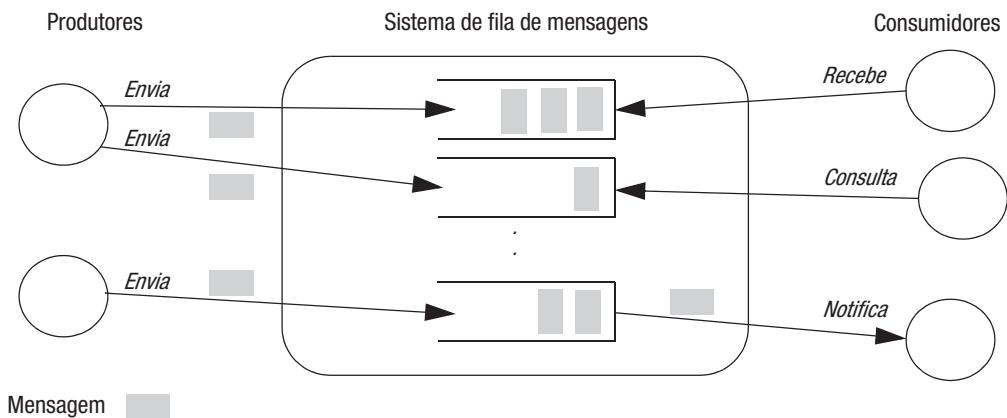


Figura 6.14 O paradigma da fila de mensagens.

normalmente é opaco e não é analisado pelo sistema de fila de mensagens. O conteúdo associado é serializado por meio das estratégias descritas na Seção 4.3; ou seja, tipos de dados empacotados, serialização de objetos ou mensagens XML estruturadas. O tamanho das mensagens pode ser configurado e muito grande – por exemplo, da ordem de 100 MB. Devido ao fato de os corpos de mensagem serem opacos, normalmente a seleção da mensagem é expressa por meio de predicados definidos sobre os metadados.

O AQ da Oracle introduz uma interessante idiossincrasia nessa ideia básica, para obter uma integração melhor com bancos de dados (relacionais); nele, as filas são tabelas do banco de dados e as mensagens são linhas dessas tabelas que podem ser consultadas usando todo o potencial de uma linguagem de consulta de banco de dados.

Uma propriedade fundamental dos sistemas de fila de mensagens é que as mensagens são *persistentes* – isto é, as filas de mensagem armazenam as mensagens indefinidamente (até serem consumidas) e também as armazenam no disco para permitir a *entrega confiável*. Em particular, seguindo a definição de comunicação confiável da Seção 2.4.2, toda mensagem enviada é recebida (validade) e a mensagem recebida é idêntica à enviada, sendo que nenhuma mensagem é entregue duas vezes (integridade). Portanto, os sistemas de fila de mensagens garantem que as mensagens sejam entregues (e apenas uma vez), mas não podem dar informações sobre o momento da entrega.

Os sistemas de passagem de mensagem também podem suportar funcionalidades adicionais:

- A maioria dos sistemas comercialmente disponíveis fornece suporte para que o envio ou recebimento de uma mensagem esteja contido dentro de uma *transação*. O objetivo é garantir que todas as etapas da transação sejam concluídas ou que a transação não tenha nenhum efeito (a propriedade “tudo ou nada”). Isso depende da interface com um serviço de transação externo fornecido pelo ambiente de *middleware*. Uma consideração detalhada sobre as transações é feita no Capítulo 16.
- Vários sistemas também suportam transformação de mensagens, segundo a qual uma modificação arbitrária pode ser realizada em uma mensagem recebida. A aplicação mais comum desse conceito é na transformação entre formatos para lidar com a heterogeneidade nas representações de dados subjacentes. Isso poderia ser tão simples como transformar uma ordem de byte em outra (*big-endian* para *little-*

- *endian*) ou mais complexo, envolvendo, por exemplo, a transformação de uma representação de dados externa em outra (como SOAP para IIOP). Alguns sistemas também permitem aos programadores desenvolver suas próprias transformações específicas da aplicação, em resposta a gatilhos do sistema de enfileiramento de mensagens subjacente. A transformação de mensagens é uma ferramenta importante para lidar com a heterogeneidade de modo geral e para obter a integração de aplicativo empresarial em particular (conforme discutido anteriormente). Note que o termo *intermediário de mensagem* é frequentemente usado para denotar um serviço responsável pela transformação da mensagem.

- Algumas implementações de fila de mensagens também fornecem suporte para *segurança*. Por exemplo, o Websphere MQ fornece suporte para a transmissão confidencial de dados usando SSL (Secure Socket Layer), junto a suporte para autenticação e controle de acesso. Consulte o Capítulo 11.

Como um comentário final sobre a abstração de programação oferecida pelas filas de mensagem, é útil comparar o estilo de programação com outros paradigmas de comunicação. De muitas maneiras, as filas de mensagens são semelhantes aos sistemas de passagem de mensagem considerados no Capítulo 4. A diferença é que, enquanto os sistemas de passagem de mensagem têm filas implícitas associadas a remetentes e destinatários (por exemplo, os *buffers* de mensagem no MPI), os sistemas de enfileiramento de mensagem têm filas explícitas que são entidades terceirizadas, separadas do remetente e do destinatário. É essa importante diferença que torna as filas de mensagem um paradigma de comunicação indireta, com as propriedades fundamentais de desacoplamento espacial e temporal.

6.4.2 Problemas de implementação

O principal problema de implementação para sistemas de enfileiramento de mensagem é a escolha entre implementações centralizadas ou distribuídas do conceito. Algumas implementações são centralizadas, com uma ou mais filas de mensagem controladas por um gerenciador de fila localizado em determinado nó. A vantagem desse esquema é a simplicidade, mas tais gerenciadores podem se tornar componentes bastante pesados e têm o potencial de se tornar um gargalo ou um único ponto de falha. Como resultado, foram propostas implementações distribuídas. Para ilustrarmos as arquiteturas distribuídas, consideremos brevemente a estratégia adotada no Websphere MQ como representativa do estado da arte nessa área.

Estudo de caso: Websphere MQ • O Websphere MQ é um *middleware* desenvolvido pela IBM com base no conceito de filas de mensagem, oferecendo uma indireção entre remetentes e destinatários de mensagens [www.redbooks.ibm.com]. No Websphere MQ, as filas são controladas por *gerenciadores de fila*, os quais as hospedam e comandam, e permitem que os aplicativos as accedam por meio da MQI (Message Queue Interface – interface de fila de mensagem). A MQI é uma interface relativamente simples que permite aos aplicativos executar operações como conectar ou desconectar de uma fila (*MQCONN* e *MQDISC*) ou enviar/receber mensagens para/de uma fila (*MQPUT* e *MQGET*). Vários gerenciadores de fila podem residir em um único servidor físico.

Os aplicativos clientes que accedem a um gerenciador de fila podem residir no mesmo servidor físico. Mais geralmente, contudo, eles ficam em máquinas diferentes e precisam se comunicar com o gerenciador de fila por meio do que é conhecido como *canal cliente*. Os canais clientes adotam o conceito bem conhecido de *proxy*, conforme apre-

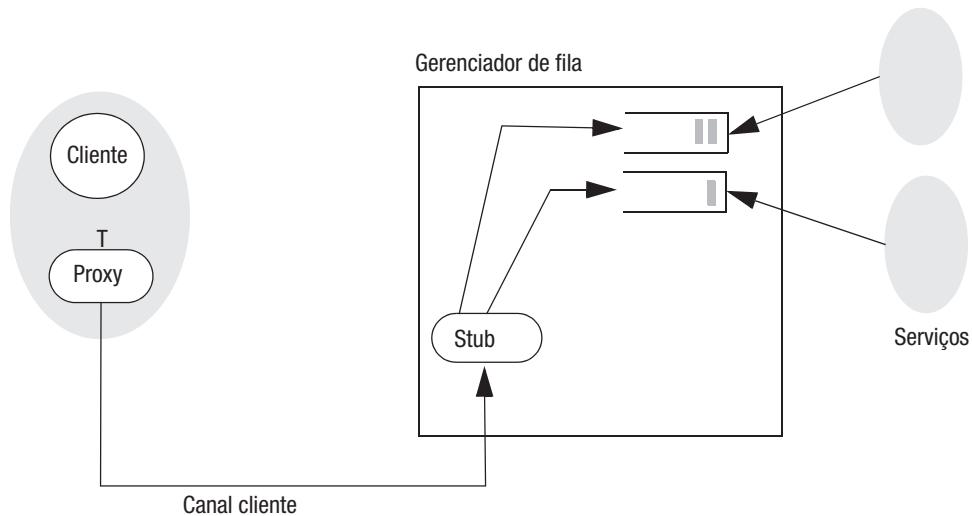


Figura 6.15 Uma topologia de rede simples no Websphere MQ.

sentado nos Capítulos 2 e 5, pelo qual os comandos MQI são executados no *proxy* e, então, enviados de forma transparente para o gerenciador de fila para execução usando RPC. Um exemplo dessa configuração aparece na Figura 6.15. Nessa configuração, um aplicativo cliente está enviando mensagens para um gerenciador de fila remoto, e vários serviços (na mesma máquina do servidor) estão consumindo as mensagens recebidas.

Esse é um uso muito simples do Websphere MQ e, na prática, é mais comum os gerenciadores de fila estarem interligados em uma estrutura confederada, espelhando a estratégia frequentemente adotada nos sistemas publicar-assinar (com redes de intermediários). Para conseguir isso, o MQ introduz o conceito de *canal de mensagem* como uma conexão unidirecional entre dois gerenciadores de fila, o qual, então, é usado para encaminhar mensagens de uma fila para outra de forma assíncrona. Note a terminologia aqui: um canal de mensagem é uma conexão entre dois gerenciadores de fila, enquanto um canal cliente é uma conexão entre um aplicativo cliente e um gerenciador de fila. Um canal de mensagem é gerenciado por um *agente de canal de mensagem* (*MCA*, *Message Channel Agent*) em cada extremidade. Os dois agentes são responsáveis por estabelecer e manter o canal, incluindo uma negociação inicial para concordar com as propriedades do canal (englobando propriedades de segurança). Tabelas de roteamento também são incluídas em cada gerenciador de fila e, junto aos canais, isso permite a criação de topologias arbitrárias.

Essa capacidade de criar topologias personalizadas é fundamental para o Websphere MQ, permitindo que os usuários determinem a topologia correta para seus domínios de aplicação; por exemplo, para fornecer certos requisitos em termos de escalabilidade e desempenho. São fornecidas ferramentas para que os administradores de sistemas criem topologias convenientes e ocultem as complexidades do estabelecimento de canais de mensagem e estratégias de roteamento.

Uma grande variedade de topologias pode ser criada, incluindo árvores, malhas ou uma configuração baseada em barramento. Para ilustrar melhor o conceito de topologias, apresentamos um exemplo de topologia frequentemente usado nas implantações de Websphere MQ, a topologia de rede em estrela (*hub-and-spoke*).

Na topologia de rede em estrela, um gerenciador de fila é designado como *hub*. O *hub* hospeda vários serviços. Os aplicativos clientes não se conectam nesse *hub* diretamente, mas sim por meio de gerenciadores de fila designados como estrelas (*spokes*). As estrelas transmitem mensagens à fila de mensagens do *hub* para processamento pelos vários serviços. As estrelas são posicionadas estratégicamente na rede para suportar diferentes clientes. O *hub* é colocado em algum lugar apropriado na rede, em um nó com recursos suficientes para lidar com o volume de tráfego. A maioria dos aplicativos e serviços fica localizada no *hub*, embora também seja possível ter alguns serviços mais locais nas estrelas.

Essa topologia é muito usada com o WebSphere MQ, particularmente em implantações de larga escala, cobrindo áreas geográficas significativas (e possivelmente ultrapassando limites organizacionais). O segredo da estratégia é ser capaz de conectar uma estrela por meio de uma conexão de alta largura de banda, por exemplo, sobre uma rede local (as estrelas podem até ser colocadas na mesma máquina física dos aplicativos clientes para minimizar a latência).

Lembre-se de que a comunicação entre um aplicativo cliente e um gerenciador de fila usa RPC, enquanto a comunicação interna entre gerenciadores de fila é assíncrona (sem bloqueio). Isso significa que o aplicativo cliente só é bloqueado até que a mensagem seja depositada no gerenciador de fila local (estrela local); a entrega subsequente, possivelmente por redes remotas, é assíncrona, mas com a garantia de ser confiável dada pelo *middleware* Websphere MQ.

Claramente, o inconveniente dessa arquitetura é que o *hub* tem o potencial de ser um gargalo e um único ponto de falha. O Websphere MQ também suporta outros recursos para superar esses problemas, incluindo grupos de gerenciadores de fila que permitem o suporte de várias instâncias do mesmo serviço por diversos gerenciadores de fila, com平衡amento de carga implícito nas diferentes instanciações [www.redbooks.ibm.com].

6.4.3 Estudo de caso: o JMS (Java Messaging Service)

O *JMS* (*Java Messaging Service*) [java.sun.com XI] é uma especificação padronizada para programas Java distribuídos, para comunicação indireta. Mais notadamente, conforme será explicado, a especificação unifica os paradigmas de sistemas publicar-assinar e fila de mensagens pelo menos superficialmente, suportando tópicos e filas como destinos de mensagens alternativos. Uma ampla variedade de implementações da especificação comum está agora disponível, incluindo Joram da OW2, Java Messaging da JBoss, Open MQ da Sun, Apache ActiveMQ e OpenJMS. Outras plataformas, incluindo Websphere MQ, também fornecem uma interface JMS em sua infraestrutura subjacente.

O JMS faz diferenciação entre as seguintes funções:

- Um cliente JMS é um programa ou componente Java que produz ou consome mensagens; um produtor JMS é um programa que cria e produz mensagens; e um consumidor JMS é um programa que recebe e consome mensagens.
- Um provedor JMS é qualquer um dos vários sistemas que implementam a especificação JMS.
- Uma mensagem JMS é um objeto usado para comunicar informações entre clientes JMS (dos produtores para os consumidores).
- Um destino JMS é um objeto que suporta a comunicação indireta no JMS. Ou é um tópico JMS ou uma fila JMS.

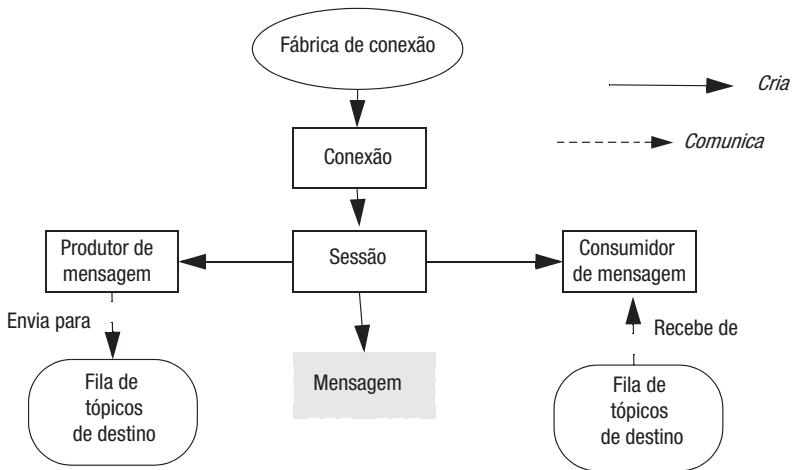


Figura 6.16 O modelo de programação oferecido pelo JMS.

Programação com JMS • O modelo de programação oferecido pela API JMS está ilustrado na Figura 6.16. Para interagir com um provedor JMS, primeiramente é necessário estabelecer uma *conexão* entre um programa cliente e o provedor. Isso é estabelecido por meio de uma *fábrica de conexão* (um serviço responsável por estabelecer conexões com as propriedades exigidas). A conexão resultante é um canal lógico entre o cliente e o provedor; a implementação subjacente pode, por exemplo, ser mapeada em um soquete TCP/IP, se for implementada na Internet. Note que podem ser estabelecidos dois tipos de conexão, *TopicConnection* ou *QueueConnection*, forçando, assim, uma clara separação entre os dois modos de operação dentro de determinadas conexões.

As conexões podem ser usadas para criar uma ou mais *sessões* – uma sessão é uma série de operações envolvendo criação, produção e consumo de mensagens relacionadas a uma tarefa lógica. O objeto sessão resultante também suporta operações para criar *transações*, suportando execução tipo *tudo ou nada* de uma série de operações, conforme discutido na Seção 6.4.1. Há uma clara distinção entre sessões de tópico e sessões de fila, pois uma *TopicConnection* pode suportar uma ou mais sessões de tópico e uma *QueueConnection* pode suportar uma ou mais sessões de fila, mas não é possível misturar estilos de sessão em uma conexão. Assim, os dois estilos de operação são integrados de maneira bastante superficial.

O objeto sessão é fundamental para o funcionamento do JMS, suportando métodos para a criação de mensagens, produtores de mensagem e consumidores de mensagem:

- No JMS, uma *mensagem* consiste em três partes: um *cabeçalho*, um conjunto de *propriedades* e o *corpo* da mensagem. O cabeçalho contém todas as informações necessárias para identificar e rotear a mensagem, incluindo o destino (uma referência para um tópico ou para uma fila), a prioridade da mensagem, a data de expiração, uma identificação de mensagem e um carimbo de tempo. A maioria desses campos é criada pelo sistema subjacente, mas alguns podem ser preenchidos especificamente por meio dos métodos construtores associados. Todas as propriedades são definidas pelo usuário e podem ser usadas para associar outros elementos de

```
import javax.jms.*;
import javax.naming.*;
public class FireAlarmJMS {
    public void raise() {
        try {
            Context ctx = new InitialContext();
            TopicConnectionFactory topicFactory =
                (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
            Topic topic = (Topic)ctx.lookup("Alarms");
            TopicConnection topicConn =
                topicFactory.createTopicConnection();
            TopicSession topicSess = topicConn.createTopicSession(false,
                Session.AUTO_ACKNOWLEDGE);
            TopicPublisher topicPub = topicSess.createPublisher(topic);
            TextMessage msg = topicSess.createTextMessage();
            msg.setText("Fire!");
            topicPub.publish(msg);
        } catch (Exception e) {
        }
    }
}
```

Figura 6.17 Classe Java *FireAlarmJMS*.

metadados específicos do aplicativo a uma mensagem. Por exemplo, na implementação de um sistema com reconhecimento de contexto (conforme discutido no Capítulo 19), as propriedades podem ser usadas para expressar contexto adicional associado à mensagem, incluindo um campo de localização. Assim como na descrição geral de sistemas de fila de mensagens, esse corpo é opaco e não é analisado pelo sistema. No JMS, o corpo pode ser uma mensagem textual, um fluxo de bytes, um objeto Java serializado, um fluxo de valores primitivos Java ou um conjunto de pares nome/valor mais estruturado.

- Um *produtor de mensagem* é um objeto usado para publicar mensagens sob um tópico específico ou para enviar mensagens para uma fila.
- Um *consumidor de mensagem* é um objeto usado para assinar mensagens relativas a determinado tópico ou para receber mensagens de uma fila. O consumidor é mais complicado do que o produtor por dois motivos. Primeiramente, é possível associar filtros aos consumidores de mensagem, especificando-se o que é conhecido como *seletor de mensagem* – um predicado definido sobre os valores presentes no cabeçalho e as partes referentes às propriedades de uma mensagem (não o corpo). Um subconjunto da linguagem de consulta a banco de dados SQL é usado para especificar propriedades. Isso poderia ser usado, por exemplo, para filtrar mensagens de determinado local, no exemplo com reconhecimento de contexto anterior. Segundo, existem dois modos para receber mensagens: o programador pode bloquear usando uma operação *receive* ou pode estabelecer um objeto *receptor de mensagens*, o qual deve fornecer um método *onMessage*, a ser chamado quando uma mensagem adequada for identificada.

Um exemplo simples • Para ilustrar o uso de JMS, voltemos ao nosso exemplo da Seção 6.2.3 – o serviço de alarme de incêndio – e mostremos como isso seria implementado em JMS. Escolhemos o serviço de publicar-assinar baseado em tópicos, pois ele é intrinseca-

```

import javax.jms.*;
import javax.naming.*;

public class FireAlarmConsumerJMS {
    public String await() {
        try {
            Context ctx = new InitialContext();
            TopicConnectionFactory topicFactory =
                (TopicConnectionFactory)ctx.lookup("TopicConnectionFactory");
            Topic topic = (Topic)ctx.lookup("Alarms");
            TopicConnection topicConn =
                topicFactory.createTopicConnection();
            TopicSession topicSess = topicConn.createTopicSession(false,
                Session.AUTO_ACKNOWLEDGE);
            TopicSubscriber topicSub = topicSess.createSubscriber(topic);
            topicSub.start();
            TextMessage msg = (TextMessage)topicSub.receive();
            return msg.getText();
        } catch (Exception e) {
            return null;
        }
    }
}

```

Figura 6.18 Classe Java *FireAlarmConsumerJMS*.

mente um aplicativo de um para muitos, com o alarme produzindo mensagens de alerta destinada a muitos aplicativos consumidores.

O código do objeto alarme de incêndio aparece na Figura 6.17. Isso é mais complicado do que o exemplo de JGroups equivalente, principalmente devido à necessidade de criar uma conexão, a sessão, o publicador e a mensagem, respectivamente, conforme mostrado nas linhas 6 a 11. Isso é relativamente simples, fora os parâmetros de *createTopicSession* que indicam se a sessão deve ser transacional (falso, neste caso) e o modo de reconhecer mensagens (*AUTO_ACKNOWLEDGE*, neste exemplo, o que significa que uma sessão reconhece o recebimento de uma mensagem automaticamente). Há a complexidade adicional associada para localizar a fábrica de conexão e o tópico no ambiente distribuído (a complexidade de se conectar a um canal nomeado no JGroups fica totalmente oculta no método *connect*). Isso é conseguido usando-se JNDI (Java Naming and Directory Interface) nas linhas 2 a 5. Isso foi incluído para sermos completos e presume-se que os leitores possam entender o objetivo dessas linhas de código sem maiores explicações. As linhas 12 e 13 contêm o código crucial para criar uma nova mensagem e, então, publicá-la no tópico apropriado. O código para criar uma nova instância da classe *FireAlarmJMS* e disparar um alarme é:

```

FireAlarmJMS alarm = new FireAlarmJMS()
alarm.raise();

```

O código correspondente no lado do destinatário é bastante semelhante e aparece na Figura 6.18. As linhas 2 a 9 são idênticas e estabelecem a conexão e a sessão exigidas, respectivamente. Desta vez, é criado um objeto de tipo *TopicSubscriber* (linha 10) e o método *start*, na linha 11, inicia essa assinatura, permitindo que mensagens sejam recebidas. Então, *receive* com bloqueio, na linha 12, espera o recebimento de uma mensagem e a linha 13 retorna o conteúdo textual dessa mensagem como um *string*. Essa classe é usada por consumidor, como segue:

```
FireAlarmConsumerJMS alarmCall = new FireAlarmConsumerJMS();
String msg = alarmCall.await();
System.out.println("Alarm received: "+msg);
```

No todo, esse estudo de caso ilustrou como sistemas publicar-assinar e filas de mensagem podem ser suportados por uma única solução de *middleware* (neste caso, JMS), oferecendo ao programador a escolha de variantes de um para muitos ou ponto a ponto da comunicação indireta, respectivamente.

6.5 Estratégias de memória compartilhada

Nesta seção, examinamos os paradigmas de comunicação indireta que oferecem uma abstração de memória compartilhada. Veremos brevemente as técnicas de memória compartilhada distribuída que foram desenvolvidas principalmente para a computação paralela, antes de passarmos para a comunicação via espaço de tuplas, uma estratégia que permite aos programadores ler e gravar tuplas a partir de um espaço de tuplas compartilhado. Enquanto a memória compartilhada distribuída opera em nível de leitura e gravação de bytes, os espaços de tuplas oferecem uma perspectiva de nível mais alto, na forma de dados semiestruturados. Além disso, enquanto a memória compartilhada distribuída é acessada pelo endereço, os espaços de tuplas são *associativos*, oferecendo uma forma de memória endereçável pelo conteúdo [Gelernter 1985].

O Capítulo 18 da quarta edição deste livro forneceu uma abordagem aprofundada sobre memória compartilhada distribuída, incluindo modelos de consistência e vários estudos de caso. Esse capítulo pode ser encontrado no *site* que acompanha o livro [www.cdk5.net/dsm] (em inglês).

6.5.1 Memória compartilhada distribuída

A memória compartilhada distribuída (DSM, distributed shared memory) é uma abstração usada para compartilhar dados entre computadores que não compartilham memória física. Os processos acessam a DSM por meio de leituras e atualizações no que parece ser memória normal dentro de seus espaços de endereçamento. Contudo, um suporte de execução *runtime* subjacente garante, de forma transparente, que os processos sendo executados em diferentes computadores observem as atualizações feitas pelos outros. É como se os processos acessassem uma única memória compartilhada, mas na verdade a memória física é distribuída (veja a Figura 6.19).

A principal vantagem da DSM é que ela dispensa o programador das preocupações com passagem de mensagens ao escrever aplicativos que, de outra forma, talvez tivessem que utilizá-la. A DSM é, principalmente, uma ferramenta para aplicativos paralelos ou para qualquer aplicativo (ou grupo de aplicativos) distribuído no qual itens de dados compartilhados individuais podem ser acessados diretamente. Em geral, a DSM é menos adequada em sistemas cliente-servidor, em que os clientes normalmente veem os recursos mantidos no servidor como dados abstratos e os acessam por requisição (por motivos de modularidade e proteção).

A passagem de mensagens não pode ser completamente evitada em um sistema distribuído: na ausência de memória fisicamente compartilhada, o suporte para tempo de execução da DSM precisa enviar atualizações entre os computadores em mensagens. Os sistemas de DSM gerenciam dados replicados: cada computador tem uma cópia lo-

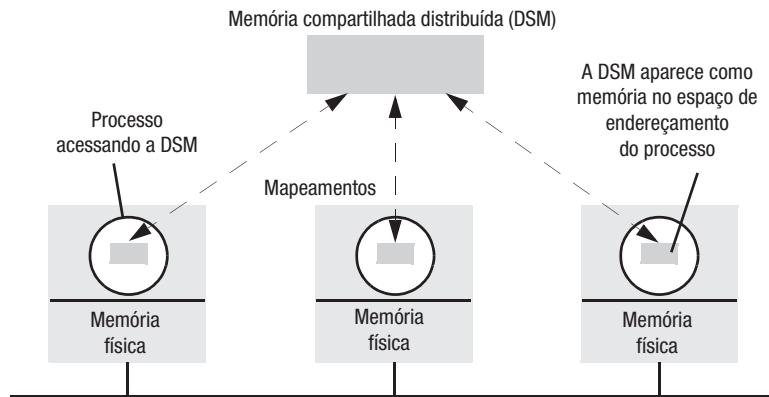


Figura 6.19 A abstração de memória compartilhada distribuída (DSM).

cal dos itens de dados armazenados na DSM e acessados recentemente, para acelerar o acesso. Os problemas de implementação da DSM estão relacionados aos discutidos no Capítulo 18, assim como aos do uso de cache para arquivos compartilhados, apresentados no Capítulo 12.

Um dos primeiros exemplos notáveis de implementação de DSM foi o sistema de arquivos Apollo Domain [Leach *et al.* 1983], no qual os processos contidos em diferentes estações de trabalho compartilham arquivos mapeando-os simultaneamente em seus espaços de endereçamento. Esse exemplo mostra que a memória compartilhada distribuída pode ser persistente, isto é, pode durar mais do que a execução de qualquer processo ou grupo de processos que a acesse e ser compartilhada por diferentes grupos de processos no decorrer do tempo.

A importância da DSM aumentou junto com o desenvolvimento dos multiprocessadores de memória compartilhada (discutidos na Seção 7.3). Muitas pesquisas foram feitas para investigar algoritmos convenientes para a computação paralela nesses multiprocessadores. Em nível de arquitetura de *hardware*, os desenvolvimentos incluem estratégias de uso de cache e interconexões processador-memória rápidas, destinadas a maximizar o número de processadores que podem ser mantidos, ao passo que se obtém baixa latência de acesso à memória e alta taxa de transferência [Dubois *et al.* 1988]. Onde os processos estão conectados a módulos de memória por meio de um barramento comum, o limite prático é da ordem de 10 processadores antes que o desempenho degrade drasticamente devido à disputa pelo barramento. Os processadores que compartilham memória normalmente são construídos em grupos de quatro, compartilhando o módulo de memória por meio de um barramento em uma única placa de circuito impresso. Multiprocessadores com até 64 processadores no total são construídos a partir dessas placas em uma arquitetura NUMA (*Non-Uniform Memory Access – acesso não uniforme à memória*). Essa é uma arquitetura hierárquica na qual placas de quatro processadores são conectadas com um *switch* de alto desempenho ou com um barramento de nível mais alto. Em uma arquitetura NUMA, os processadores veem um único espaço de endereçamento contendo toda a memória de todas as placas. No entanto, a latência de acesso da memória da placa é menor do que a de um módulo de memória em outra placa – daí o nome dessa arquitetura.

Nos *multiprocessadores de memória distribuída* e agregados de computadores (*cluster*) (novamente, consulte a Seção 7.3), os processadores não compartilham memória, mas são conectados por uma rede de alta velocidade. Esses sistemas, assim como os sistemas distribuídos de propósito geral, podem chegar a números muito maiores de processadores do que os aproximadamente 64 do multiprocessador de memória compartilhada. Uma questão central que tem sido examinada pelas comunidades de pesquisa sobre DSM e multiprocessadores é se o investimento no entendimento dos algoritmos de memória compartilhada e o *software* associado pode ser diretamente transferido para uma arquitetura de memória distribuída mais expansível.

Passagem de mensagens versus DSM • Como mecanismo de comunicação, o DSM se compara com a passagem de mensagens e não com a comunicação baseada em requisição-resposta, pois sua aplicação em processamento paralelo, em particular, requer o uso de comunicação assíncrona. As estratégias de programação de DSM e passagem de mensagens podem ser contrastadas como segue:

Serviço oferecido: no modelo de passagem de mensagens, as variáveis de um processo precisam ser empacotadas, transmitidas e desempacotadas em outras variáveis no processo de destino. Em contraste, em memória compartilhada os processos envolvidos compartilham as variáveis diretamente, de modo que não é necessário nenhum empacotamento – nem mesmo de ponteiros para as variáveis compartilhadas – e, assim, não são necessárias operações de comunicação separadas. A maioria das implementações permite que as variáveis armazenadas na DSM sejam nomeadas e acessadas de modo similar às variáveis não compartilhadas normais. Por outro lado, a vantagem da passagem de mensagens é que ela permite aos processos se comunicarem, ao passo que eles são protegidos uns dos outros por terem espaços de endereçamento privativos, enquanto os processos que compartilham a DSM podem, por exemplo, causar falhas recíprocas por alterar dados erroneamente. Além disso, quando é usada passagem de mensagens entre computadores heterogêneos, o empacotamento cuida das diferenças na representação de dados; mas como a memória pode ser compartilhada entre computadores com diferentes representações de inteiro, por exemplo?

A sincronização entre processos é obtida no modelo de mensagem por meio das próprias primitivas de passagem de mensagens, usando técnicas como a implementação de servidor com o uso de travas (*locks*), discutida no Capítulo 16. No caso da DSM, a sincronização é feita por meio de construções normais da programação de memória compartilhada, como travas e semáforos (embora exijam implementações diferentes no ambiente de memória distribuída). O Capítulo 7 discute brevemente esses objetos de sincronização no contexto da programação com *threads*.

Por fim, como a DSM pode ser persistente, os processos que se comunicam via DSM podem executar com tempos não sobrepostos. Um processo pode deixar dados em um local da memória combinado para que outro examine quando for executado. Em contraste, os processos que se comunicam por meio de passagem de mensagens devem ser executados ao mesmo tempo.

Eficiência: as experiências mostram que certos programas paralelos desenvolvidos para DSM podem ser feitos de forma a executar quase tão bem quanto programas funcionalmente equivalentes escritos para plataformas de passagem de mensagens no mesmo *hardware* [Carter *et al.* 1991] – pelo menos no caso de números de com-

putadores relativamente pequenos (dez, mais ou menos). Contudo, esse resultado não pode ser generalizado. O desempenho de um programa baseado em DSM depende de muitos fatores, conforme vamos discutir a seguir – particularmente do padrão de compartilhamento de dados (como se um item fosse atualizado por vários processos).

Há uma diferença na visibilidade dos custos associados aos dois tipos de programação. Na passagem de mensagens, todos os acessos a dados remotos são explícitos e, portanto, o programador sempre sabe se uma operação em particular é interna ao processo ou envolve o custo da comunicação. Contudo, com DSM, qualquer leitura ou atualização em particular pode ou não envolver comunicação por parte do suporte de execução subjacente. Se isso acontece ou não, depende de fatores como se os dados foram acessados antes e do padrão de compartilhamento entre processos nos diferentes computadores.

Não há uma resposta conclusiva quanto à DSM ser preferível à passagem de mensagens para uma aplicação em particular. A DSM continua sendo uma ferramenta cuja situação final depende da eficiência com que pode ser implementada.

6.5.2 Comunicação via espaço de tuplas

Os espaços de tuplas foram apresentados pela primeira vez por David Gelernter, da Universidade de Yale, como uma nova forma de computação distribuída baseada no que ele chama de *comunicação gerativa* [Gelernter 1985]. Nessa estratégia, os processos se comunicam indiretamente, colocando tuplas em um espaço de tuplas, enquanto outros processos podem ler ou remover tuplas desse espaço. As tuplas não têm endereço, mas são acessadas por casamento de padrões no conteúdo (memória endereçada pelo conteúdo, conforme discutido anteriormente). O modelo de programação Linda, resultante dessas pesquisas, influenciou e levou a desenvolvimentos importantes na programação distribuída, incluindo sistemas como Agora [Bisiani e Forin 1988] e, mais significativamente, JavaSpaces, da Sun (discutido a seguir), e TSpace, da IBM. A comunicação via espaço de tuplas também tem sido utilizada no campo da computação ubíqua, por motivos explorados no Capítulo 19.

Esta seção examina o paradigma do espaço de tuplas conforme ele se aplica à computação distribuída. Começamos estudando o modelo de programação oferecido pelo espaço de tuplas, antes de considerarmos brevemente os problemas de implementação associados. A seção termina examinando a especificação JavaSpaces como um estudo de caso, ilustrando como o espaço de tuplas evoluiu para abranger o mundo orientado a objetos.

O modelo de programação • No modelo de programação com espaço de tuplas, os processos se comunicam por meio de um espaço de tuplas – uma coleção de tuplas compartilhada. As tuplas, por sua vez, consistem em uma sequência de um ou mais campos de dados tipados, como <“fred”, 1958>, <“sid”, 1964> e <4, 9.8, “Yes”>. Qualquer combinação de tipos de tuplas pode existir no mesmo espaço de tuplas. Os processos compartilham dados acessando o mesmo espaço de tuplas: eles colocam tuplas no espaço de tuplas usando a operação *write* e as leem ou extraem do espaço de tuplas usando a operação *read* ou *take*. A operação *write* adiciona uma tupla sem afetar as tuplas existentes no espaço. A operação *read* retorna o valor de uma tupla sem afetar o conteúdo do espaço de tuplas. A operação *take* também retorna uma tupla, mas, nesse caso, também remove a tupla do espaço de tuplas.

Ao ler ou remover uma tupla do espaço de tuplas, um processo fornece uma especificação de tupla, e o espaço de tuplas retorna qualquer tupla que corresponda a essa

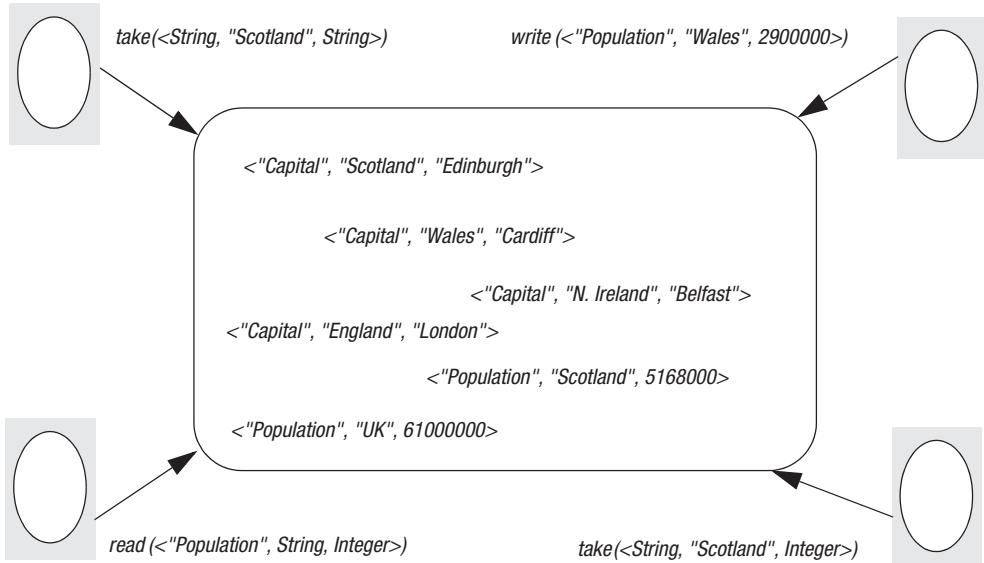


Figura 6.20 A abstração do espaço de tuplas.

especificação – conforme mencionado anteriormente, esse é um tipo de endereçamento associativo. Para permitir que os processos sincronizem suas atividades, as operações *read* e *take* bloqueiam até que haja uma tupla correspondente no espaço de tuplas. Uma especificação de tupla inclui o número de campos e os valores ou tipos exigidos dos campos. Por exemplo, *take(<String, integer>)* poderia extrair `<"fred", 1958>` ou `<"sid", 1964>`; *take(<String, 1958>)* extraaria somente `<"fred", 1958>` dessas duas.

No paradigma do espaço de tuplas, nenhum acesso direto às tuplas é permitido, e os processos têm de substituir as tuplas no espaço de tuplas, em vez de modificá-las. Assim, as tuplas são *imutáveis*. Suponha, por exemplo, que um conjunto de processos mantenha um contador compartilhado no espaço de tuplas. A contagem atual (digamos, 64) está na tupla `<"counter", 64>`. Um processo deve executar código da seguinte forma para incrementar o contador em um espaço de tuplas *myTS*:

```
<s, count>:= myTS.take(<"counter", integer>);
myTS.write(<"counter", count+1>);
```

Outra ilustração do paradigma do espaço de tuplas é dada na Figura 6.20. Esse espaço de tuplas contém uma variedade de tuplas representando informações geográficas sobre países do Reino Unido, incluindo as populações e as capitais. A operação *take take(<String, "Scotland", String>)* vai corresponder a `<"Capital", "Scotland", "Edinburgh">`, enquanto *take(<String, "Scotland", Integer>)* vai corresponder a `<"Population", "Scotland", 5168000>`. A operação *write*, *write(<"Population", "Wales", 2900000>)*, vai inserir uma nova tupla no espaço de tuplas, com informações sobre a população de Gales (Wales). Por fim, *read(<"Population", String, Integer>)* pode corresponder às tuplas equivalentes para as populações do Reino Unido (UK), Escócia (Scotland) ou mesmo Gales, caso essa operação seja executada após a operação *write* correspondente. Uma delas vai ser selecionada de modo não determinístico pela implementação de espaço de tuplas e, sendo essa uma operação *read*, a tupla vai permanecer no espaço de tuplas.

Note que *write*, *read* e *take* são conhecidas como *out*, *rd* e *in* no Linda; usamos os primeiros nomes, mais descritivos, ao longo de todo o livro. Essa terminologia também é usada no JavaSpaces, conforme apresentado a seguir.

Propriedades associadas ao espaço de tuplas: Gelernter [1985] apresenta algumas propriedades interessantes associadas à comunicação via espaço de tuplas, destacando, em particular, o desacoplamento espacial e temporal discutido na Seção 6.1.

Desacoplamento espacial: uma tupla colocada no espaço de tuplas pode se originar de qualquer número de processos remetentes e pode ser entregue a qualquer um de vários destinatários em potencial. Essa propriedade também é referida como atribuição de nomes distribuída no Linda.

Desacoplamento temporal: uma tupla colocada no espaço de tuplas vai permanecer nesse espaço de tuplas até que seja removida (talvez indefinidamente) e, assim, o remetente e o destinatário não precisam se sobrepor no tempo.

Juntos, esses pontos fornecem uma estratégia totalmente distribuída no espaço e no tempo e uma forma de *compartilhamento distribuído* de variáveis distribuídas por meio do espaço de tuplas.

Gelernter [1985] também explora várias outras propriedades associadas ao estilo bastante flexível de atribuição de nomes empregado no Linda (referido como *atribuição de nomes livre*). O leitor que estiver interessado nisso pode ler o artigo de Gelernter para mais informações sobre esse assunto.

Variações sobre o tema: desde a introdução do Linda, foram propostos refinamentos ao seu modelo original:

- O modelo original do Linda propôs um único espaço de tuplas global. Isso não é o mais adequado em sistemas grandes, levando ao perigo de sinônimos não intencionais de tuplas; à medida que o número de tuplas em um espaço de tuplas aumenta, há uma chance cada vez maior de uma operação *read* ou *take* corresponder a uma tupla acidentalmente. Isso é particularmente provável ao se fazer a correspondência de tipos, como no caso de *take(<String, integer>)*, conforme mencionado anteriormente. Devido a isso, vários sistemas propuseram *múltiplos espaços de tuplas*, incluindo a capacidade de criar espaços de tuplas dinamicamente, introduzindo um grau de escopo no sistema (consulte, por exemplo, o estudo de caso do JavaSpaces, a seguir).
- O Linda foi previsto para ser implementado como uma entidade centralizada, mas, desde então, os sistemas têm experimentado implementações *distribuídas* de espaços de tuplas (incluindo estratégias para oferecer mais tolerância a falhas). Dada a importância desse assunto para este livro, enfocamos isso na subseção sobre os problemas de implementação, a seguir.
- Os pesquisadores também tentaram a modificação ou a extensão das operações fornecidas nos espaços de tuplas e a adaptação da semântica subjacente. Uma proposta bastante interessante é unificar os conceitos de tuplas e espaços de tuplas, modelando tudo como conjuntos (não ordenados) – isto é, os espaços de tuplas são conjuntos de tuplas e as tuplas são conjuntos de valores que também podem incluir tuplas (Bauhaus Linda) [Carriero *et al.* 1995].
- Talvez, de modo mais interessante, as implementações recentes de espaços de tuplas mudaram de tuplas de itens de dados tipados para objetos de dados (com atributos), transformando o espaço de tuplas em um *espaço de objetos*. Essa proposta é adotada, por exemplo, no sistema JavaSpaces, discutido a seguir.

Problemas de implementação • Muitas implementações de espaços de tuplas adotam uma solução centralizada na qual o recurso de espaço de tuplas é gerenciado por um único servidor. Isso tem vantagens em termos de simplicidade, mas essas soluções claramente não são tolerantes a falhas e não mudam de escala. Por isso, foram propostas soluções distribuídas:

Replicação: vários sistemas propuseram o uso de *replicação* para resolver os problemas identificados anteriormente [Bakken e Schlichting 1995, Bessani *et al.* 2008, Xu e Liskov 1989].

As propostas de Bakken e Schlichting [1995] e Bessani *et al.* [2008] adotam uma estratégia semelhante, referida como *estratégia da máquina de estado* para replicação e discutida com mais detalhes no Capítulo 18. Essa estratégia presume que um espaço de tuplas se comporta como uma máquina de estado, mantendo o estado e alterando-o em resposta aos eventos recebidos de outras réplicas ou do ambiente. Para garantir a coerência, (i) as réplicas devem começar no mesmo estado (um espaço de tuplas vazio), (ii) devem executar eventos na mesma ordem e (iii) devem reagir de forma determinística a cada evento. A segunda propriedade importante pode ser garantida pela adoção de um algoritmo de *multicast* totalmente ordenado, conforme discutido na Seção 6.2.2.

Xu e Liskov [1989] adotam uma estratégia diferente, a qual otimiza a estratégia de replicação usando a semântica das operações de espaço de tupla em particular. Nessa proposta, as atualizações são feitas no contexto da *visão* atual (o conjunto de réplicas combinado), e as tuplas são particionadas em *conjuntos de tuplas* distintos, baseados em seus nomes lógicos associados (designados como o primeiro campo na tupla). O sistema consiste em um conjunto de *workers* realizando computação no espaço de tuplas e em um conjunto de réplicas do espaço de tuplas. Determinado nó físico pode conter qualquer número de *workers*, réplicas ou mesmo ambos; portanto, determinado *worker* pode ter uma réplica local ou não. Os nós são conectados por uma rede de comunicação, a qual pode perder, duplicar ou retardar mensagens e pode entregar mensagens fora de ordem. Também podem ocorrer partições de rede.

Uma operação *write* é implementada pelo envio de uma mensagem *multicast* para todos os membros da visão, pelo canal de comunicação não confiável. Na recepção, os membros colocam essa tupla em suas réplicas e reconhecem o recebimento. A requisição *write* é repetida até que todos os reconhecimentos sejam recebidos. Para o funcionamento correto do protocolo, as réplicas devem detectar requisições duplicadas, reconhecê-las, mas não executar as operações *write* associadas.

A operação *read* consiste no envio de uma mensagem *multicast* para todas as réplicas. Cada réplica procura uma correspondência e retorna essa correspondência para o *site* solicitante. A primeira tupla retornada é entregue como resultado da operação *read*. Ela pode vir de um nó local, mas dado que muitos *workers* não vão ter uma réplica local, isso não é garantido.

A operação *take* é mais complexa, devido à necessidade de concordar com a tupla a ser selecionada e remover essa tupla concordante de todas as cópias. O algoritmo ocorre em duas fases. Na fase 1, a especificação de tupla é enviada para todas as réplicas, e a réplica tenta adquirir a trava no conjunto de tuplas associado para serializar requisições *take* nas réplicas (as operações *write* e *read* não são afetadas pela trava); se a trava não pode ser adquirida, a requisição *take* é recusada. Cada réplica que consegue obter a trava responde com o *conjunto* de tuplas correspondentes. Essa etapa é repetida até que todas as réplicas tenham aceitado a requisição e respondido. O processo iniciador pode, então, selecionar uma tupla a partir da interseção de todas as respostas e retornar isso como

<i>write</i>	<ol style="list-style-type: none"> 1. O <i>site</i> requisitante faz um <i>multicast</i> da requisição <i>write</i> para todos os membros da visão. 2. Ao receber essa requisição, os membros inserem a tupla em suas réplicas e confirmam essa ação. 3. O passo 1 é repetido até que todas as confirmações sejam recebidas.
<i>read</i>	<ol style="list-style-type: none"> 1. O <i>site</i> requisitante faz um <i>multicast</i> da requisição <i>read</i> para todos os membros da visão. 2. Ao receber essa requisição, um membro retorna uma tupla correspondente para o requisitante. 3. O requisitante retorna a primeira tupla correspondente recebida como resultado da operação (ignorando as outras). 4. O passo 1 é repetido até que pelo menos uma resposta seja recebida.
<i>take</i>	<p><i>Fase 1: Seleção da tupla a ser removida</i></p> <ol style="list-style-type: none"> 1. O <i>site</i> requisitante faz um <i>multicast</i> da requisição <i>take</i> para todos os membros da visão. 2. Ao receber essa requisição, cada réplica adquire uma trava no conjunto de tuplas associado e, se a trava não puder ser adquirida, a requisição <i>take</i> é rejeitada. 3. Todos os membros que aceitam respondem com o conjunto de todas as tuplas correspondentes. 4. O passo 1 é repetido até que todos os <i>sites</i> tenham aceitado a requisição e respondam com seus conjuntos de tuplas e a interseção seja não nula. 5. Uma tupla em particular é selecionada como resultado da operação (selecionada aleatoriamente a partir da interseção de todas as respostas). 6. Se apenas uma minoria aceitar a requisição, essa minoria é solicitada a liberar suas travas e a fase 1 é repetida. <p><i>Fase 2: Remoção da tupla selecionada</i></p> <ol style="list-style-type: none"> 1. O <i>site</i> requisitante faz um <i>multicast</i> da requisição <i>remove</i> para todos os membros da visão, citando a tupla a ser removida. 2. Ao receber essa requisição, os membros removem a tupla de suas réplicas, enviam uma confirmação e liberam a trava. 3. O passo 1 é repetido até que todas as confirmações sejam recebidas.

Figura 6.21 Replicação e as operações de espaço de tupla [Xu e Liskov 1989].

resultado da requisição *take*. Se não for possível obter uma maioria de travas, as réplicas são solicitadas a liberar suas travas e a fase 1 é repetida.

Na fase 2, essa tupla deve ser removida de todas as réplicas. Isso é conseguido por meio do envio repetido de *multicast* para as réplicas na visão, até que todas tenham reconhecido a exclusão. Assim como acontece com as requisições *write*, é necessário que as réplicas detectem requisições repetidas na fase 2 e simplesmente enviem outra confirmação sem realizar outra exclusão (caso contrário, várias tuplas podem ser excluídas erroneamente nesse estágio).

As etapas envolvidas em cada operação estão resumidas na Figura 6.21. Note que é necessário um algoritmo separado para gerenciar mudanças de visão, caso ocorram falhas de nó ou a rede seja particionada (consulte Xu e Liskov [1989] para detalhes).

Esse algoritmo é projetado para minimizar o atraso devido à semântica das três operações de espaço de tuplas:

- As operações *read* só bloqueiam até que a primeira réplica responda à requisição.
- As operações *take* bloqueiam até o final da fase 1, quando a tupla a ser excluída tiver sido selecionada.
- As operações *write* podem retornar imediatamente.

Contudo, isso introduz níveis de concorrência inaceitáveis. Por exemplo, uma operação *read* pode acessar uma tupla que talvez tenha sido excluída na segunda fase de uma

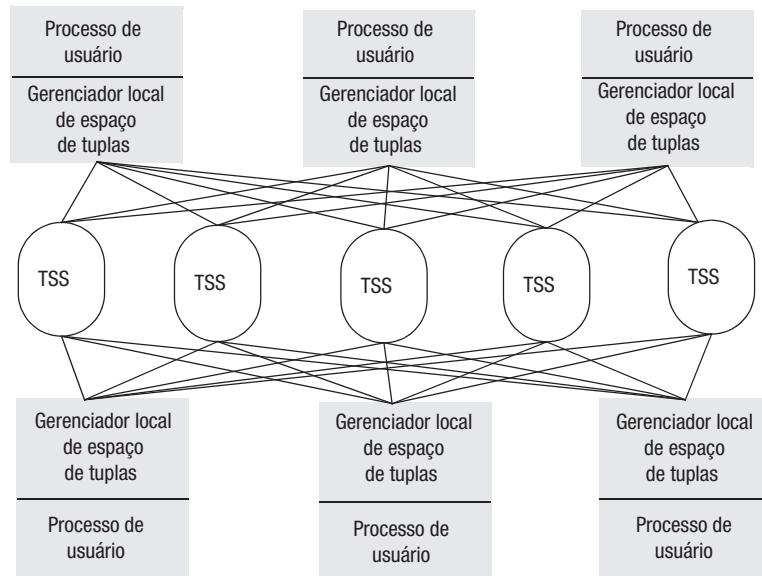


Figura 6.22 Particionamento no York Linda Kernel.

operação *take*. Portanto, são necessários níveis de controle de concorrência adicionais. Em particular, Xu e Liskov [1989] introduzem as seguintes restrições adicionais:

- As operações de cada *worker* devem ser executadas em cada réplica na mesma ordem em que foram efetuadas pelo *worker*;
- Uma operação *write* não deve ser executada em nenhuma réplica até que todas as operações *take* anteriores efetuadas pelo mesmo *worker* tenham terminado em todas as réplicas na visão do *worker*.

Outro exemplo de uso de replicação é fornecido no Capítulo 19, em que apresentamos a estratégia L²imbo, que usa replicação para fornecer alta disponibilidade em ambientes móveis [Davies *et al.* 1998].

Outras estratégias: diversas outras estratégias têm sido empregadas na implementação da abstração de espaço de tuplas, incluindo o particionamento do espaço de tuplas por vários nós ou o mapeamento em redes *peer-to-peer*:

- O Linda Kernel, desenvolvido na Universidade de York [Rowstron e Wood 1996], adota uma estratégia na qual as tuplas são particionadas por diversos servidores de espaço de tuplas (TSSs) disponíveis, conforme ilustrado na Figura 6.22. Não há replicação de tuplas; isto é, existe apenas uma cópia de cada tupla. O motivo é aumentar o desempenho do espaço de tuplas, especialmente para computação altamente paralela. Quando uma tupla é colocada no espaço de tuplas, um algoritmo de *hashing* seleciona um dos servidores de espaço de tuplas para ser usado. A implementação de *read* ou *take* é um pouco mais complexa, pois é fornecida uma especificação de tupla que pode descrever tipos ou valores dos campos associados. O algoritmo de *hashing* usa essa especificação para gerar um conjunto de possíveis servidores que podem conter tuplas correspondentes e, então, uma busca linear

<i>Operação</i>	<i>Efeito</i>
<i>Lease write(Entry e, Transaction txn, long lease)</i>	Coloca uma entrada em um JavaSpace específico
<i>Entry read(Entry tmpl, Transaction txn, long timeout)</i>	Retorna uma cópia de uma entrada correspondente a um modelo especificado
<i>Entry readIfExists(Entry tmpl, Transaction txn, long timeout)</i>	Como o anterior, mas sem bloqueio
<i>Entry take(Entry tmpl, Transaction txn, long timeout)</i>	Recupera (e remove) uma entrada correspondente a um modelo especificado
<i>Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)</i>	Como o anterior, mas sem bloqueio
<i>EventRegistration notify(Entry tmpl, Transaction txn, RemoteEventListener listen, long lease, MarshalledObject handback)</i>	Notifica um processo, caso uma tupla correspondente a um modelo especificado seja gravada em um JavaSpace

Figura 6.23 A API JavaSpaces.

deve ser empregada até que uma tupla correspondente seja descoberta. Note que, como existe apenas uma cópia de determinada tupla, a implementação de *take* é bastante simplificada.

- Algumas implementações de espaços de tuplas adotaram estratégias *peer-to-peer* nas quais todos os nós cooperam para fornecer o serviço de espaço de tuplas. Essa estratégia é particularmente atraente, em razão da disponibilidade e da escalabilidade intrínsecas das soluções *peer-to-peer*. Exemplos de implementações *peer-to-peer* incluem PeerSpaces [Busi *et al.* 2003], que foi desenvolvido usando o middleware *peer-to-peer* JXTA [jxta.dev.java.net], LIME e TOTA (estes dois últimos sistemas serão apresentados no Capítulo 19).

Estudo de caso: JavaSpaces • O JavaSpaces é uma ferramenta para comunicação via espaço de tuplas desenvolvido pela Sun [[java.sun.com X](http://java.sun.com/X), java.sun.com VI]. Mais especificamente, a Sun fornece a especificação de um serviço JavaSpaces e, então, os desenvolvedores estão livres para oferecer implementações de JavaSpaces (implementações importantes incluem GigaSpaces [www.gigaspaces.com] e Blitz [www.dancres.org]). A ferramenta é fortemente dependente do Jini (o serviço de descoberta da Sun, discutido com mais detalhes na Seção 19.2.1), conforme ficará claro a seguir. O *Jini Technology Starter Kit* também inclui uma implementação de JavaSpaces, referida como Outrigger.

Os objetivos da tecnologia JavaSpaces são:

- oferecer uma plataforma que simplifique o projeto de aplicativos e serviços distribuídos;
- ser simples e mínima em termos do número e de tamanho de classes associadas e ter uma base pequena para permitir a execução de código em equipamentos de recursos limitados (como os *smartphones*);
- permitir implementações replicadas da especificação (embora, na prática, a maioria das implementações seja centralizada).

Programação com JavaSpaces: JavaSpaces permite ao programador criar qualquer número de instâncias de um *espaço*, onde espaço é um repositório persistente e compartilhado de *objetos* (oferecendo, assim, um espaço de objetos, na terminologia introduzida anteriormente). Mais especificamente, um item em um JavaSpace é referido como *entrada*: um

```
import net.jini.core.entry.*;
public class AlarmTupleJS implements Entry {
    public String alarmType;
    public AlarmTupleJS() {
    }
    public AlarmTupleJS(String alarmType) {
        this.alarmType = alarmType;
    }
}
```

Figura 6.24 Classe Java *AlarmTupleJS*.

grupo de objetos contido em uma classe que implementa *net.jini.core.entry.Entry*. Note que, com as entradas contendo objetos (em vez de tuplas), é possível associar comportamento arbitrário a elas, aumentando significativamente o poder expressivo da estratégia.

As operações definidas no JavaSpaces estão resumidas na Figura 6.23 (que mostra as assinaturas completas de cada uma das operações):

- Um processo pode colocar uma entrada em uma instância de JavaSpace com o método *write*. Assim como no Jini, uma entrada pode ter um *arrendamento* associado (consulte a Seção 5.4.3), o tempo durante o qual o acesso é garantido aos objetos associados. Pode ser para sempre (*Lease.FOREVER*) ou um valor numérico, especificado em milissegundos. Após esse período, a entrada é destruída. A operação *write* também pode ser usada no contexto de uma *transação*, conforme discutido a seguir (o valor *null* indica que essa não é uma operação transacional). A operação *write* retorna o valor *Lease* representando o arrendamento garantido pelo JavaSpace (que pode ser menor do que o tempo solicitado).
- Um processo pode acessar uma entrada em um JavaSpace com a operação *read* ou *take*; *read* retorna uma cópia da entrada correspondente e *take* remove uma entrada correspondente do JavaSpace (como no modelo de programação geral apresentado anteriormente). Os requisitos de correspondência são especificados por um *modelo*, o qual é de tipo *entry*. Campos específicos no modelo podem ser configurados com valores determinados e outros podem ser deixados sem especificação. Então, uma correspondência é definida como uma entrada que é da mesma classe do modelo (ou uma subclasse válida) e onde há uma correspondência exata para o conjunto de valores especificados. Assim como acontece com *write*, *read* e *take* podem ser executadas no contexto de uma transação especificada (discutido a seguir). As duas operações também bloqueiam; o último parâmetro especifica um tempo limite, representando o período de tempo máximo que um processo ou *thread* em particular bloqueará, por exemplo, para lidar com a falha de um processo fornecendo determinada entrada. As operações *readyIfExists* e *takeIfExists* são equivalentes a *read* e *take*, respectivamente, mas essas operações retornam uma entrada correspondente, se existir uma; caso contrário, retornarão *null*.
- A operação *notify* usa notificação de evento distribuída Jini, mencionada na Seção 6.3, para registrar interesse em determinado evento – neste caso, a chegada de entradas correspondentes ao modelo dado. Esse registro é governado por um arrendamento; isto é, o período de tempo que o registro deve persistir no JavaSpace. A notificação é feita por meio de uma interface *RemoteEventListener* especificada.

```

import net.jini.space.JavaSpace;
public class FireAlarmJS {
    public void raise() {
        try {
            JavaSpace space = SpaceAccessor.findSpace("AlarmSpace");
            AlarmTupleJS tuple = new AlarmTupleJS("Fire!");
            space.write(tuple, null, 60*60*1000);
        } catch (Exception e) {
        }
    }
}

```

Figura 6.25 Classe Java *FireAlarmJS*.

Mais uma vez, essa operação pode ser efetuada no contexto de uma transação especificada.

Conforme mencionado na discussão anterior, as operações no JavaSpaces podem ocorrer no contexto de uma *transação*, garantindo que todas as operações contidas nessa transação sejam executadas ou nenhuma o seja. As transações são entidades distribuídas e podem abranger vários JavaSpaces e vários processos participantes. A discussão sobre o conceito geral de transações é deixada para o Capítulo 16.

Um exemplo simples: Concluímos este exame do JavaSpaces apresentando um exemplo – o alarme de incêndio inteligente que aparece na Seção 6.2.3 e é revisto na Seção 6.4.3. Neste exemplo, há necessidade de disseminar uma mensagem de emergência para todos os destinatários quando um incêndio é detectado.

Começamos definindo um objeto entrada de tipo *AlarmTupleJS*, como mostrado na Figura 6.24. Isso é relativamente simples e mostra a criação de uma nova entrada com apenas um campo, *alarmType*. O código de alarme de incêndio associado aparece na Figura 6.25. O primeiro passo para disparar o alarme é obter acesso a uma instância apropriada de JavaSpace (chamada “*AlarmSpace*”), que supomos já estar criada. A maioria das implementações de JavaSpaces fornece funções utilitárias para isso e, por simplicidade, é isso que mostramos nesse código, usando a classe *SpaceAccessor* e o método *findSpace* fornecidos no GigaSpaces (por conveniência, uma cópia dessa classe é fornecida no site que acompanha o livro [www.cdk5.net] – em inglês). Então, uma entrada é criada como uma instância de *AlarmTupleJS* definida anteriormente. Essa entrada tem apenas um campo, um *string* chamado de *alarmType*, e isso é configurado como “Fire!”. Por fim, essa entrada é colocada no JavaSpace usando-se o método *write*, onde permanecerá por uma hora. Esse código pode, então, ser chamado usando-se o seguinte:

```

FireAlarmJS alarm = new FireAlarmJS();
alarm.raise();

```

O código correspondente do lado consumidor aparece na Figura 6.26. O acesso ao JavaSpace apropriado é obtido da mesma maneira. Depois disso, um modelo é criado, o único campo é configurado como “Fire!” e um método *read* associado é chamado. Note que, configurando o campo como “Fire！”, garantimos que sejam retornadas somente as entradas com esse tipo e com esse valor (deixar o campo em branco tornaria qualquer entrada de tipo *AlarmTupleJS* uma correspondência válida). Isso é chamado como segue em um consumidor:

```
import net.jini.space.JavaSpace;
public class FireAlarmConsumerJS {
    public String await() {
        try {
            JavaSpace space = SpaceAccessor.findSpace();
            AlarmTupleJS template = new AlarmTupleJS("Fire!");
            AlarmTupleJS recv = (AlarmTupleJS) space.read(template, null,
                Long.MAX_VALUE);
            return recv.alarmType;
        }
        catch (Exception e) {
            return null;
        }
    }
}
```

Figura 6.26 Classe Java *FireAlarmReceiverJS*.

```
FireAlarmConsumerJS alarmCall = new FireAlarmConsumerJS();
String msg = alarmCall.await();
System.out.println("Alarm received: "+msg);
```

Esse exemplo simples ilustra como é fácil escrever aplicativos em vários módulos separados desacoplados no tempo e no espaço, usando JavaSpaces.

6.6 Resumo

Este capítulo examinou a comunicação indireta em detalhes, complementando o estudo dos paradigmas de invocação remota do capítulo anterior. Definimos a comunicação indireta em termos de comunicação por meio de um intermediário, com o desacoplamento resultante entre produtores e consumidores de mensagens. Isso leva a propriedades interessantes, particularmente em termos de lidar com a alteração e o estabelecimento de estratégias tolerantes à falha.

Consideramos cinco estilos de comunicação indireta neste capítulo:

- comunicação em grupo;
- sistemas publicar-assinar;
- filas de mensagem;
- memória compartilhada distribuída;
- espaços de tuplas.

A discussão enfatizou suas características comuns em termos de que todos suportam comunicação indireta em forma de intermediários, incluindo grupos, canais ou tópicos, filas, memória compartilhada ou espaços de tuplas. Os sistemas publicar-assinar baseados em conteúdo se comunicam por meio do sistema publicar-assinar como um todo, com as assinaturas efetivamente definindo canais lógicos gerenciados pelo roteamento baseado em conteúdo.

Assim como enfocar as características comuns, é instrutivo considerar também as principais diferenças entre as várias estratégias. Começamos reconsiderando o nível de

desacoplamento espacial e temporal, retomando a discussão da Seção 6.1. Todas as técnicas consideradas neste capítulo exibem desacoplamento espacial, no sentido de que as mensagens são direcionadas a um intermediário e não para qualquer destinatário (ou destinatários). A posição com relação ao desacoplamento temporal é mais sutil e dependente do nível de persistência no paradigma. Filas de mensagens, memória compartilhada distribuída e espaços de tuplas, todos exibem desacoplamento temporal. Os outros paradigmas podem depender da implementação. Por exemplo, em algumas implementações na comunicação em grupo, é possível um destinatário ingressar em um grupo em um ponto arbitrário no tempo e ser atualizado com relação às trocas de mensagens anteriores (esse é um recurso opcional no JGroups, por exemplo, selecionado pela construção de uma pilha de protocolos apropriada). Muitos sistemas publicar-assinar não suportam persistência de eventos e, assim, não são desacoplados no tempo, mas existem exceções. O JMS, por exemplo, suporta eventos persistentes, de acordo com sua integração de publicar-assinar e filas de mensagem.

A observação seguinte é a de que as três técnicas iniciais (grupos, publicar-assinar e filas de mensagem) oferecem um modelo de programação que enfatiza a *comunicação* (por meio de mensagens ou eventos), enquanto a memória compartilhada distribuída oferece uma *abstração baseada em estado*. Essa é uma diferença fundamental e tem repercussões significativas, em termos de escalabilidade; de modo geral, as abstrações baseadas em comunicação têm o potencial de abranger sistemas de escala muito grande, com infraestrutura de roteamento apropriada (embora esse não seja o caso para a comunicação em grupo, devido à necessidade de se manter a participação como membro do grupo, conforme discutido na Seção 6.2.2). Em contraste, as duas estratégias baseadas em estado têm limitações com relação à mudança de escala. Isso ocorre como resultado da necessidade de manter visões coerentes do estado compartilhado, por exemplo, entre vários leitores e escritores de memória compartilhada. A situação, no caso dos espaços de tuplas, é um pouco mais sutil, em razão da natureza imutável das tuplas. O principal problema está na implementação da operação de leitura destrutiva, *take*, em um sistema de grande escala; é interessante observar que, sem essa operação, os espaços de tuplas são muito parecidos com os sistemas publicar-assinar (e, assim, potencialmente podem mudar bastante de escala).

A maioria dos sistemas citados anteriormente também oferece estilos de comunicação de *um para muitos*; ou seja, *multicast* em termos dos serviços baseados em comunicação e do acesso global a valores compartilhados nas abstrações baseadas em estado. As exceções são o enfileiramento de mensagens, que é fundamentalmente *ponto a ponto* (e, portanto, frequentemente oferecido em combinação com os sistemas publicar-assinar em *middleware* comercial), e os espaços de tuplas, que podem ser de *um para muitos* ou *ponto a ponto*, dependendo de os processos receptores usarem operações *read* ou *take*, respectivamente.

Também existem diferenças no *objetivo* dos vários sistemas. A comunicação em grupo é projetada principalmente para suportar sistemas distribuídos confiáveis e, assim, a ênfase está em fornecer suporte algorítmico para confiabilidade e ordenação de entrega de mensagens. É interessante notar que os algoritmos para garantir confiabilidade e ordenação (especialmente esta última) podem ter um efeito negativo significativo sobre a escalabilidade, por motivos semelhantes à manutenção de visões coerentes do estado compartilhado. Os sistemas publicar-assinar têm-se destinado amplamente à disseminação de informações (por exemplo, em sistemas financeiros) e para integração de aplicativo empresarial. Por fim, as estratégias de memória compartilhada geralmente têm sido

	<i>Grupos</i>	<i>Sistemas publicar-assinar</i>	<i>Filas de mensagem</i>	<i>DSM</i>	<i>Espaços de tuplas</i>
<i>Desacoplado no espaço</i>	Sim	Sim	Sim	Sim	Sim
<i>Desacoplado no tempo</i>	Possível	Possível	Sim	Sim	Sim
<i>Estilo de serviço</i>	Baseado em comunicação	Baseado em comunicação	Baseado em comunicação	Baseado em estado	Baseado em estado
<i>Padrão de comunicação</i>	Um para muitos	Um para muitos	Um para um	Um para muitos	Um para um ou um para muitos
<i>Principal objetivo</i>	Computação distribuída	Disseminação de informações ou EAI; sistemas móveis e ubíquos	Computação distribuída Disseminação de informações ou EAI; processamento de transações comerciais	Computação paralela e distribuída	Computação paralela e distribuída; sistemas móveis e ubíquos
<i>Escalabilidade</i>	Limitada	Possível	Possível	Limitada	Limitada
<i>Associativo</i>	Não	Somente publicar-assinar baseada em conteúdo	Não	Não	Sim

Figura 6.27 Resumo dos estilos de comunicação indireta.

aplicadas em processamento paralelo e distribuído, inclusive na comunidade Grid (embora os espaços de tuplas sejam usados eficientemente em uma variedade de domínios de aplicação). Tanto os sistemas publicar-assinar como a comunicação via espaço de tuplas têm encontrado aceitação na computação móvel e ubíqua, devido a seu suporte para ambientes voláteis (conforme discutido no Capítulo 19).

Outra questão importante associada aos cinco esquemas é que tanto a publicação-assinatura baseada em conteúdo como os espaços de tuplas oferecem uma forma de *endereçamento associativo* baseado no conteúdo, permitindo o casamento de padrões entre assinaturas e de eventos ou modelos em relação às tuplas, respectivamente. As outras estratégias não oferecem isso.

A discussão está resumida na Figura 6.27.

Exercícios

- 6.1 Formule um argumento explicando por que a comunicação indireta pode ser apropriada em ambientes voláteis. Até que ponto isso pode ter origem no desacoplamento temporal, no desacoplamento espacial ou mesmo em uma combinação de ambos?

página 230

- 6.2 A Seção 6.1 declara que a passagem de mensagens é acoplada no tempo e no espaço – isto é, as mensagens são direcionadas a uma entidade em particular e exigem que o destinatário esteja presente no momento do envio da mensagem. Contudo, considere o caso em que as mensagens são direcionadas para um nome, em vez de um endereço, e esse nome é solucionado usando DNS. Esse sistema exibe o mesmo nível de indireção? *página 231, Seção 13.2.3*
- 6.3 A Seção 6.1 se refere a sistemas que são acoplados no espaço, mas desacoplados no tempo – isto é, as mensagens são direcionadas para determinado destinatário (ou destinatários), mas esse destinatário pode ter tempo de vida independente em relação ao remetente. Você pode construir um paradigma de comunicação com essas propriedades? Por exemplo, o *e-mail* cai nessa categoria? *página 231*
- 6.4 Como um segundo exemplo, considere o paradigma de comunicação referido como RPC enfileirado, conforme apresentado no Rover [Joseph *et al.* 1997]. O Rover é um *toolkit* para suportar programação de sistemas distribuídos em ambientes móveis, em que os participantes na comunicação podem ficar desconectados por períodos de tempo. O sistema oferece o paradigma RPC e, assim, as chamadas são direcionadas para determinado servidor (claramente acoplado no espaço). Contudo, as chamadas são roteadas por meio de um intermediário – uma fila no lado *remetente* – e mantidas na fila até que o destinatário esteja disponível. Até que ponto isso é desacoplado no tempo? Dica: considere a questão quase filosófica de se um destinatário que está temporariamente indisponível existe nesse ponto do tempo. *página 231, Capítulo 19*
- 6.5 Se um paradigma de comunicação é assíncrono, ele também é desacoplado no tempo? Explique sua resposta com exemplos apropriados. *página 232*
- 6.6 No contexto de um serviço de comunicação em grupo, dê exemplos de trocas de mensagem que ilustrem a diferença entre ordenação causal e total. *página 236*
- 6.7 Considere o exemplo *FireAlarm* escrito usando JGroups (Seção 6.2.3). Suponha que ele foi generalizado para suportar uma variedade de tipos de alarme, como incêndio, enchente, invasão, etc. Quais são os requisitos desse aplicativo em termos de confiabilidade e de ordenação? *página 230, página 240*
- 6.8 Sugira um projeto para um serviço de caixa de correio destinada a armazenar notificações em nome de vários assinantes, permitindo a eles especificar quando desejam que as notificações sejam entregues. Explique como os assinantes que nem sempre estão ativos podem usar o serviço descrito. Como o serviço vai lidar com assinantes que falham por colapso enquanto estão com a entrega ativada? *página 245*
- 6.9 Nos sistemas publicar-assinar, explique como as estratégias baseadas em canal podem ser implementadas de forma trivial, usando um serviço de comunicação em grupo. Por que essa abordagem não é a mais adequada para implementar uma estratégia baseada em conteúdo? *página 245*
- 6.10 Usando o algoritmo de roteamento baseado em filtragem da Figura 6.11 como ponto de partida, desenvolva um algoritmo alternativo que ilustre como o uso de anúncios pode resultar em otimização significativa em termos do tráfego de mensagens gerado. *página 251*
- 6.11 Formule um guia passo a passo, explicando o funcionamento do algoritmo de roteamento baseado em *rendez-vous* alternativo fornecido na Figura 6.12. *página 252*
- 6.12 Complementando sua resposta ao Exercício 6.11, discuta duas possíveis implementações de $EN(e)$ e $SN(s)$. Por que a interseção de $EN(e)$ e $SN(s)$ deve ser não nula para um e dado correspondente a s (a regra de interseção)? Isso se aplica a suas possíveis implementações? *página 252*

- 6.13 Explique como o acoplamento fraco inerente às filas de mensagem pode ajudar na integração de aplicativo empresarial. Como no Exercício 6.1, considere até que ponto isso pode ter origem no desacoplamento temporal, no desacoplamento espacial ou em uma combinação de ambos. *página 254*
- 6.14 Considere a versão do programa *FireAlarm* escrita em JMS (Seção 6.4.3). Como você estenderia o consumidor para receber alarmes apenas de determinado local? *página 261*
- 6.15 Explique sob quais aspectos a DSM é conveniente ou inconveniente para sistemas cliente-servidor. *página 262*
- 6.16 Discuta se passagem de mensagens ou DSM é preferível para aplicativos tolerantes a falhas. *página 262*
- 6.17 Supondo que um sistema DSM é implementado no *middleware* sem qualquer suporte de *hardware* e de maneira neutra quanto à plataforma, como você lidaria com o problema das diferentes representações de dados em computadores heterogêneos? Sua solução abrange ponteiros? *página 262*
- 6.18 Como você implementaria o equivalente a uma chamada de procedimento remoto usando um espaço de tuplas? Quais são as vantagens e desvantagens de implementar uma interação estilo chamada de procedimento remoto dessa maneira? *página 265*
- 6.19 Como você implementaria um semáforo usando um espaço de tuplas? *página 265*
- 6.20 Implemente um espaço de tuplas replicado usando o algoritmo de Xu e Liskov [1989]. Explique como esse algoritmo usa a semântica das operações de espaço de tuplas para otimizar a estratégia de replicação. *página 269*

7

Sistema Operacional

- 7.1 Introdução
- 7.2 A camada do sistema operacional
- 7.3 Proteção
- 7.4 Processos e threads
- 7.5 Comunicação e invocação
- 7.6 Arquiteturas de sistemas operacionais
- 7.7 Virtualização em nível de sistema operacional
- 7.8 Resumo

Este capítulo descreve como o *middleware* é suportado pelos recursos do sistema operacional nos nós de um sistema distribuído. O sistema operacional facilita o encapsulamento e a proteção dos recursos dentro dos servidores e disponibiliza mecanismos necessários para acessar esses recursos, incluindo a comunicação e o escalonamento.

Um tema importante do capítulo é a função do núcleo do sistema operacional. O capítulo pretende permitir ao leitor entender as vantagens e desvantagens da divisão da funcionalidade entre os domínios de proteção – em particular, da divisão da funcionalidade entre o núcleo (*kernel*) e o código em nível de usuário. Serão discutidos os compromissos entre os recursos em nível de núcleo e em nível de usuário, incluindo o compromisso entre eficiência e robustez.

O capítulo examina o projeto e a implementação de *multithreading* e de recursos de comunicação. Ele explora as principais arquiteturas de núcleo de sistemas operacionais que já foram inventadas e examina o importante papel que a virtualização está desempenhando na arquitetura dos sistemas operacionais.

7.1 Introdução

O Capítulo 2 apresentou as camadas de *software* mais importantes de um sistema distribuído. Aprendemos que um aspecto importante dos sistemas distribuídos é o compartilhamento de recursos. Os aplicativos clientes invocam operações em recursos que frequentemente estão em outro nó, ou pelo menos em outro processo. Aplicativos (na forma de clientes) e serviços (na forma de gerenciadores de recursos) usam a camada de *middleware* para suas interações. O *middleware* fornece comunicação remota entre objetos ou processos nos nós de um sistema distribuído. O Capítulo 5 explicou os principais tipos de invocação remota encontrados na camada de *middleware*, como a RMI Java e o CORBA. O Capítulo 6 explorou estilos de comunicação indiretos alternativos. Neste capítulo, abordaremos o suporte para essa comunicação remota, sem garantias de tempo real. (O Capítulo 20 examinará o suporte para comunicação multimídia, que se dá em tempo real e é orientada a fluxo.)

Abaixo da camada de *middleware* está a camada do sistema operacional (SO), assunto deste capítulo. Vamos examinar o relacionamento entre as duas e, em particular, até que ponto os requisitos do *middleware* podem ser satisfeitos pelo sistema operacional. Esses requisitos incluem o acesso eficiente e robusto aos recursos físicos e a flexibilidade para implementar uma variedade de políticas de gerenciamento de recursos.

A tarefa de qualquer sistema operacional é fornecer abstrações dos recursos físicos subjacentes – processadores, memória, comunicação e mídias de armazenamento. Um sistema operacional como o UNIX (e suas variantes, como o Linux e o Mac OS X) ou o Windows (e suas variantes, como o XP, o Vista e o Windows 7) fornecem ao programador, por exemplo, a abstração de arquivos, em vez de blocos de disco, e soquetes, em vez de acesso direto à rede. O sistema operacional gerencia os recursos físicos de um computador apresentando-os através de suas abstrações via uma interface denominada de chamada de sistema.

Antes de iniciarmos nossa abordagem detalhada do suporte que um sistema operacional oferece para *middlewares*, é interessante ter certa perspectiva histórica, examinando dois conceitos do sistema operacional que estão relacionados ao desenvolvimento de sistemas distribuídos: sistemas operacionais de rede e sistemas operacionais distribuídos. As definições variam, mas os conceitos por trás delas são os que vêm a seguir.

O UNIX e o Windows são exemplos de *sistemas operacionais de rede*. Eles têm um recurso de interligação em rede incorporado e, portanto, podem ser usados para acessar recursos remotos. O acesso é transparente com relação à rede para alguns tipos de recurso, mas não para todos. Por exemplo, por meio de um sistema de arquivos distribuído, como o NFS, os usuários têm acesso transparente aos arquivos no que diz respeito à rede. Isto é, muitos arquivos que os usuários acessam são armazenados de forma remota em um servidor e isso é totalmente transparente para seus aplicativos.

Contudo, a característica marcante é que os nós que estão sendo executados em um sistema operacional de rede mantêm a autonomia no gerenciamento de seus próprios recursos de processamento. Em outras palavras, existem várias imagens do sistema, uma por nó. Com um sistema operacional de rede, um usuário pode se conectar em outro computador de forma remota, usando *ssh*, por exemplo, e executar processos nele. Entretanto, ao contrário do controle exercido pelo sistema operacional sobre os processos que estão sendo executados em seu próprio nó, ele não escalona os processos nos vários nós.

Em contraste, alguém poderia imaginar um sistema operacional no qual o usuário nunca se preocupasse com o local onde seus programas são executados, ou com a localização de quaisquer recursos. Haveria uma *imagem única do sistema*. O sistema operacional teria controle sobre todos os nós do sistema e dispararia novos processos de forma

transparente, no nó mais conveniente, de acordo com sua política de escalonamento. Por exemplo, o sistema operacional poderia criar um novo processo no nó menos carregado do sistema para evitar que nós individuais se tornassem desnecessariamente sobrecarregados.

Um sistema operacional que produz uma imagem única do sistema como esse, para todos os recursos de um sistema distribuído, é chamado de *sistema operacional distribuído* [Tanenbaum e van Renesse 1985].

Middleware e sistemas operacionais de rede • Na verdade, não existem sistemas operacionais distribuídos para uso geral, existem apenas sistemas operacionais de rede como UNIX, Mac OS e Windows. É provável que continue assim, por dois motivos principais. O primeiro é que os usuários têm muito investimento feito em *software* aplicativo, o qual frequentemente atende a suas necessidades atuais; eles não adotarão um novo sistema operacional que não execute seus aplicativos, independentemente das vantagens relacionadas à eficiência que ele ofereça. Foram feitas tentativas de simular o UNIX, e outros núcleos de sistema operacional, sobre novos núcleos, mas o desempenho das simulações não foi satisfatório. De qualquer modo, manter simulações atualizadas de todos os principais sistemas operacionais, à medida que eles evoluem, seria uma tarefa enorme.

O segundo motivo contra a adoção de sistemas operacionais distribuídos é que os usuários preferem ter certo grau de autonomia em suas máquinas, mesmo em uma empresa muito fechada. Isso acontece particularmente devido ao desempenho [Douglis e Ousterhout 1991]. Por exemplo, Jones precisa de boa capacidade de resposta interativa enquanto escreve seus documentos e se ressentiria se os programas de Smith diminuíssem a velocidade de processamento de sua máquina.

A combinação de *middleware* e sistemas operacionais de rede proporciona um equilíbrio aceitável entre os requisitos de autonomia, por um lado, e o acesso aos recursos, transparente com relação à rede, por outro. O sistema operacional de rede permite que os usuários executem, de forma independente dos demais, seu processador de textos predileto e outros aplicativos. A camada de *middleware* permite que eles tirem proveito de serviços distribuídos que se tornem disponíveis para seu sistema operacional.

A próxima seção explicará a função da camada do sistema operacional. A Seção 7.3 examinará os mecanismos de baixo nível para proteção de recursos, os quais precisamos entender para avaliar o relacionamento entre processos e *threads* e a função do núcleo em si. A Seção 7.4 examinará as abstrações de processo, espaço de endereçamento e de *thread*. Aqui, os principais tópicos são a concorrência, o gerenciamento e a proteção de recurso local e o escalonamento. Em seguida, a Seção 7.5 abordará a comunicação como parte dos mecanismos de invocação. A Seção 7.6 discutirá os diferentes tipos de arquitetura de sistemas operacionais, incluindo os assim chamados projetos monolíticos e de micronúcleo. O leitor poderá encontrar estudos de caso do núcleo Mach e dos sistemas operacionais Amoeba, Chorus e Clouds, no endereço www.cdk5.net/oss (em inglês). O capítulo termina examinando o papel que a virtualização está desempenhando no projeto de sistemas operacionais, apresentando o Xen como um estudo de caso de estratégias de virtualização (Seção 7.7).

7.2 A camada do sistema operacional

Os usuários só ficarão satisfeitos se sua combinação de *middleware*-SO tiver bom desempenho. Em um sistema distribuído, o *middleware* pode ser executado sobre uma variedade de sistemas operacionais, sobre diferentes *hardwares*, em cada nó. O par SO-*hardware* é genericamente denominado plataforma. O SO que está sendo executado em um nó – composto por um núcleo e por serviços em nível de usuário, por exemplo,

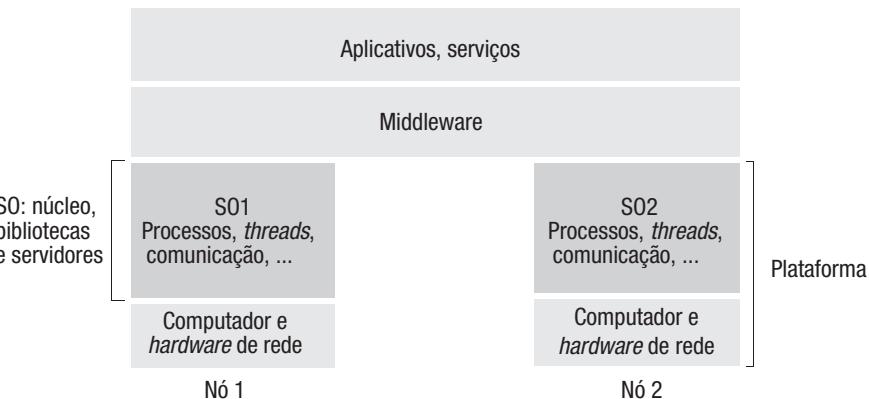


Figura 7.1 Camadas de sistema.

bibliotecas de comunicação – fornece, de acordo com seus recursos de *hardware* locais, seu próprio conjunto de abstrações para processamento, armazenamento e comunicação. O *middleware* utiliza essas abstrações para implementar seus mecanismos de invocações remotas entre objetos ou processos nos nós.

A Figura 7.1 mostra como a camada de sistema operacional, em cada um de dois nós, oferece suporte para uma camada de *middleware* comum para fornecer uma infraestrutura distribuída para aplicativos e serviços.

Nosso objetivo neste capítulo é examinar o impacto dos mecanismos do SO, em particular, sobre a capacidade do *middleware* de apresentar compartilhamento de recursos distribuído para os usuários. Os núcleos e os processos clientes e servidores neles executados são os principais componentes que nos interessam. Os núcleos e os processos servidores são os componentes que gerenciam os recursos e que os apresentam aos clientes por meio de uma interface. Desse modo, exigimos deles pelo menos o seguinte:

Encapsulamento: eles devem fornecer uma interface de serviço útil para seus recursos – isto é, um conjunto de operações que satisfaça as necessidades de seus clientes. Os detalhes, como gerenciamento de memória e dispositivos usados para implementar os recursos, devem ser ocultados dos clientes.

Proteção: os recursos exigem proteção contra acessos ilegítimos – por exemplo, os arquivos são protegidos contra leitura de usuários sem permissões de leitura e os registradores dos dispositivos de E/S são protegidos contra acessos de processos aplicativos.

Processamento concorrente: os clientes podem compartilhar recursos e acessá-los concorrentemente. Os gerenciadores de recurso são responsáveis pela transparência da concorrência.

Os clientes acessam recursos fazendo, por exemplo, invocações a métodos remotos em um objeto servidor ou chamadas de sistema ao núcleo. Denominamos a maneira de acessar um recurso encapsulado de *mecanismo de invocação*, não importando como isso seja implementado. Uma combinação de bibliotecas, núcleos e servidores pode ser empregada para realizar as seguintes tarefas relacionadas à invocação:

Comunicação: parâmetros e resultados de operação precisam ser passados por gerenciadores de recursos, via rede ou internamente ao próprio computador.

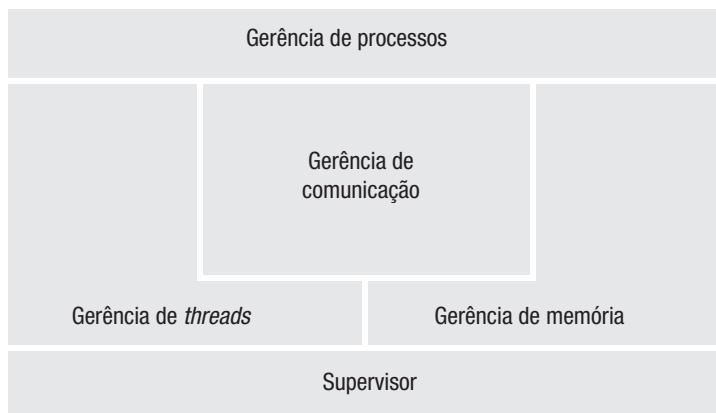


Figura 7.2 Funcionalidades básicas de um SO.

Escalonamento: quando uma operação é invocada, seu processamento deve ser agendado dentro do núcleo ou do servidor.

A Figura 7.2 mostra as funcionalidades básicas de um SO com que vamos nos preocupar: gerenciamento de processos e *threads*, gerenciamento de memória e comunicação entre processos no mesmo computador (as divisões horizontais na figura denotam dependências). O núcleo fornece grande parte dessa funcionalidade – toda ela, no caso de alguns sistemas operacionais.

O *software* do SO é projetado para ser portável entre arquiteturas de computador. Isso significa que a maior parte dele é codificada em uma linguagem de alto nível, como C, C++ ou Modula-3, e que seus recursos são dispostos em camadas para que os componentes dependentes de máquina sejam reduzidos a uma camada inferior mínima. Alguns núcleos podem ser executados em multiprocessadores de memória compartilhada, os quais estão descritos no quadro a seguir.

Multiprocessadores de memória compartilhada • Os multiprocessadores de memória compartilhada vêm equipados com vários processadores que compartilham um ou mais módulos de memória (RAM). Os processadores também podem ter sua própria memória privativa. Os multiprocessadores podem ser construídos de diversas formas [Stone 1993]. Os mais simples e mais baratos são construídos por meio da incorporação de uma placa contendo alguns processadores (2–8) em um computador pessoal.

Na arquitetura de processamento simétrico, cada processador executa o mesmo núcleo e os núcleos desempenham papéis equivalentes no gerenciamento dos recursos de *hardware*. Os núcleos compartilham suas principais estruturas de dados, como a fila de *threads* prontas para execução, mas alguns de seus dados de trabalho são privativos. Cada processador pode, simultaneamente, executar uma *thread*, acessando dados na memória compartilhada, que pode ser privativa (protegida pelo *hardware*) ou compartilhada com as demais *threads*.

Os multiprocessadores podem ser usados para muitas tarefas de computação de alto desempenho. Nos sistemas distribuídos, eles são particularmente úteis para a implementação de servidores, pois o servidor pode executar um único programa com várias *threads*, tratando simultaneamente requisições de diversos clientes – por exemplo, fornecendo acesso a um banco de dados compartilhado (veja a Seção 7.4).

Os componentes básicos do SO são os seguintes:

Gerência de processos: trata da criação de processos e das operações neles executadas. Um processo é uma unidade de gerenciamento de recursos, incluindo um espaço de endereçamento e uma ou mais *threads*.

Gerência de threads: serve para a criação de *threads*, sincronização e escalonamento. As *threads* são fluxos de execução associados aos processos e serão descritas na Seção 7.4.

Gerência de comunicação: trata da comunicação entre *threads* associadas a diferentes processos no mesmo computador. Alguns núcleos também suportam comunicação entre *threads* de processos remotos. Outros necessitam de serviços adicionais para prover comunicação externa com outras máquinas. A Seção 7.5 discutirá o projeto de sistemas de comunicação.

Gerência de memória: trata do gerenciamento da memória física e virtual. As Seções 7.4 e 7.5 descrevem a utilização das técnicas de gerenciamento de memória para cópia e compartilhamento eficiente dos dados.

Supervisor: trata do envio de interrupções, da captura das chamadas de sistema e de exceções; do controle da unidade de gerenciamento de memória e de suas caches; do processador e da unidade em ponto flutuante. No Windows, ele é conhecido como camada de abstração de *hardware*, ou HAL (*Hardware Abstraction Layer*). O leitor pode consultar Bacon [2002] e Tanenbaum [2007] para uma descrição mais completa sobre os aspectos do núcleo dependentes do computador.

7.3 Proteção

Dissemos anteriormente que os recursos exigem proteção contra acessos ilegítimos. Note que a ameaça à integridade de um sistema não é proveniente apenas de código construído de forma mal-intencionada. Um código benigno, contendo um erro ou um comportamento imprevisto, pode fazer com que parte do sistema aja incorretamente.

Para explicar o que queremos dizer com “acesso ilegítimo” a um recurso, consideremos um arquivo. Vamos supor, para esta explicação, que os arquivos abertos tenham apenas duas operações, *leitura* e *escrita*. A proteção do arquivo consiste em dois subproblemas. O primeiro é garantir que cada uma das duas operações do arquivo possa ser executada somente pelos clientes que tenham o direito de executá-la. Por exemplo, Smith, a quem pertence o arquivo, tem direitos de *leitura* e *escrita* nele. Jones só pode executar a operação de *leitura*. Um acesso ilegítimo seria se Jones conseguisse, de algum modo, executar uma operação de *escrita* no arquivo. Uma solução completa para esse subproblema de proteção de recurso em um sistema distribuído exige técnicas de criptografia e vamos deixá-la para o Capítulo 11.

O outro tipo de acesso ilegítimo, que vamos tratar aqui, é quando um cliente mal comportado consegue realizar operações diferentes das disponibilizadas por um recurso. Em nosso exemplo, isso aconteceria se Smith, ou Jones, conseguissem de algum modo executar uma operação que não fosse nem de *leitura* nem de *escrita*. Suponha, por exemplo, que Smith conseguisse acessar diretamente a variável de ponteiro do arquivo. Ele poderia, então, construir uma operação *setFilePointerRandomly*, que configuraria o ponteiro do arquivo com um número aleatório. Como essa função tem uma utilidade praticamente nula, ela nunca faria parte do conjunto de funções disponibilizadas por um sistema de arquivos.

Podemos proteger os recursos contra invocações ilegítimas, como a operação *setFilePointerRandomly*. Uma maneira é usar uma linguagem fortemente tipada, como

Sing#, uma extensão da linguagem C# usada no projeto Singularity [Hunt *et al.* 2007], ou Modula-3. Uma linguagem fortemente tipada é desenvolvida de tal forma que nenhum módulo pode acessar um módulo de destino, a menos que tenha uma referência válida para ele – ou seja, não é permitido criar um ponteiro e acessá-lo, como seria viável em C ou C++. Além disso, nesses casos, a referência ao módulo de destino só pode ser usada para realizar as invocações, a métodos ou procedimentos, que o programador do destino tornou disponíveis. Em outras palavras, não é possível alterar as variáveis do destino arbitrariamente. Em contraste, em C++, o programador pode definir um ponteiro, convertê-lo para um tipo qualquer (*casting*) e, assim, realizar quaisquer invocações e acessos.

Também é possível empregar suporte de *hardware* para proteger os módulos uns dos outros, independentemente da linguagem em que eles foram escritos. Para explorar essa possibilidade em um computador de propósito geral, é necessário contar com o auxílio do núcleo do sistema operacional.

Núcleo e proteção • O núcleo é um programa diferenciado pelo fato de que permanece carregado a partir da inicialização do sistema e seu código é executado com privilégios de acesso completos aos recursos físicos presentes em seu computador. Em particular, ele pode controlar a unidade de gerenciamento de memória e configurar os registradores do processador de modo que nenhum outro código possa acessar os recursos físicos da máquina, exceto de formas consideradas aceitáveis.

A maioria dos processadores tem, em *hardware*, um registrador de modo cuja configuração determina se instruções privilegiadas podem ser executadas, como aquelas usadas para determinar quais tabelas de proteção são correntemente empregadas pela unidade de gerenciamento de memória. O processo núcleo é executado com o processador no modo *supervisor* (privilegiado), e o núcleo providencia para que os outros processos sejam executados no modo *usuário* (não privilegiado).

O núcleo também configura *espaços de endereçamento* para proteger a si mesmo, e a outros processos, dos acessos de um processo anômalo e para fornecer aos processos uma área de memória virtual. Um espaço de endereçamento é um conjunto de intervalos de posições de memória virtual em cada um dos quais se aplica uma combinação específica de direitos de acesso à memória como, por exemplo, somente leitura ou escrita. Um processo não pode acessar posições de memória fora do seu espaço de endereçamento. Os termos *processo de usuário* ou *processo modo usuário* são normalmente usados para descrever aquele processo que é executado no modo usuário e tem um espaço de endereçamento em nível de usuário, isto é, possui direitos de acesso à memória restritos, comparado ao espaço de endereçamento do núcleo.

Quando um processo executa, pode alternar entre espaço de endereçamento de usuário e espaço de endereçamento do núcleo, dependendo se o código em execução está vinculado à aplicação ou ao núcleo. O processo chaveia do espaço de endereçamento de usuário para o espaço de endereçamento do núcleo por meio de exceções, como uma interrupção ou uma *chamada do sistema* – o mecanismo de invocação de recursos do núcleo. Uma chamada do sistema é implementada por uma instrução de máquina do tipo *TRAP*, que coloca o processador no modo supervisor e passa para o espaço de endereçamento do núcleo. Quando a instrução *TRAP* é executada, assim como acontece com qualquer tipo de exceção, o *hardware* obriga o processador a executar uma função de tratamento de exceção, fornecida pelo núcleo, para que nenhum processo possa ganhar o controle ilícito do *hardware*.

Os programas pagam um preço pela proteção. O chaveamento entre os espaços de endereçamento pode ocupar muitos ciclos do processador, e uma chamada do sistema é uma operação mais dispendiosa do que uma simples chamada de procedimento ou de método. Veremos, na Seção 7.5.1, como essas penalidades entram nos custos de uma invocação.

7.4 Processos e threads

Nos anos 80, foi descoberto que a noção tradicional do sistema operacional, de um processo que executa um único fluxo de execução, era diferente dos requisitos dos sistemas distribuídos – e também dos aplicativos mais sofisticados que utilizam um único processador, mas que exigiam concorrência de atividades interna. O problema, conforme mostraremos, é que o processo tradicional torna complicado e dispendioso o compartilhamento de recursos entre atividades relacionadas.

A solução encontrada foi aprimorar a noção de processo, para que ele pudesse ser associado a múltiplas atividades. Atualmente, um processo consiste em um ambiente de execução, junto a uma ou mais *threads*. Uma *thread* é uma abstração do sistema operacional de uma atividade (o termo é derivado da frase “fio (*thread*) de execução”). O *ambiente de execução* é a unidade de gerenciamento de recursos: um conjunto de recursos locais gerenciados pelo núcleo, aos quais suas *threads* têm acesso. Um ambiente de execução consiste, principalmente, em:

- um espaço de endereçamento;
- recursos de sincronização e comunicação entre *threads*, como semáforos e interfaces de comunicação (por exemplo, soquetes);
- recursos de nível mais alto, como arquivos e janelas abertas.

Normalmente, a criação e o gerenciamento de ambientes de execução são operações dispendiosas, mas várias *threads* podem compartilhar esses ambientes – isto é, elas podem compartilhar todos os recursos disponíveis dentro deles. Em outras palavras, um ambiente de execução representa um domínio de proteção no qual suas *threads* são executadas.

As *threads* podem ser criadas e destruídas dinamicamente, conforme necessário. O objetivo principal em se ter múltiplas *threads* de execução é maximizar o grau de execução concorrente entre operações, permitindo, assim, a sobreposição da computação com operações de entrada e saída, e possibilitando a execução simultânea de atividades em máquinas do tipo multiprocessadores. Isso pode ser particularmente útil para servidores, em que o processamento simultâneo de requisições de clientes pode reduzir a tendência de criação de gargalos de processamento. Por exemplo, uma *thread* pode processar a requisição de um cliente, enquanto uma segunda *thread*, que está atendendo a outra requisição, espera que um acesso ao disco termine.

Uma analogia para threads e processos • A maneira memorizável, talvez ligeiramente repugnante, de pensar sobre os conceitos de *threads* e ambientes de execução apareceu no grupo da USENET *comp.os.mach* e é atribuída a Chris Lloyd. Um ambiente de execução consiste em um jarro tampado com ar e o alimento dentro dele. Inicialmente, há uma mosca – uma *thread* – no jarro. Essa mosca, assim como suas descendentes, pode produzir outras moscas e matá-las. Qualquer mosca pode consumir qualquer recurso (ar ou alimento) no jarro. As moscas podem ser programadas para entrarem em uma fila, de maneira ordenada, para consumir recursos. Se não tiverem essa disciplina, elas podem se chocarumas com as outras dentro do jarro – isto é, colidir e produzir resultados imprevisíveis ao tentarem consumir os mesmos recursos de maneira não controlada. As moscas podem se comunicar com (enviar mensagens para) moscas de outros jarros, mas nenhuma pode escapar do jarro e nenhuma mosca de fora pode entrar nele. Nessa visão, um processo padrão UNIX é um único jarro com uma única mosca estéril dentro dele.

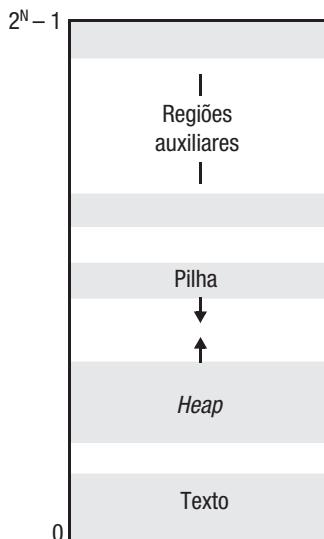


Figura 7.3 Espaço de endereçamento.

Um ambiente de execução fornece proteção contra as *threads* que estão fora dele, de modo que os dados e outros recursos nele contidos são, por padrão, inacessíveis pelas *threads* residentes em outros ambientes de execução. Entretanto, certos núcleos permitem o compartilhamento controlado de recursos, como a memória física, entre ambientes de execução residentes no mesmo computador.

Como muitos sistemas operacionais mais antigos só permitem uma *thread* por processo, às vezes vamos usar o termo *processo multithreaded* para dar ênfase aos que possuem mais de uma *thread*. Infelizmente, para confundir tudo, em alguns modelos de programação e sistemas operacionais, o termo “processo” significa executar uma *thread*. O leitor pode encontrar na literatura os termos *processo pesado*, para se referenciar aos ambientes de execução (processo), e *processo leve* para as *threads*. Veja o quadro da página anterior para uma analogia descrevendo *threads* e ambientes de execução.

7.4.1 Espaços de endereçamento

Um espaço de endereçamento é a unidade de gerenciamento da memória virtual de um processo. Normalmente, o espaço de endereçamento possui uma grande capacidade (2^{32} bytes e, às vezes, até 2^{64} bytes) e consiste em uma ou mais *regiões*, separadas por áreas não acessíveis de memória virtual. Uma região (Figura 7.3) é uma área contígua de memória virtual que é acessível para as *threads* do processo que a possui. As regiões não se sobrepõem. Note que fazemos distinção entre as regiões e seus conteúdos. Cada região é especificada pelas seguintes propriedades:

- sua extensão (menor endereço virtual e tamanho);
- permissões de leitura/escrita/execução para as *threads* do processo;
- pode crescer em ambas as direções de endereçamento (para baixo e para cima)

Consideraremos, para o restante deste capítulo, que o modelo de gerência de memória é baseado em paginação e não em segmentação. As regiões podem se sobrepor, caso aumentem de tamanho. Para possibilitar seu crescimento, são deixadas lacunas entre as regiões. Essa representação do espaço de endereços como um conjunto esparsos de regiões desmembradas é uma generalização do espaço de endereçamento do UNIX, que possui três regiões: uma região de texto fixa, não modificável, contendo código de programa; uma de região *heap*, parte da qual é inicializada por valores armazenados no arquivo binário do programa e que pode crescer em direção dos endereços virtuais mais altos; e uma região pilha, que pode aumentar em direção dos endereços virtuais mais baixos.

O uso de um número indefinido de regiões é motivado por diversos fatores. Um deles é a necessidade de suportar uma pilha separada para cada *thread*. A alocação de uma região de pilha separada para cada *thread* torna possível detectar tentativas de ultrapassar seus limites e controlar seu crescimento. A memória virtual não alocada fica além de cada região de pilha e tentativas de acessá-la causarão uma exceção (um erro de acesso inválido a página). A alternativa é alocar pilhas para *threads* no *heap*, mas ficará difícil detectar quando uma *thread* tiver ultrapassado seu limite de pilha.

Outra motivação é permitir que arquivos em geral – e não apenas as seções de texto e dados dos arquivos binários – sejam mapeados no espaço de endereçamento. Um *arquivo mapeado* é aquele que pode ser acessado como um vetor de bytes na memória. O sistema de memória virtual garante que os acessos feitos na memória se refletem no sistema de arquivos. A Seção CDK3-18.6 (no endereço [www.cdk5.net/oss/mach] – em inglês) descreve como o núcleo Mach estende a abstração de memória virtual para que as regiões possam corresponder a “objetos de memória” arbitrários – e não apenas a arquivos.

A necessidade de compartilhar memória entre processos, ou entre processos e núcleo, é outro fator que leva a regiões extras no espaço de endereçamento. Uma *região de memória compartilhada* (ou, resumidamente, *região compartilhada*) é aquela em que uma porção de memória física é mapeada para uma ou mais regiões pertencentes a vários espaços de endereçamento. Portanto, os processos acessam as mesmas posições de memória nas regiões compartilhadas, enquanto suas regiões não compartilhadas permanecem protegidas. Os usos de regiões compartilhadas incluem:

Bibliotecas: o código de uma biblioteca pode ser muito grande e desperdiçaria uma memória considerável se fosse carregada separadamente para cada processo que a usasse. Em vez disso, uma única cópia da biblioteca pode ser compartilhada, sendo mapeada como uma região nos espaços de endereçamentos dos processos que a utilizam.

Núcleo: frequentemente, o código e os dados do núcleo são mapeados em uma mesma região (compartilhada) em cada espaço de endereçamento. Quando um processo faz uma chamada de sistema, ou quando ocorre uma exceção, não há necessidade de trocar para um novo conjunto de mapeamentos de endereço.

Compartilhamento de dados e comunicação: dois processos, ou um processo e o núcleo, talvez precisem compartilhar dados para colaborarem em alguma tarefa. Pode ser consideravelmente mais eficiente os dados serem compartilhados sendo mapeados como regiões nos dois espaços de endereçamentos do que passados em mensagens entre eles. O uso de compartilhamento de região para comunicação será descrito na Seção 7.5.

7.4.2 Criação de um novo processo

Tradicionalmente, a criação de um novo processo é uma operação indivisível fornecida pelo sistema operacional. Por exemplo, a chamada de sistema *fork* do UNIX cria um processo com um ambiente de execução copiado do processo que efetuou a chamada (exceto pelo valor de retorno de *fork*). A chamada de sistema *exec* faz o processo que a realiza carregar um novo código e executá-lo.

Para um sistema distribuído, o projeto do mecanismo de criação de processos precisa levar em conta a utilização de vários computadores; consequentemente, a infraestrutura de suporte a processos é dividida em distintos serviços de sistema.

A criação de um novo processo pode ser separada em dois aspectos independentes:

- A escolha de um computador (*host*) de destino. Por exemplo, o *host* pode ser escolhido dentre os nós de um agrupamento (*cluster*) de computadores atuando como um servidor de computação, conforme apresentado no Capítulo 1.
- A criação de um ambiente de execução (e de uma *thread* inicial dentro dele).

Escolha do host • A escolha do nó em que o novo processo residirá – a decisão de alocação do processo – é uma questão de emprego de políticas. Em geral, as políticas de alocação de processos variam desde sempre executar os novos processos na estação de trabalho de seus criadores até o balanceamento da carga de processamento entre um conjunto de computadores. Eager *et al.* [1986] distinguem as seguintes categorias de políticas para o balanceamento de carga.

A *política de transferência* determina se um novo processo será criado localmente ou remotamente. Isso pode depender, por exemplo, do nó local estar pouco ou muito carregado.

A *política de localização* determina qual nó deve receber um novo processo selecionado para transferência. Essa decisão pode depender das cargas relativas dos nós, de suas arquiteturas de máquina e dos recursos especializados que eles possam ter. O sistema V [Cheriton 1984] e o Sprite [Douglis e Ousterhout 1991] fornecem comandos para os usuários executarem um programa em uma estação de trabalho ociosa (frequentemente, existem muitas delas em dado momento), escolhida pelo sistema operacional. No sistema Amoeba [Tanenbaum *et al.* 1990], o *servidor de execução* escolhe um *host* para cada processo a partir de um conjunto de processadores compartilhados. Em todos os casos, a escolha do *host* de destino é transparente para o programador e para o usuário. Entretanto, programas desenvolvidos para explorar paralelismo explícito, ou para considerar aspectos de tolerância a falhas, podem exigir uma maneira de especificar a localização do processo.

As políticas de localização de processo podem ser *estáticas* ou *adaptativas*. As primeiras operam sem considerar o estado corrente do sistema, embora sejam projetadas de acordo com as características de longo prazo esperadas pelo sistema. Elas são baseadas em uma análise matemática destinada a otimizar um parâmetro, como a quantidade de processos executados por unidade de tempo. Elas podem ser determinísticas (“o nó A sempre deve transferir processos para o nó B”) ou probabilísticas (“o nó A deve transferir processos para qualquer um dos nós B–E, aleatoriamente”). Por outro lado, as políticas adaptativas aplicam heurísticas para tomar suas decisões de alocação com base em fatores de tempo de execução imprevisíveis, como a medida da carga em cada nó.

Os sistemas de平衡amento de carga podem ser centralizados, hierárquicos ou descentralizados. No primeiro caso, existe um único componente gerenciador de carga e, no segundo, existem vários, organizados em uma estrutura em árvore. Os gerenciadores de carga reúnem informações sobre os nós e as utilizam para alocar novos processos nos nós. Nos sistemas hierárquicos, os gerenciadores tomam as decisões de alocação de

processo no nível da árvore mais baixo possível, mas, sob certas condições de carga, eles podem transferir processos uns para os outros por meio de um nó antecessor comum. Em um sistema de balanceamento de carga descentralizado, os nós trocam informações diretamente uns com os outros, para tomar decisões de alocação. O sistema Spawn [Waldspurger *et al.* 1992], por exemplo, considera que os nós são “compradores” e “vendedores” de recursos computacionais e os organiza em uma “economia de mercado” (descentralizada).

Nos algoritmos de balanceamento de carga *iniciados pela origem*, o nó que faz a criação de um novo processo é responsável por iniciar a decisão de transferência. Normalmente, ele inicia uma transferência quando sua própria carga ultrapassa um limite. Em contraste, nos algoritmos *iniciados pelo destino*, um nó cuja carga está abaixo de determinado limite anuncia sua existência para outros nós, para que os nós relativamente carregados transfiram trabalho para ele.

Os sistemas de balanceamento de carga que oferecem suporte à *migração* podem transferir carga a qualquer momento, e não apenas quando um novo processo é criado. Eles usam um mecanismo chamado *migração de processo*: a transferência de um processo em execução de um nó para outro. Milojicic *et al.* [1999] fornecem um conjunto de artigos sobre migração de processo e outros tipos de mobilidade. Embora vários mecanismos de migração de processo tenham sido construídos, eles não são amplamente usados. Isso se deve principalmente ao seu custo computacional e à grande dificuldade de extrair o estado de um processo de dentro do núcleo para movê-lo para outro.

Eager *et al.* [1986] estudaram três estratégias de balanceamento de carga e concluíram que a simplicidade é uma propriedade importante para qualquer esquema, pois o custo de coletar informações de carga e de tomar decisões de balanceamento pode comprometer a vantagem de usá-lo.

Criação de um novo ambiente de execução • Uma vez que o *host* tenha sido selecionado, a criação de um novo processo exige um ambiente de execução composto por um espaço de endereçamento com conteúdos inicializados (e talvez outros recursos, como arquivos padrão abertos).

Existem duas estratégias para definir e inicializar o espaço de endereçamento de um processo recentemente criado. A primeira estratégia é usada onde o espaço de endereçamento tem um formato definido estaticamente. Por exemplo, ele pode conter apenas uma região de texto, uma região de *heap* e uma região de pilha. Neste caso, as regiões do espaço de endereçamento são criadas a partir de uma lista que especifica seus respectivos tamanhos. As regiões do espaço de endereçamento são inicializadas a partir de um arquivo executável, ou preenchidas com valores zero, conforme apropriado.

Como alternativa, o espaço de endereçamento pode ser definido com relação a um ambiente de execução existente. No caso da semântica *fork* do UNIX, por exemplo, o processo filho, recentemente criado, compartilha fisicamente a região de texto do processo pai (seu criador) e tem cópias das regiões de *heap* e pilha. Esse esquema foi generalizado de modo que cada região do processo pai possa ser herdada (ou omitida) pelo processo filho. Uma região herdada pode ser compartilhada, ou logicamente copiada, da região do pai. Quando pai e filho compartilham uma região, os quadros (*frames*) – unidades de memória física correspondentes às páginas de memória virtual – pertencentes à região do pai são mapeados simultaneamente na região correspondente do filho.

Os núcleos Mach [Accetta *et al.* 1986] e Chorus [Rozier *et al.* 1988, 1990], por exemplo, aplicam uma otimização chamada *cópia na escrita* (*copy-on-write*), quando uma região herdada é copiada do pai. A região é copiada, mas nenhuma cópia física

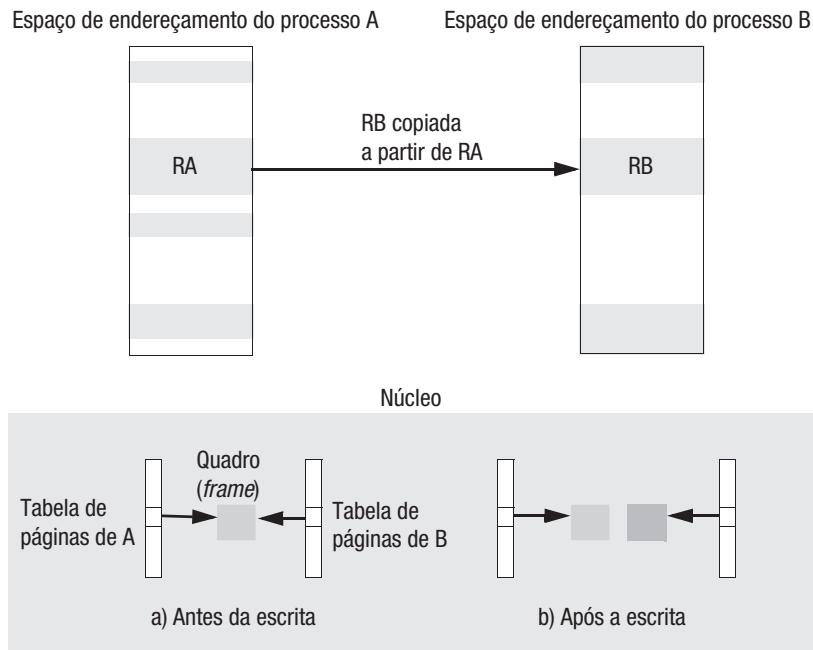


Figura 7.4 Cópia na escrita.

ocorre por padrão. Os quadros que compõem a região herdada são compartilhados entre os dois espaços de endereçamento até que um ou outro processo tente modificá-la. Neste ponto, o quadro é fisicamente copiado de uma região para outra.

A cópia na escrita é uma técnica genérica – por exemplo, ela também é usada para mensagens grandes; portanto, passaremos algum tempo explicando seu funcionamento. Vamos acompanhar um exemplo das regiões *RA* e *RB*, cuja memória é compartilhada com base em *cópia na escrita* entre dois processos, *A* e *B* (Figura 7.4). Por questões de clareza, vamos supor que o processo *A* configure a região *RA* para ser uma copiada por seu filho, o processo *B*, e que a região *RB* foi, portanto, criada no processo *B*.

Por simplicidade, supomos que as páginas pertencentes à região *A* residem na memória. Inicialmente, todos os quadros associados às regiões são compartilhados entre as tabelas de páginas dos dois processos. As páginas são inicialmente protegidas contra escrita, mesmo que possam pertencer a regiões que sejam logicamente de escrita. Se uma *thread* de um dos dois processos tentar modificar os dados, será gerada uma exceção de *hardware* chamada *erro de acesso inválido a página*. Digamos que o processo *B* tentou a escrita. A rotina de *erro de acesso inválido a página* aloca um novo quadro para o processo *B* e copia nele os dados do quadro original, byte por byte. O número de quadro antigo é substituído pelo novo número de quadro na tabela de páginas de um processo – não importa qual deles – e o número de quadro antigo é deixado na outra tabela de páginas. Cada uma das duas páginas correspondentes nos processos *A* e *B*, mapeados em quadros da memória, passam a ter autorização para escrita (modificação). Depois que tudo isso tiver acontecido, a instrução de escrita do processo *B* poderá prosseguir.

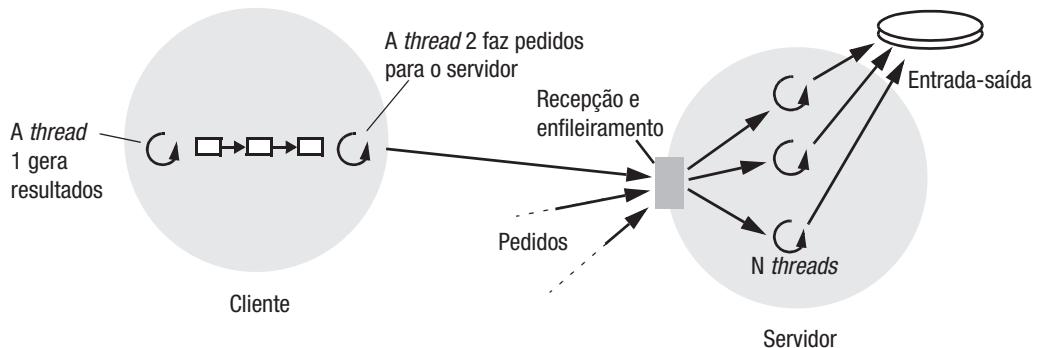


Figura 7.5 Cliente e servidor baseados em *threads*.

7.4.3 Threads

O próximo aspecto importante de um processo a considerar com mais detalhes são suas *threads*. Esta subseção examina as vantagens de permitir que os processos clientes e servidores possuam mais de uma *thread*. Então, ela discute a programação com *threads*, usando *threads Java* como um estudo de caso, e termina com projetos alternativos para implementação de *threads*.

Considere o servidor mostrado na Figura 7.5 (veremos o cliente em breve). O servidor tem um conjunto de uma ou mais *threads*, cada uma das quais retira um pedido de uma fila de requisições recebidas e o processa. Não vamos nos preocupar, por enquanto, com o modo como os pedidos são recebidos e enfileirados para as *threads*. Além disso, por simplicidade, supomos que cada *thread* execute a mesma rotina para processar os pedidos. Vamos supor que cada pedido tenha, em média, um atraso de 2 milissegundos de processamento e de 8 milissegundos de E/S (entrada/saída), quando o servidor lê um disco (não há utilização de cache). Vamos supor ainda, por enquanto, que o servidor é executado em um computador com um único processador.

Considere a *tакса de rendimento ou vazão (throughput)* máxima do servidor, medida em pedidos de clientes manipulados por segundo, para diferentes números de *threads*. Se uma única *thread* tiver de realizar todo o processamento, então o tempo gasto para manipular qualquer pedido será, em média, $2 + 8 = 10$ milissegundos; portanto, esse servidor pode manipular 100 pedidos de clientes por segundo. Todas as novas mensagens de pedido que chegam enquanto o servidor está manipulando um pedido são enfileiradas.

Agora, considere o que acontece se o servidor contiver duas *threads*. Supomos que as *threads* sejam escalonadas independentemente – isto é, uma *thread* pode ser executada quando a outra for bloqueada para E/S. Então, a *thread* número dois pode processar um segundo pedido, enquanto a *thread* número um está bloqueada e vice-versa. Isso aumenta a taxa de rendimento. Infelizmente, em nosso exemplo, por acessarem uma única unidade de disco, ambas as *threads* podem ficar bloqueadas à espera da conclusão de E/S. Se todos os pedidos de disco forem dispostos em série e demorarem 8 milissegundos cada, então o desempenho de saída máximo será de $1.000/8 = 125$ pedidos por segundo.

Suponha, agora, que seja introduzida uma cache de blocos de disco. O servidor mantém os dados que lê em *buffers* no seu espaço de endereçamento; uma *thread*, ao ler dados, primeiro examina a cache e, se os encontrar lá, evita o acesso ao disco. Se possuir uma taxa de acerto de 75%, o tempo de E/S médio por pedido será reduzido a $(0,75 \times 0 + 0,25 \times 8) = 2$ milissegundos, e a taxa de rendimento máxima aumentará para 500 pedidos por segundo.

No entanto, se o tempo de *processamento* médio para um pedido aumentou para 2,5 milissegundos, como resultado do uso de cache (leva tempo para pesquisar dados colocados na cache em cada operação), então esse valor não pode ser atingido. O servidor, limitado pelo tempo de processamento, pode agora manipular no máximo $1.000/2,5 = 400$ pedidos por segundo.

A taxa de rendimento pode ser aumentada, usando-se um multiprocessador de memória compartilhada para diminuir o gargalo de processamento. Um processo *multithreaded* é mapeado naturalmente em um multiprocessador de memória compartilhada. O ambiente de execução pode ser implementado na memória compartilhada e as múltiplas *threads* podem ser programadas para executar nos múltiplos processadores. Considere, agora, o caso em que nosso exemplo de servidor é executado em um multiprocessador com dois processadores. Como as *threads* podem ser escalonadas independentemente nos diferentes processadores, então até duas *threads* podem processar pedidos em paralelo. Como exercício, o leitor deve verificar que duas *threads* podem processar 444 pedidos por segundo; e três ou mais *threads*, limitadas pelo tempo de E/S, podem processar 500 pedidos por segundo.

Arquiteturas de servidores multithreadeds • Descrevemos como o uso de várias *threads* permite aos servidores maximizarem sua taxa de rendimento, medida como o número de pedidos processados por segundo. Para descrevermos as diversas formas de mapear pedidos em *threads* dentro de um servidor, resumimos a narrativa de Schmidt [1998], que descreve as arquiteturas baseadas em *threads* de várias implementações do ORB (Object Request Broker) do CORBA. Os ORBs processam os pedidos que chegam através de mensagens em soquetes TCP. Essas arquiteturas baseadas em *threads* são relevantes para muitos tipos de servidores, independentemente do CORBA ser usado.

A Figura 7.5 mostra uma das possíveis arquiteturas baseadas em *threads*, a *arquitetura do conjunto de trabalhadores*. Em sua forma mais simples, ao ser inicializado, o servidor cria um conjunto fixo de *threads* “trabalhadores” para processar os pedidos. O módulo identificado como “recepção e enfileiramento”, na Figura 7.5, é normalmente implementado por uma *thread* de “E/S”, que recebe pedidos de um ou mais de soquetes, ou portas, e os coloca em uma fila de pedidos para serem recuperados pelas trabalhadoras.

Às vezes, há necessidade de tratar dos pedidos com prioridades distintas. Por exemplo, um servidor Web corporativo poderia dar prioridade ao processamento dos pedidos de acordo com a classe de cliente da qual o pedido deriva [Bhatti e Friedrich 1999]. Podemos tratar as prioridades de pedidos introduzindo múltiplas filas na arquitetura de conjunto de trabalhadoras, de modo que as *threads* trabalhadoras percorram as filas na ordem decrescente de prioridade. Uma desvantagem dessa arquitetura é sua falta de flexibilidade: conforme vimos em nosso exemplo, o número de *threads* trabalhadores no conjunto pode ser pequeno demais para tratar adequadamente da taxa de chegada de pedidos. Outra desvantagem é a grande quantidade de chaveamentos de contexto entre as *threads* de E/S e as trabalhadoras, pois elas manipulam uma fila compartilhada.

Na *arquitetura thread por pedido* (Figura 7.6a), a *thread* de E/S gera uma nova *thread* trabalhadora para cada pedido, e esse trabalhador se auteterminará quando tiver processado o pedido. Essa arquitetura tem a vantagem de que as *threads* não disputam uma fila compartilhada e a taxa de rendimento é potencialmente maximizada, pois a *thread* de E/S pode criar tantos trabalhadoras quantos forem os pedidos pendentes. Sua desvantagem é a sobrecarga das operações de criação e destruição de *threads*.

A *arquitetura de thread por conexão* (Figura 7.6b) associa uma *thread* a cada conexão. O servidor cria uma nova *thread* trabalhador quando um cliente estabelece uma conexão e destrói a *thread* quando o cliente fecha a conexão. Nesse meio-tempo, o cliente pode fazer vários pedidos pela conexão, destinados a um ou mais objetos remotos. A *arquitetura de*

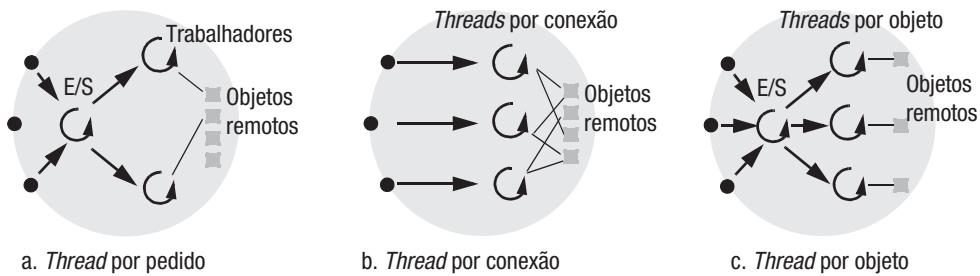


Figura 7.6 Arquiteturas baseadas em *threads* (veja também a Figura 7.5).

thread por objeto (Figura 7.6c) associa uma *thread* a cada objeto remoto. Uma *thread* de E/S recebe pedidos e os enfileira para os trabalhadores, mas desta vez há uma fila por objeto.

Em cada uma dessas duas últimas arquiteturas, o servidor se beneficia das menores sobrecargas de gerenciamento de *thread*, se comparadas à arquitetura de *thread* por pedido. Sua desvantagem é que os clientes podem sofrer atrasos quando uma *thread* trabalhador tem vários pedidos pendentes, mas outra *thread* não tem trabalho a fazer.

Schmidt [1998] descreve variações dessas arquiteturas, assim como combinações delas, e discute suas vantagens e desvantagens com mais detalhes. A Seção 7.5 descreverá um modelo de uso de *threads* diferente, no contexto das invocações dentro de uma única máquina, no qual as *threads* clientes têm acesso ao espaço de endereçamento do servidor.

Threads em clientes • As *threads* podem ser úteis para os clientes, assim como para os servidores. A Figura 7.5 também mostra um processo cliente com duas *threads*. A primeira *thread* gera resultados a serem passados para um servidor por meio de invocação a método remoto, mas não exige uma resposta. As invocações a métodos remotos normalmente bloqueiam o chamador, mesmo quando não há rigorosamente nenhuma necessidade de esperar. Esse processo cliente pode incorporar uma segunda *thread*, a qual realiza as invocações a métodos remotos e bloqueia, enquanto a primeira *thread* é capaz de continuar calculando mais resultados. A primeira *thread* coloca seus resultados em buffers, os quais são esvaziados pela segunda *thread*. Ela só é bloqueada quando todos os buffers estiverem cheios.

O caso dos clientes *multithreaded* também fica evidente no exemplo de navegadores Web. Os usuários sentem demoras substanciais enquanto páginas são procuradas; portanto, é fundamental que os navegadores manipulem vários pedidos de páginas Web paralelamente.

Threads versus múltiplos processos • A partir dos exemplos anteriores, podemos ver a utilidade das *threads*, as quais permitem que a computação seja sobreposta com E/S e, no caso de um multiprocessador, com outra computação. Entretanto, o leitor pode ter notado que a mesma sobreposição poderia ser obtida pelo uso de vários processos *single-threaded*. Por que, então, o modelo do processo *multithreaded* deve ser preferido? A resposta é dupla: as *threads* são computacionalmente mais baratas de serem criadas e gerenciadas do que os processos, e o compartilhamento de recursos pode ser obtido de forma mais eficiente entre *threads* do que entre processos, pois elas compartilham um ambiente de execução.

A Figura 7.7 mostra alguns dos principais componentes de estado que devem ser mantidos para ambientes de execução e, para *threads*, respectivamente. Um ambiente de execução tem um espaço de endereçamento; interfaces de comunicação, como os soquetes;

<i>Ambiente de execução</i>	<i>Thread</i>
Tabelas de espaço de endereçamento	Registradores internos do processador salvos
Interfaces de comunicação, arquivos abertos	Prioridade e estado da execução (como <i>BLOCKED</i>)
Semáforos, outros objetos de sincronização	Informações do tratamento da interrupção de <i>software</i>
Lista de identificadores de <i>thread</i>	Identificador do ambiente de execução
Páginas do espaço de endereçamento residentes na memória; entradas de cache em <i>hardware</i>	

Figura 7.7 Estados associados aos ambientes de execução e às *threads*.

recursos de mais alto nível, como arquivos abertos e objetos de sincronização de *threads*, como os semáforos; e, por fim, ele também lista as *threads* que possui. Uma *thread* tem uma prioridade de escalonamento, um estado de execução (como *BLOCKED* ou *RUNNABLE*), cópia dos valores dos registradores internos do processador, quando a *thread* está no estado *BLOCKED*, e o estado da *thread* relativo à execução de uma *interrupção de software*. Uma *interrupção de software* é um evento que faz uma *thread* ser interrompida (semelhante ao caso de uma interrupção de *hardware*). Se a *thread* tiver um tratador de interrupção, o controle será transferido para ela. Os sinais do UNIX são exemplos de interrupções de *software*.

A Figura 7.7 mostra que um ambiente de execução e as *threads* pertencentes a ele são associados às páginas pertencentes ao espaço de endereçamento mantido na memória principal, e os dados e instruções são mantidos em caches em *hardware*.

Podemos fazer um resumo da comparação entre processos e *threads*, como segue:

- Criar uma nova *thread* dentro de um processo existente é computacionalmente menos oneroso do que criar um processo.
- O chaveamento para uma *thread* diferente dentro de um mesmo processo é menos oneroso do que chavear entre *threads* pertencentes a processos diferentes.
- As *threads* dentro de um processo podem compartilhar dados e outros recursos conveniente e eficientemente, em comparação a processos distintos.
- Porém, além disso, as *threads* dentro de um processo não são protegidasumas das outras.

Considere o custo da criação de uma nova *thread* em um ambiente de execução existente. As principais tarefas são: alocar uma região para sua pilha e fornecer valores iniciais para os registradores internos do processador, para o estado de execução (inicialmente, ele pode ser *SUSPENDED* ou *RUNNABLE*) e para a prioridade inicial da *thread*. Como o ambiente de execução já existe, apenas um identificador para ele precisa ser posto no registro descritor de uma *thread* (o qual contém os dados necessários para gerenciar a execução da *thread*).

As sobrecargas computacionais associadas à criação de um processo são, em geral, consideravelmente maiores do que as da criação de uma nova *thread*, pois um novo ambiente de execução deve ser criado primeiro, incluindo suas tabelas de espaço de endereçamento. Anderson *et al.* [1991] citam um valor de cerca de 11 milissegundos para criar um novo processo UNIX e de cerca de 1 milissegundo para criar uma *thread* na arquitetura de processador CVAX executando o núcleo Topaz; em cada caso, o tempo medido inclui a nova entidade simplesmente realizando uma chamada a um procedimento vazio e depois terminando. Esses valores são fornecidos apenas como um parâmetro aproximado.

Quando a nova entidade realiza algum trabalho útil, em vez de chamar um procedimento vazio, existem também custos a longo prazo, os quais estão sujeitos a serem maiores para um novo processo do que para uma nova *thread* dentro de um processo existente. Em um núcleo que suporta memória virtual, o novo processo acarretará faltas de página quando dados e instruções forem referenciados pela primeira vez; inicialmente, as caches não conterão valores de dados para o novo processo e ele deverá povoar as entradas de cache enquanto for executando. Por outro lado, no caso da criação de *threads*, essas sobrecargas a longo prazo também podem ocorrer, mas tendem a ser menores. Quando a *thread* acessa código e dados que foram acessados recentemente por outras *threads* dentro do processo, ela tira proveito, automaticamente, de qualquer uso de cache ou de memória principal que tenha ocorrido anteriormente.

A segunda vantagem para o desempenho das *threads* está relacionada ao *chaveamento* entre *threads* – isto é, executar uma *thread*, em vez de outra, em determinado processador. Esse custo é o mais importante, pois ele pode acontecer muitas vezes durante a duração de uma *thread*. O chaveamento entre *threads* que compartilham o mesmo ambiente de execução é consideravelmente menos oneroso do que o chaveamento entre *threads* pertencentes a processos diferentes. As sobrecargas associadas ao chaveamento de *thread* são o escalonamento (a escolha da próxima *thread* a executar) e a troca de contexto.

Um contexto de processador compreende os valores dos registradores internos do processador, como o contador de programa e o domínio de proteção de *hardware* corrente – o espaço de endereçamento e o modo de proteção do processador (supervisor ou usuário). Uma *troca de contexto* é a transição que ocorre no chaveamento entre *threads*, ou quando uma única *thread* faz uma chamada de sistema ou, ainda, quando recebe uma exceção. Ela envolve o seguinte:

- O salvamento do estado original dos registradores internos do processador e a carga de um novo estado.
- Em alguns casos, uma transferência para um novo domínio de proteção – isso é conhecido como *transição de domínio*.

O chaveamento de *threads* que compartilham o mesmo ambiente de execução inteiramente em nível de usuário não envolve nenhuma transição de domínio e é relativamente barato. O chaveamento para o núcleo, ou para outra *thread* pertencente ao mesmo ambiente de execução por meio do núcleo, envolve transição de domínio. Portanto, o custo é maior, mas, se o núcleo for mapeado no espaço de endereçamento do processo, ele ainda será relativamente baixo. Entretanto, no chaveamento entre *threads* pertencentes a diferentes ambientes de execução, existem sobrecargas maiores. O quadro anterior explica

O problema de alias • Normalmente, as unidades de gerenciamento de memória incluem uma cache em *hardware* para acelerar a conversão entre endereços virtuais para endereços físicos, chamada de *translation lookaside buffer* (TLB). As TLBs, e também as caches de dados e instruções, ao usar endereços virtuais apresentam o *problema do alias*. O mesmo endereço virtual pode ser válido em dois espaços de endereçamentos diferentes, mas, em geral, se supõe que eles se refiram a diferentes dados físicos desses dois espaços de endereçamento. A menos que suas entradas sejam rotuladas (*tagged*) com um identificador de contexto, as TLBs e as caches não têm conhecimento disso e, portanto, podem conter dados incorretos. Assim, o conteúdo da TLB e da cache precisa ser invalidado quando ocorre um chaveamento entre espaços de endereçamento diferentes. As caches que são endereçadas fisicamente não sofrem do problema de *alias*, porém é mais comum usar endereços virtuais para caches, principalmente porque elas permitem que as pesquisas sejam sobrepostas com conversão de endereço.

as implicações do uso da cache de *hardware* para essas transições de domínio. Custos de prazos maiores, como atualizar entradas de cache de *hardware* e carregar páginas na memória principal, são mais propensos de ocorrerem quando ocorre tal transição de domínio. Os valores citados por Anderson *et al.* [1991] são de 1,8 milissegundos para o chaveamento entre processos UNIX e de 0,4 milissegundos para o núcleo Topaz fazer o chaveamento entre *threads* pertencentes ao mesmo ambiente de execução. Custos ainda mais baixos (0,04 milissegundos) são obtidos, caso as *threads* sejam chaveadas em nível de usuário. Esses valores são dados apenas como um parâmetro aproximado; eles não medem os custos de uso de cache em prazos maiores.

No exemplo anterior, do processo cliente com duas *threads*, a primeira *thread* gera dados e os passa para a segunda *thread*, a qual faz uma invocação a método remoto ou uma chamada de procedimento remota; como as *threads* compartilham um espaço de endereçamento, não há necessidade de utilizar passagem de mensagem para transmitir os dados. As duas *threads* podem acessar os dados por meio de uma variável comum. Aqui, residem a vantagem e o perigo de usar processos *multithreaded*. A conveniência e a eficiência do acesso a dados compartilhados é uma vantagem. Isso é particularmente verdade para os servidores, conforme mostrou o exemplo de dados de arquivo armazenados em cache, dado anteriormente. Entretanto, as *threads* que compartilham um espaço de endereçamento, e que não são escritos em uma linguagem fortemente tipada, não são protegidasumas das outras. Uma *thread* pode, arbitrariamente, alterar os dados usando um tipo diferente daquele usado por outra *thread*, causando um erro. Se for exigida tal proteção, então uma linguagem fortemente tipada deve ser usada, ou pode ser preferível utilizar múltiplos processos em vez de múltiplas *threads*.

Programação com threads • A programação com *threads* é concorrente, conforme estudado tradicionalmente, por exemplo, na área de sistemas operacionais. Esta seção se refere aos seguintes conceitos de programação concorrente, que são totalmente explicados por Bacon [2002]: *condição de corrida*, *seção crítica* (Bacon chama de *região crítica*), *monitor*, *variável de condição* e *semáforo*.

A maior parte dos programas baseados em *threads* é feita em uma linguagem convencional, como C, que foi estendido para ter uma biblioteca de *threads*. O pacote C *Threads*, desenvolvido para o sistema operacional Mach, é um exemplo disso. Mais recentemente, o padrão POSIX *Threads* IEEE 1003.1c-1995, conhecido como *pthreads*, foi amplamente adotado. Boykin *et al.* [1993] descrevem o C *Threads* e o *pthreads* no contexto do Mach.

Algumas linguagens fornecem suporte direto para *threads*, incluindo Ada95 [Burns e Wellings 1998], Modula-3 [Harbison 1992] e Java [Oaks e Wong 1999]. Daremos aqui uma visão geral das *threads* Java.

Assim como em qualquer implementação de *threads*, a linguagem Java fornece métodos para criá-las, destruí-las e sincronizá-las. A classe Java *Thread* inclui o construtor e os métodos de gerenciamento listados na Figura 7.8. Os métodos de sincronização, *Thread* e *Object*, aparecem na Figura 7.9.

Ciclo de vida de uma thread • Uma nova *thread* é criada na mesma máquina virtual Java (JVM) que da sua criadora, no estado *SUSPENDED*. Após se tornar *RUNNABLE* com o método *start()*, ela executa o método *run()* de um objeto fornecido em seu construtor. A JVM e as *threads* são todas executadas em um único processo no sistema operacional. As *threads* podem receber uma prioridade, de modo que as implementações Java que suportam prioridades executam uma *thread* em particular, em detrimento de qualquer outra *thread* com prioridade mais baixa. Uma *thread* termina quando retorna do método *run()* ou quando seu método *destroy()* é chamado.

Thread(ThreadGroup group, Runnable target, String name)
Cria uma nova *thread* no estado *SUSPENDED*, a qual pertencerá a *group* e será identificada como *name*; a *thread* executará o método *run()* de *target*.

setPriority(int newPriority), getPriority()
Configura e retorna a prioridade da *thread*.

run()
A *thread* executa o método *run()* de seu objeto de destino, caso ele tenha um; caso contrário, ela executa seu próprio método *run()* (*Thread* implementa *Runnable*).

start()
Muda o estado da *thread* de *SUSPENDED* para *RUNNABLE*.

sleep(long millisecs)
Passa a *thread* para o estado *SUSPENDED* pelo tempo especificado.

yield()
Passa para o estado *READY* e ativa o escalonamento.

destroy()
Termina (destrói) a *thread*.

Figura 7.8 Construtor e métodos de gerenciamento de *threads* Java.

Os programas podem gerenciar as *threads* em grupos. Toda *thread* pertence a um grupo, o qual é designado no momento de sua criação. Os grupos de *threads* são úteis quando vários aplicativos coexistem na mesma JVM. Um exemplo de uso de grupos é relacionado com a segurança: por padrão, uma *thread* de um grupo não pode executar operações de gerenciamento em uma *thread* de outro grupo.

Assim, por exemplo, uma *thread* de aplicativo não pode interromper maliciosamente uma *thread* do sistema de janelas (AWT).

Os grupos de *threads* também facilitam o controle das prioridades relativas das *threads* (nas implementações Java que suportam prioridades). Isso é útil para os navegadores que executam *applets* e para servidores Web que executam programas chamados *servlets* [Hunter e Crawford 1998], os quais criam páginas Web dinâmicas. Uma *thread* não privilegiada dentro de um *applet*, ou de um *servlet*, só pode criar uma nova *thread* que pertença ao seu próprio grupo ou a um grupo descendente criado dentro dela (as restrições exatas dependem do gerenciador de segurança – *SecurityManager* – que esteja em vigor). Os navegadores e servidores podem atribuir *threads* pertencentes aos diversos *applets* ou *servlets* a diferentes grupos, e configurar a prioridade máxima de cada grupo como um todo (incluindo os grupos descendentes). Não há meios de uma *thread* de *applet* ou *servlet* anular as prioridades do grupo configuradas pelas *threads* gerenciadoras, pois elas não podem ser modificadas por chamadas de *setPriority()*.

Sincronização de threads • A programação de um processo *multithreaded* exige bastante cuidado. Os principais problemas são o compartilhamento dos objetos e as técnicas usadas para coordenação e cooperação entre *threads*. As variáveis locais de cada *thread*, presentes nos métodos, são privativas – assim como suas pilhas. Entretanto, as *threads* não possuem cópias privativas de variáveis estáticas (classe), nem variáveis de instância de objeto.

Considere, por exemplo, as filas compartilhadas que descrevemos anteriormente nesta seção, em que as *threads* de E/S e as *threads* trabalhadores inserem e retiram

```

thread.join(long millisecs)
    Bloqueia até a thread terminar, mas não mais que o tempo especificado.

thread.interrupt()
    Interrompe a thread: a faz retornar de uma invocação a método que
    causa bloqueio, como sleep().

object.wait(long millisecs, int nanosecs)
    Bloqueia a thread até que uma chamada feita para notify(), ou notifyAll(), em object, ative a
    thread, ou que a thread seja interrompida ou, ainda, que o tempo especificado tenha decorrido.

object.notify(), object.notifyAll()
    Ativa, respectivamente, uma ou todas as threads que tenham chamado wait() em object.

```

Figura 7.9 Chamadas de sincronização de *thread* Java.

pedidos. Em princípio, condições de corrida (*race conditions*) podem surgir quando as *threads* manipulam estruturas de dados, como as filas, concorrentemente. Os pedidos enfileirados podem ser perdidos ou duplicados, a não ser que as manipulações de ponteiro pelas *threads* sejam cuidadosamente coordenadas.

A linguagem Java fornece a palavra-chave *synchronized* para os programadores designarem a construção conhecida como *monitor* para a coordenação de *threads*. Os programadores designam métodos inteiros, ou blocos de código arbitrários, como pertencentes a um monitor associado a um objeto individual. A garantia do monitor é que no máximo uma *thread* pode ser executada dentro dele em certo momento. Em nosso exemplo, poderíamos serializar as tarefas das *threads* de E/S e trabalhadores, designando os métodos *addTo()* e *removeFrom()* na classe *Queue* como métodos *synchronized*. Assim, todos os acessos às variáveis dentro desses métodos seriam realizados com exclusão mútua, com relação às invocações desses métodos.

A linguagem Java permite que as *threads* sejam bloqueadas e liberadas por meio de objetos arbitrários que atuam como variáveis de condição. Uma *thread* que precisa bloquear até que ocorra uma certa condição, chama o método *wait()* de um objeto. Todos os objetos implementam esse método, pois ele pertence à classe raiz *Object*. Outra *thread* chama *notify()* para desbloquear no máximo uma *thread*, ou *notifyAll()* para desbloquear todas as *threads* que estão esperando nesse objeto. Os dois métodos de notificação também pertencem à classe *Object*.

Como exemplo, quando uma *thread* trabalhador descobre que não existe nenhum pedido para processar, ela chama *wait()* na instância de *Queue*. Quando a *thread* de E/S adiciona, subsequentemente, um pedido na fila, ela chama o método *notify()* da fila para ativar um trabalhador.

Os métodos de sincronização Java aparecem na Figura 7.9. Além das primitivas de sincronização que já mencionamos, o método *join()* bloqueia o chamador até o término de uma outra *thread*. O método *interrupt()* é útil para ativar prematuramente uma *thread* que esteja bloqueada. Todas as primitivas de sincronização padrão, como os semáforos, podem ser implementadas em Java. Contudo, é necessário cuidado, pois as garantias do monitor de Java se aplicam somente ao código *synchronized* de um objeto; uma classe pode ter uma mistura de métodos *synchronized* e não-*synchronized*. Note também que o monitor implementado por um objeto Java tem apenas uma variável de condição implícita, enquanto, em geral, um monitor pode ter diversas variáveis de condição.

Escalonamento de threads • Há uma distinção importante entre escalonamento preemptivo e não-preemptivo de *threads*. No *escalonamento preemptivo*, uma *thread* pode ser suspensa, em qualquer ponto de sua execução, para permitir a execução de outra *thread*. No *escalonamento não-preemptivo*, uma *thread* é executada até realizar uma operação, por exemplo, uma chamada de sistema, que a bloqueie e leve ao escalonamento de uma outra *thread*.

A vantagem do escalonamento não-preemptivo é que qualquer seção de código que não contenha uma operação que possa bloqueá-la é automaticamente uma seção crítica. Assim, as condições de corrida são convenientemente evitadas. Por outro lado, as *threads* cujo escalonamento é não-preemptivo não são apropriadas para tirar proveito de um multiprocessador, pois elas são executadas exclusivamente. Deve-se tomar cuidado com seções de código de execução longa que não contenham chamadas que provoquem escalonamento. Nesses casos, talvez o programador necessite prever a inserção de chamadas a *yield()*, cuja única função é permitir que outras *threads* sejam escalonadas e executadas. As *threads* cujo escalonamento é não-preemptivo também não são convenientes para aplicativos em tempo real, nos quais eventos devem ser processados dentro um tempo limite após sua ocorrência.

Por padrão, a linguagem Java não suporta processamento em tempo real, embora existam implementações para tempo real [www.rtj.org]. Por exemplo, os aplicativos multimídia que processam dados, como voz e vídeo, têm requisitos de tempo real tanto para comunicação como para processamento (por exemplo, filtragem e compactação) [Govindan e Anderson 1991]. O Capítulo 20 examinará os requisitos de escalonamento de *threads* para tempo real. O controle de processos é outro exemplo de aplicações em tempo real. Em geral, cada aplicação em tempo real tem seus próprios requisitos de escalonamento de *threads*. Portanto, às vezes é desejável que os aplicativos implementem sua própria política de escalonamento. Para considerarmos isso, veremos agora a implementação das *threads*.

Implementação de threads • Muitos núcleos de sistemas operacionais fornecem suporte nativo para processos *multithreaded*, incluindo Windows, Linux, Solaris, Mach e Mac OS X. Esses núcleos fornecem chamadas de sistema para criação e gerenciamento de *threads* e escalonam as *threads* individualmente. Outros núcleos têm apenas a abstração de processo *single-threaded*. Nesse caso, o suporte a processos *multithreaded* é, então, implementado em uma biblioteca de procedimento ligada aos programas aplicativos. Em tais casos, o núcleo não tem conhecimento dessas *threads* em nível de usuário e, portanto, não pode escaloná-las individualmente. A própria biblioteca de suporte a *threads* realiza o escalonamento. Uma *thread* que se bloqueia, ao realizar uma chamada de sistema, bloqueia todo o processo e, portanto, todas as *threads* dentro dele. Uma forma de evitar isso é usar apenas chamadas de sistema não bloqueantes (ou assíncronas). Analogamente, a implementação da biblioteca de *threads* pode utilizar temporizadores e recursos de interrupção de *software* fornecidos pelo núcleo para conceder fatias de tempo de execução entre as várias *threads*.

Quando não é fornecido suporte do núcleo para processos *multithreaded*, a implementação de *threads* em nível de usuário apresenta os seguintes problemas:

- As *threads* pertencentes a um mesmo processo não podem tirar proveito de um multiprocessador.
- Uma *thread* que gera uma falta de página bloqueia o processo inteiro e, consequentemente, todas as *threads* dentro dele.
- As *threads* pertencentes a diferentes processos não podem ser escalonadas de acordo com um único esquema de prioridade relativa.

Por outro lado, as implementações de *threads* em nível de usuário têm vantagens significativas em relação às implementações em nível de núcleo (ou sistema):

- Certas operações de *threads* são significativamente menos dispendiosas. Por exemplo, o chaveamento entre *threads* pertencentes ao mesmo processo não envolve, necessariamente, uma chamada de sistema, que é uma tarefa relativamente onerosa no núcleo.
- Como o módulo de escalonamento das *threads* é implementado fora do núcleo, ele pode ser personalizado ou alterado de acordo com os requisitos específicos do aplicativo. Variações nos requisitos do escalonamento ocorrem, geralmente, devido às considerações específicas do aplicativo, como a natureza de tempo real do processamento de aplicações multimídia.
- Pode-se ter um número maior de *threads* do que poderia ser razoavelmente fornecido, por padrão, em um núcleo.

É possível combinar as vantagens das implementações de *threads* em nível de usuário e em nível de núcleo. Uma estratégia aplicada, por exemplo, no núcleo Mach [Black 1990], é permitir que o código em nível de usuário forneça sugestões sobre agendamentos de execução para o escalonador de *threads* do núcleo. Outra, adotada no sistema operacional Solaris 2, é uma forma hierárquica de escalonamento. Cada processo cria uma ou mais *threads* em nível de núcleo, conhecidos no Solaris como “processos leves”. Também são suportadas *threads* em nível de usuário. Um escalonador em nível de usuário atribui cada *thread* em nível de usuário a uma *thread* em nível de núcleo. Esse esquema pode tirar proveito dos multiprocessadores e se beneficia do fato de que algumas operações de criação e chaveamento entre *threads* ocorrem em nível de usuário. A desvantagem do esquema é que ainda falta flexibilidade: se uma *thread* em nível de núcleo bloqueia, todas as *threads* em nível de usuário atribuídas a ela também serão impedidas de executar, independentemente de estarem aptas a isso.

Vários projetos de pesquisa têm desenvolvido escalonamento hierárquico para oferecer mais eficiência e flexibilidade. Isso inclui o trabalho feito com ativações do escalonador [Anderson *et al.* 1991], o trabalho sobre multimídia de Govindan e Anderson [1991], o sistema operacional Psyche [Marsh *et al.* 1991], o núcleo Nemesis [Leslie *et al.* 1996] e o núcleo SPIN [Bershad *et al.* 1995]. A ideia que orienta esses projetos é que, o que um escalonamento em nível de usuário exige do núcleo não é apenas um conjunto de *threads* em nível de núcleo, nos quais se possa mapear as *threads* em nível de usuário, mas também que o núcleo notifique sobre *eventos* relevantes para a tomada de decisão. Descreveremos o projeto de ativações do escalonador para tornar isso mais claro.

O pacote FastThreads, de Anderson *et al.* [1991], é uma implementação de um sistema de escalonamento hierárquico baseada em eventos. Eles consideram como principais componentes do sistema um núcleo sendo executado em um computador com um ou mais processadores e um conjunto de programas aplicativos funcionando sobre ele. Cada processo aplicativo contém um escalonador em nível de usuário, o qual gerencia as *threads* dentro do processo. O núcleo é responsável por alocar *processadores virtuais* para os processos. O número de processadores virtuais atribuídos a um processo depende de fatores como os requisitos dos aplicativos, suas prioridades relativas e a demanda total nos processadores. A Figura 7.10a mostra um exemplo de uma máquina com três processadores, na qual o núcleo aloca um processador virtual para o processo A, executando uma tarefa de prioridade relativamente baixa, e dois processadores virtuais ao processo B. Eles são processadores *virtuais* porque o núcleo pode alocar, à medida que o tempo passa, diferentes processadores físicos para cada processo, enquanto mantém sua garantia da quantidade de processadores alocados.

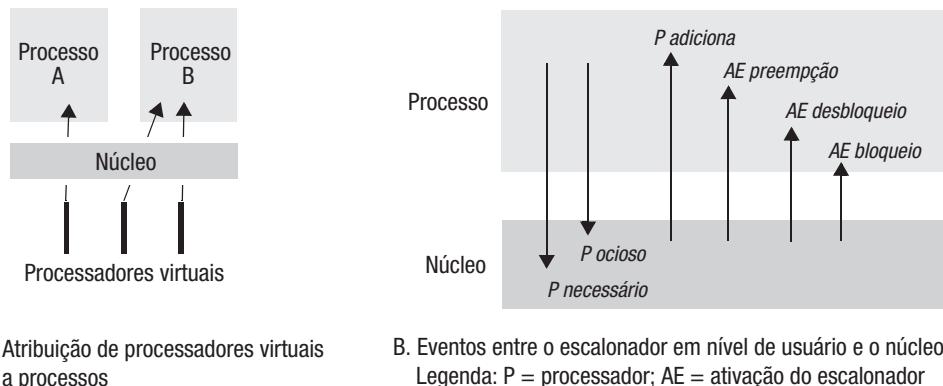


Figura 7.10 Ativações do escalonador.

O número de processadores virtuais atribuídos a um processo também pode variar. Os processos podem devolver um processador virtual de que não precisam mais; eles também podem solicitar processadores virtuais extras. Por exemplo, se o processo A tiver solicitado um processador virtual extra e B terminar, o núcleo poderá atribuir um processador para A.

A Figura 7.10b mostra que um processo notifica o núcleo quando ocorre um dos dois tipos de evento: quando um processador virtual está “ocioso” e não é mais necessário ou quando um processador virtual extra é exigido.

A Figura 7.10b também mostra que o núcleo notifica o processo, quando ocorre um dos quatro tipos de evento. Uma *ativação do escalonador* (AE) é uma chamada do núcleo para um processo, a qual notifica o escalonador do processo sobre um evento. Esse tipo de “intervenção” de uma camada inferior (o núcleo) no código de uma camada superior é, às vezes, chamada de *upcall*. O núcleo cria uma AE carregando os registradores internos de um processador físico com um contexto que o faz iniciar a execução do processo em um endereço designado pelo escalonador em nível de usuário. Assim, uma AE também é uma unidade de alocação de fatia de tempo em um processador virtual. O escalonador em nível de usuário tem a tarefa de atribuir suas *threads READY* ao conjunto de AEs que estão em execução dentro dele. O número de AEs é, no máximo, igual ao número de processadores virtuais que o núcleo atribuiu ao processo.

Os quatro tipos de evento que o núcleo notifica para o escalonador em nível de usuário (ao qual vamos nos referir simplesmente como “escalonador”) são os seguintes:

Processador virtual alocado: o núcleo atribuiu um novo processador virtual para o processo e essa é a primeira fatia de tempo nele; o escalonador pode carregar a AE com o contexto de uma *thread READY*, a qual pode, então, recomeçar a execução.

AE bloqueio: uma AE foi bloqueada no núcleo e este está usando uma nova AE para notificar o escalonador. O escalonador configura o estado da *thread* correspondente como *BLOCKED* e pode alocar uma *thread READY* para a AE que está fazendo a notificação.

AE desbloqueio: uma AE que foi bloqueada no núcleo desbloqueia e está apta para executar em nível de usuário novamente; agora, o escalonador pode devolver a *thread* correspondente para a lista *READY*. Para criar a AE que faz a notificação, o núcleo aloca um novo processador virtual para o processo, ou faz a preempção de outra

AE no mesmo processo. Neste último caso, ele também comunica o evento de preempção para o escalonador, o qual pode reavaliar sua alocação de *threads* para AEs.

AE preempção: o núcleo retirou a AE especificada do processo (embora ele possa fazer isso para alocar um processador para uma nova AE no mesmo processo); o escalonador coloca a *thread* preemptada na lista *READY* e reavalia a alocação de *threads*.

Esse esquema de escalonamento hierárquico é flexível, pois o escalonador em nível de usuário do processo pode alocar *threads* para AEs de acordo com políticas construídas sobre os eventos de baixo nível. O núcleo sempre se comporta da mesma maneira. Ele não tem influência sobre o comportamento do escalonador em nível de usuário, mas o ajuda por meio de suas notificações de evento e fornecendo o estado das *threads* bloqueadas e preemptadas. O esquema é potencialmente eficiente, pois nenhuma *thread* em nível de usuário permanece no estado *READY* caso haja um processador virtual para ela ser executada.

7.5 Comunicação e invocação

Nesta seção, abordaremos aspectos de comunicação como parte da implementação do que chamamos genericamente de *invocação* – cujo propósito é efetuar uma operação sobre um recurso e, no caso de chamadas de procedimentos remotos, uma notificação de eventos e uma invocação a métodos remotos em um espaço de endereçamento diferente de quem a executa.

Vamos abordar os problemas e conceitos de projeto do sistema operacional fazendo as seguintes perguntas sobre o SO:

- Quais primitivas de comunicação ele fornece?
- Quais protocolos ele suporta e o quanto a implementação da comunicação é aberta?
- Quais passos são dados para tornar a comunicação o mais eficiente possível?
- Que suporte é fornecido para operações com alta latência e em modo desconectado?

Abordaremos as duas primeiras questões aqui e, depois, trataremos das outras duas nas Seções 7.5.1 e 7.5.2, respectivamente.

Primitivas de comunicação • Alguns núcleos projetados para sistemas distribuídos fornecem primitivas de comunicação customizadas para os tipos de invocação descritos no Capítulo 5. O Amoeba [Tanenbaum *et al.* 1990], por exemplo, fornece *doOperation*, *getRequest* e *sendReply* como primitivas. O Amoeba, o sistema V e o Chorus fornecem primitivas de comunicação em grupo. Colocar funcionalidade de comunicação em um nível relativamente alto no núcleo tem a vantagem da eficiência na programação de aplicativos. Se, por exemplo, a camada de *middleware* fornecer RMI sobre soquetes UNIX conectados (TCP), então um cliente deverá fazer duas chamadas de sistema de comunicação (escrita e leitura de soquete) para cada invocação remota. No Amoeba, seria exigida apenas uma chamada para *doOperation*. A economia na sobrecarga de chamada de sistema está sujeita a ser ainda maior na comunicação em grupo.

Na prática, a camada de *middleware*, e não o núcleo, fornece a maior parte dos recursos de comunicação de alto nível encontrados nos sistemas atuais, incluindo RPC/RMI, notificação de evento e comunicação em grupo. Desenvolver esse *software* complexo como código em nível de usuário é muito mais simples do que desenvolvê-lo para o núcleo. Normalmente, os desenvolvedores implementam a camada de *middleware* em soquetes, fornecendo acesso aos protocolos Internet padrão – frequentemente, soquetes TCP (com conexão), mas também soquetes UDP (sem conexão). Os principais motivos

para o uso de soquetes são a portabilidade e a interação: a camada de *middleware* deve executar no máximo possível dos sistemas operacionais amplamente usados. Todos os sistemas operacionais mais comuns, como UNIX e a família Windows, fornecem APIs de soquete semelhantes, dando acesso aos protocolos TCP e UDP.

Apesar do uso difundido de soquetes TCP e UDP, fornecidos pelos núcleos comuns, pesquisas continuam sendo feitas sobre primitivas de comunicação de menor custo, computacional e latência, em núcleos experimentais. Examinaremos melhor os problemas de desempenho na Seção 7.5.1.

Protocolos e sistemas abertos • Um dos principais requisitos do sistema operacional é fornecer protocolos padrão que permitam a interligação em rede entre implementações de *middleware* sobre diferentes plataformas. Nos anos 80, vários núcleos de pesquisa incorporaram seus próprios protocolos de rede otimizados para interações de RPC – notadamente a RPC do Amoeba [van Renesse *et al.* 1989], VMTP [Cheriton 1986] e a RPC do Sprite [Ousterhout *et al.* 1988]. Entretanto, esses protocolos não foram usados fora de seus ambientes de pesquisa nativos. Em contraste, os projetistas dos núcleos Mach 3.0 e Chorus (assim como o L4 [Härtig *et al.* 1997]) decidiram deixar a escolha dos protocolos de interligação em rede totalmente aberta. Esses núcleos fornecem passagem de mensagens apenas entre processos locais e deixam o processamento do protocolo de rede para um servidor que é executado no núcleo.

Diante da necessidade diária de acesso à Internet, a compatibilidade com o TCP e o UDP é exigida dos sistemas operacionais para todos os dispositivos interligados em rede, a não ser os menores. O sistema operacional ainda é obrigado a permitir que o *middleware* tire proveito dos novos protocolos de baixo nível. Por exemplo, os usuários querem usufruir as tecnologias sem fio, como a transmissão de raios infravermelhos e rádio-frequência (RF), preferivelmente, sem ter de atualizar seus aplicativos. Isso exige que protocolos correspondentes, como o IrDA para interligação em rede por raios infravermelhos, o Bluetooth, ou o IEEE 802.11 para interligação em rede com RF, possam ser facilmente integrados ao sistema operacional.

Normalmente, os protocolos são organizados em uma *pilha* de camadas (veja o Capítulo 3). Muitos sistemas operacionais permitem que novas camadas sejam integradas estaticamente, como um “*driver*” de protocolo instalado, como, por exemplo, para o IrDA. Em contraste, a *composição dinâmica de protocolo* é uma técnica pela qual uma pilha de protocolos pode se adaptar dinamicamente para atender aos requisitos de um aplicativo em particular e para utilizar as camadas físicas que estiverem disponíveis, em razão da conectividade corrente da plataforma. Por exemplo, um navegador Web sendo executado em um computador *notebook* deve tirar proveito de um enlace sem fio remoto enquanto o usuário está em trânsito e, depois, de uma conexão Ethernet ou IEEE 802.11 mais rápida, quando volta ao escritório.

Outro exemplo de composição dinâmica de protocolo é o uso de um protocolo requisição-resposta customizado para uma camada de interligação em rede sem fio, para reduzir as latências de ida e volta. Tem-se verificado que as implementações de TCP padrão têm desempenho deficiente em redes sem fio [Balakrishnan *et al.* 1996], as quais tendem a exibir taxas mais altas de perda de pacotes do que as redes cabeadas. Em princípio, um protocolo de requisição-resposta, como o HTTP, poderia ser construído de modo a funcionar de forma mais eficiente entre nós sem fio, usando diretamente uma camada de transporte sem fio, em vez de usar uma camada TCP intermediária.

O suporte para composição de protocolo apareceu no projeto de Streams no UNIX [Ritchie 1984], no Horus [van Renesse *et al.* 1995] e no x-núcleo [Hutchinson e

Peterson 1991]. Um exemplo mais recente é a construção de um protocolo de transporte configurável CTP sobre o sistema Cactus, para a composição dinâmica de protocolos [Bridges *et al.* 2007].

7.5.1 Desempenho de uma invocação

O desempenho de uma invocação é um fator crítico no projeto de sistemas distribuídos. Quanto mais os projetistas separam a funcionalidade entre espaços de endereçamentos, mais invocações remotas são exigidas. Os clientes e os servidores podem executar milhões de operações relacionadas à invocação, de modo que pequenas frações de milissegundos contam nos custos de invocação. As tecnologias de rede continuam a melhorar, mas os tempos de invocação não têm diminuído proporcionalmente ao aumento da largura de banda da rede. Esta seção explicará como as sobrecargas de *software* frequentemente predominam sobre as sobrecargas de rede nos tempos de invocação – pelos menos para o caso de uma rede local ou intranet. Isso está em contraste com uma invocação remota pela Internet – por exemplo, na busca de um recurso Web. Na Internet, as latências de rede são altamente variáveis e, em média, relativamente altas; a vazão (*throughput*) pode ser relativamente baixa e a carga do servidor muitas vezes predomina sobre os custos de processamento por requisição. Como um exemplo de latências, Bridges *et al.* [2007] relatam tempos de ida e de volta mínimos de mensagens UDP, levando em média cerca de 400 milissegundos, pela Internet, entre dois computadores conectados em regiões geográficas dos Estados Unidos, em contraste com cerca de 0,1 milissegundo, quando computadores idênticos eram ligados por meio de uma conexão Ethernet.

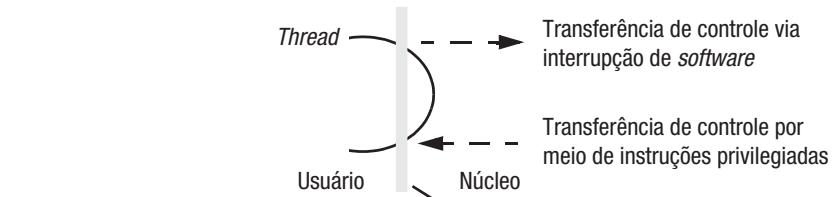
As implementações de RPC e RMI têm sido o assunto de vários estudos, devido à ampla aceitação desses mecanismos para processamento cliente-servidor de propósito geral. Muitas pesquisas têm sido feitas sobre invocações na rede e, particularmente, sobre como os mecanismos de invocação podem tirar proveito das redes de alto desempenho [Hutchinson *et al.* 1989, van Renesse *et al.* 1989, Schroeder e Burrows 1990, Johnson e Zwaenepoel 1993, von Eicken *et al.* 1995, Gokhale e Schmidt 1996]. Há também, conforme mostraremos, um importante caso especial de RPCs entre processos contidos no mesmo computador [Bershad *et al.* 1990, 1991].

Custos de uma invocação • Chamar um procedimento convencional ou um método; fazer uma chamada de sistema; enviar uma mensagem; realizar uma chamada de procedimento remoto ou uma invocação a método remoto: esses são todos exemplos de mecanismos de invocação. Cada mecanismo faz código ser executado fora do escopo do procedimento, ou objeto, que fez a chamada. Em geral, cada um envolve a comunicação de argumentos para esse código e o retorno de valores para o chamador. Os mecanismos de invocação podem ser síncronos, como, no caso das chamadas de procedimentos remotos ou convencionais, ou assíncronos.

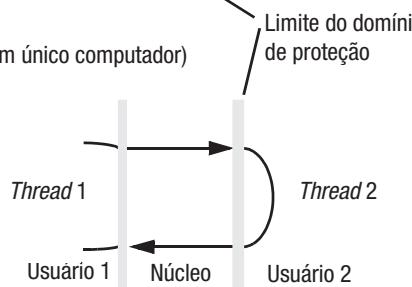
As distinções importantes, relacionadas ao desempenho, entre os mecanismos de invocação, independentemente de serem síncronos ou não, são se eles implicam uma transição de domínio (isto é, se ultrapassam um espaço de endereçamento), se envolvem comunicação em rede e se causam escalonamento e chaveamento de contexto. A Figura 7.11 mostra os casos particulares de uma chamada de sistema, de uma invocação remota entre processos contidos no mesmo computador e de uma invocação remota entre processos em diferentes nós no sistema distribuído.

Invocação via rede • Uma *RPC nula* (analogamente, uma *RMI nula*) é definida como uma RPC sem parâmetros que executa um procedimento nulo e não retorna valores.

(a) Chamada de sistema



(b) RPC/RMI (dentro de um único computador)



(c) RPC/RMI (entre computadores)

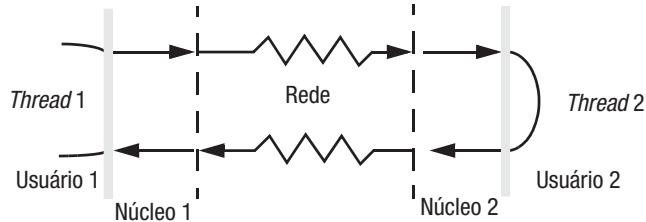


Figura 7.11 Invocações entre espaços de endereçamento.

Sua execução envolve a troca de mensagens que transportam dados do sistema, mas não dados de usuário. O tempo de uma RPC nula entre processos de usuário conectados por uma rede local é da ordem de décimos de milissegundo (veja, por exemplo, as medidas feitas por Bridges *et al.* [2007] de tempos de ida e volta de UDP, usando dois PCs Pentium 3 Xeon de 2,2GHz, em uma rede Ethernet de 100 megabits/segundo). Em comparação, uma chamada de procedimento convencional nula pode demorar uma fração de um microssegundo. Cerca de 100 bytes, no total, são passados pela rede para uma RPC nula. Com uma largura de banda bruta de 100 megabits/segundo, o tempo de transferência total da rede para esse volume de dados é de cerca de 0,01 milissegundos. Claramente, grande parte do *atraso* observado – o tempo de chamada total da RPC sentido por um cliente – leva em conta as ações do núcleo do sistema operacional e pelo código em tempo de execução da RPC em nível de usuário.

Os custos da invocação nula (RPC, RMI) são importantes, pois medem uma sobrecarga fixa, a *latência*. Os custos da invocação aumentam com o tamanho dos argumentos e dos resultados, mas, em muitos casos, a latência é significativa se comparada com o restante do atraso.

Considere uma RPC que busca um volume de dados especificado em um servidor. Ela tem um único argumento de entrada, um valor inteiro, especificando o tamanho dos

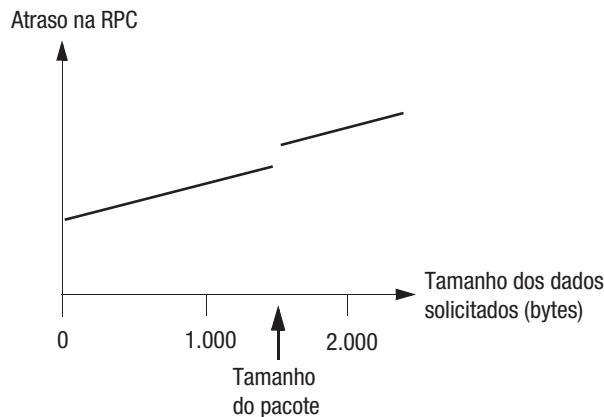


Figura 7.12 Atraso na RPC em relação ao tamanho do parâmetro.

dados solicitados, e dois argumentos de resposta, um valor inteiro, especificando sucesso ou falha (o cliente pode ter fornecido um tamanho inválido) e, quando a chamada tem êxito, um vetor de bytes do servidor.

A Figura 7.12 mostra, esquematicamente, o atraso do cliente em relação ao tamanho dos dados solicitados. O atraso é aproximadamente proporcional ao tamanho, até que este chega a um limite, quando o atraso na RPC atinge praticamente o tamanho do pacote. Além desse limite, pelo menos um pacote extra precisa ser enviado para transportar os dados a mais. Dependendo do protocolo, mais um pacote pode ser usado para confirmar esse pacote extra. Saltos no gráfico ocorrem sempre que o número de pacotes aumenta.

O atraso não é o único valor de interesse para uma implementação de RPC: o *desempenho de saída* (ou a largura de banda) da RPC também preocupa quando grandes volumes de dados são transferidos. Ele determina a taxa de transferência de dados entre computadores em uma única RPC. Se examinarmos a Figura 7.12, veremos que o desempenho de saída é relativamente baixo para pequenos volumes de dados, quando as sobrecargas de processamento fixas predominam. À medida que o volume de dados aumenta, o desempenho de saída sobe, já que essas sobrecargas se tornam globalmente menos significativas.

Lembre-se de que as etapas em uma RPC são as seguintes (a RMI envolve etapas semelhantes):

- um *stub* cliente empacota os argumentos de chamada em uma mensagem, envia a mensagem de requisição, recebe e desempacota a resposta;
- no servidor, uma *thread* trabalhador recebe a requisição que chega, ou uma *thread* de E/S recebe a requisição e a repassa para uma *thread* trabalhador; em qualquer caso, o trabalhador chama o *stub* servidor apropriado;
- o *stub* servidor desempacota a mensagem de requisição, chama o procedimento designado, empacota e envia a resposta.

A seguir, estão os principais componentes responsáveis pelo atraso da invocação remota, além dos tempos de transmissão na rede:

Empacotamento: o empacotamento e o desempacotamento, que envolvem a cópia e a conversão dos dados, tornam-se uma sobrecarga significativa à medida que o volume de dados cresce.

Cópia de dados: potencialmente, mesmo após o empacotamento, os dados da mensagem são copiados várias vezes no andamento de uma RPC:

1. entre o limite usuário–núcleo, entre o espaço de endereçamento do cliente, ou do servidor, e os *buffers* do núcleo;
2. entre cada camada de protocolo (por exemplo, RPC/UDP/IP/Ethernet);
3. entre a interface de rede e os *buffers* do núcleo.

As transferências entre a interface de rede e a memória principal normalmente são feitas por acesso direto à memória (DMA, Direct Memory Access). O processador manipula as outras cópias.

Inicialização de pacotes: envolve a construção de todos os cabeçalhos do protocolo, incluindo as somas de verificação. Portanto, o custo é proporcional, em parte, ao volume de dados enviado.

Escalonamento de threads e troca de contexto: pode ocorrer como segue:

1. várias chamadas de sistema (isto é, trocas de contexto) são feitas durante uma RPC, pois os *stubs* ativam as operações de comunicação do núcleo;
2. uma ou mais *threads* de servidor são escalonadas;
3. se o sistema operacional possui um processo gerenciador de rede a parte, cada operação de envio envolve uma troca de contexto para uma de suas *threads*.

Espera por confirmações: a escolha do protocolo de RPC pode influenciar o atraso, particularmente, quando grandes volumes de dados são enviados.

Um projeto cuidadoso do sistema operacional pode ajudar a reduzir alguns desses custos. O estudo de caso do projeto da RPC Firefly, disponível no endereço [www.cdk5.net/oss] (em inglês), mostra parte disso em detalhes, assim como técnicas que podem ser aplicadas dentro da implementação do *middleware*. Já mostramos como o suporte apropriado do sistema operacional para *threads* pode ajudar a reduzir os custos de usar *multithreading*. O sistema operacional também pode ter impacto na redução das sobrecargas de cópia na memória, por meio de recursos de compartilhamento de memória.

Compartilhamento de memória • As regiões compartilhadas (apresentadas na Seção 7.4) podem ser usadas para uma comunicação rápida entre um processo de usuário e o núcleo, ou entre processos de usuário. Os dados são trocados pela escrita e leitura na região compartilhada. Assim, os dados são passados de forma eficiente, sem necessidade de copiá-los no espaço de endereçamento do núcleo. Entretanto, chamadas de sistema e interrupções de *software* podem ser necessárias para sincronização – como quando o processo de usuário tiver escrito os dados que devem ser transmitidos, ou quando o núcleo tiver escrito dados para o processo de usuário consumir. É claro que uma região compartilhada é justificada apenas se for usada o suficiente para compensar o custo de sua configuração.

Mesmo com regiões compartilhadas, o núcleo ainda precisa copiar dados dos *buffers* na interface de rede. A arquitetura U-Net [von Eicken *et al.* 1995] permite que código em nível de usuário tenha acesso direto à própria interface de rede, para que ele possa transferir os dados para a rede sem nenhuma cópia.

Escolha do protocolo • O atraso que um cliente sente durante as interações de requisição-resposta no TCP não é necessariamente pior do que para UDP e, às vezes, é melhor, particularmente para mensagens grandes. Entretanto, é preciso cuidado ao implementar interações de requisição-resposta em um protocolo como o TCP, que não foi especifici-

camente projetado para esse propósito. Em particular, o comportamento dos *buffers* do TCP pode atrapalhar o bom desempenho, e suas sobrecargas de conexão o colocam em desvantagem, se comparado ao UDP, a não ser que mensagens suficientes sejam enviadas em uma única conexão para tornar a sobrecarga por pedido desprezível.

As sobrecargas de conexão do TCP são particularmente evidentes nas invocações Web. O HTTP 1.0, agora relativamente pouco utilizado, estabelece uma conexão TCP separada para cada invocação. Os navegadores clientes são “travados”, enquanto a conexão é estabelecida. Além disso, em muitos casos, o algoritmo de inicialização lenta (*slow-start*) do TCP tem o efeito de atrasar desnecessariamente a transferência de dados HTTP. O algoritmo de inicialização lenta opera de forma pessimista diante de um possível congestionamento da rede, permitindo que apenas uma pequena janela de dados seja enviada primeiro, antes que uma confirmação seja recebida. Nielsen *et al.* [1997] discutem como o HTTP 1.1, agora amplamente usado em lugar do HTTP 1.0, utiliza as chamadas *conexões persistentes*, que são mantidas durante o curso de várias invocações. Assim, os custos iniciais de conexão são amortizados, desde que várias invocações sejam feitas no mesmo servidor Web. É provável que isso aconteça, pois os usuários frequentemente buscam várias páginas no mesmo *site*, cada uma contendo várias imagens.

Nielsen *et al.* também descobriram que modificar o comportamento padrão dos *buffers* do sistema operacional poderia ter um impacto significativo nos atrasos de uma invocação. Frequentemente, é vantajoso reunir várias mensagens pequenas e depois enviá-las em conjunto, em vez de enviá-las em pacotes separados, devido à latência por pacote que descrevemos anteriormente. Por esse motivo, o SO não envia, necessariamente, dados pela rede imediatamente após a chamada *write()* no soquete correspondente. O comportamento padrão do SO é esperar até que seu *buffer* esteja cheio, ou usar um *timeout*, como critério para enviar os dados pela rede, na esperança de que mais dados cheguem.

Nielsen *et al.* descobriram que, no caso do HTTP 1.1, o comportamento padrão dos *buffers* do sistema operacional poderia causar atrasos significativos desnecessários, por causa dos *timeouts*. Para eliminar esses atrasos, eles alteraram as configurações do TCP no núcleo e forçaram o envio da rede nos limites das requisições HTTP. Esse é um bom exemplo de como um sistema operacional pode ajudar, ou atrapalhar, o *middleware* devido às políticas que implementa.

Invocação em um mesmo computador • Bershad *et al.* [1990] relatam um estudo mostrando que, na instalação examinada, a maior parte das trocas entre espaços de endereçamentos ocorreu dentro de um mesmo computador e não, como poderia se esperar em uma instalação cliente-servidor, entre computadores. A tendência de colocar funcionalidade de serviço dentro de servidores em nível de usuário significa que cada vez mais invocações serão feitas para um processo local. Isso é particularmente verdade à medida que o uso de cache é exageradamente explorado quando os dados necessários para um cliente tendem a ser mantidos em um servidor local. O custo de uma RPC, dentro de um mesmo computador, está crescendo em importância como parâmetro de desempenho do sistema. Essas considerações sugerem que esse caso local deve ser otimizado.

A Figura 7.11 sugere que uma invocação entre espaços de endereçamentos seja implementada dentro de um computador, exatamente como acontece entre computadores, exceto que a passagem de mensagens é local. Na verdade, frequentemente, esse tem sido o modelo implementado. Bershad *et al.* [1990] desenvolveram um mecanismo de invocação mais eficiente para o caso de dois processos em uma mesma máquina, chamado *lightweight RPC (LRPC)*, ou RPC leve. O projeto da LRPC é baseado em otimizações relativas à cópia de dados e ao escalonamento de *threads*.

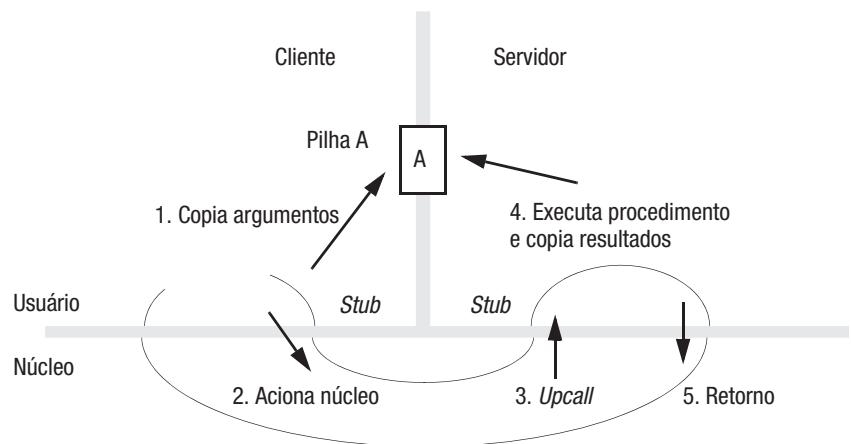


Figura 7.13 Uma chamada de procedimento remoto leve.

Primeiramente, eles notaram que seria mais eficiente usar regiões de memória compartilhadas para a comunicação cliente-servidor, com uma região diferente (privativa) entre o servidor e cada um de seus clientes locais. Tal região contém uma ou mais *pilhas A* (de argumentos) (veja a Figura 7.13). Em vez dos parâmetros do RPC serem copiados entre o núcleo e os espaços de endereçamento dos usuários envolvidos, o cliente e o servidor são capazes de passar argumentos e valores de retorno diretamente, por meio dessa pilha A. A mesma pilha é usada pelos *stubs* cliente e servidor. Na LRPC, os argumentos são copiados uma vez: quando são empacotados na pilha A. Em uma RPC equivalente, eles são copiados quatro vezes: da pilha do *stub* cliente para uma mensagem; da mensagem para um *buffer* do núcleo; do *buffer* do núcleo para uma mensagem de servidor e da mensagem para a pilha do *stub* servidor. Podem existir várias pilhas A em uma região compartilhada, pois várias *threads* do mesmo cliente podem chamar o servidor simultaneamente.

Bershad *et al.* também consideraram o custo do escalonamento das *threads*. Compare o modelo de chamada de sistema e das chamadas de procedimento remoto, na Figura 7.11. Quando ocorre uma chamada de sistema, a maioria dos núcleos não escalona uma nova *thread* para tratar dela; em vez disso, fazem uma troca de contexto na *thread* que fez a chamada para que ele manipule a chamada de sistema. Em uma RPC, um procedimento remoto pode existir em um computador diferente da *thread* cliente; portanto, uma *thread* diferente deve ser escalonada para executá-lo. No caso local, entretanto, pode ser mais eficiente a *thread* cliente – que, de outro modo, estaria no estado *BLOCKED* – chamar o procedimento no espaço de endereçamento do servidor.

Nesse caso, um servidor deve ser programado de forma diferente da maneira como descrevemos anteriormente. Em vez de configurar uma ou mais *threads*, as quais, então, “escutariam” as requisições nas portas, o servidor exporta um conjunto de procedimentos para serem chamados. As *threads* dos processos locais podem entrar no ambiente de execução do servidor, desde que comecem chamando um dos procedimentos exportados pelo servidor. Um cliente que precise invocar operações em um servidor deve, primeiro, vincular-se na interface do servidor (não mostrada na Figura 7.13). Ele faz isso por meio do núcleo, o qual notifica o servidor. Quando o servidor tiver respondido ao núcleo com uma lista de endereços dos procedimentos permitidos, o núcleo, por sua vez, responderá ao cliente.

Uma invocação aparece na Figura 7.13. Uma *thread* cliente entra no ambiente de execução do servidor, primeiramente acionando o núcleo e apresentando a ele um recurso. O núcleo verifica esse pedido e só permite a troca de contexto para um procedimento no servidor; se ele for válido, o núcleo troca o contexto da *thread* para efetuar o procedimento no ambiente de execução do servidor. Quando o procedimento no servidor retorna, a *thread* retorna para o núcleo, o qual repassa o contexto da *thread* para o ambiente de execução do cliente. Note que os clientes e os servidores empregam procedimentos *stub* para ocultar dos desenvolvedores dos aplicativos os detalhes que acabamos de descrever.

Discussão sobre a LRPC • Não há dúvidas de que a LRPC é mais eficiente do que a RPC para o caso local, desde que ocorram invocações suficientes para compensar os custos de gerenciamento de memória. Bershad *et al.* [1990] informam um fator três vezes menor para os atrasos da LRPC do que os da RPC executada de forma local.

A transparência da localização não é sacrificada na implementação de Bershad. Um *stub* cliente examina um conjunto de bits no momento da vinculação, que registra se o servidor é local ou remoto, e passa a usar LRPC ou RPC, respectivamente. O aplicativo não sabe qual é utilizada. Entretanto, a transparência da migração pode ser difícil de obter quando um recurso é transferido de um servidor local para um servidor remoto, ou vice-versa, devido à necessidade de trocar os mecanismos de invocação.

Em um trabalho posterior, Bershad *et al.* [1991] descrevem várias melhorias de desempenho, as quais são tratadas para operação em multiprocessador. As melhorias estão relacionadas ao fato de evitar o acionamento do núcleo e ao escalonamento dos processadores de maneira a evitar transições de domínio desnecessárias. Por exemplo, se um processador, alocado ao contexto de gerenciamento de memória do servidor, está ocioso no momento em que uma *thread* cliente tenta invocar um procedimento do servidor, então a *thread* deve ser transferida para esse processador. Isso evita uma transição de domínio; ao mesmo tempo, o processador do cliente pode ser reutilizado por outra *thread* no cliente. Essas melhorias envolvem uma implementação de escalonamento de *threads* em dois níveis (usuário e núcleo), conforme descrito na Seção 7.4.

7.5.2 Operação assíncrona

Já discutimos como o sistema operacional pode ajudar a camada de *middleware* a fornecer mecanismos de invocação remota eficientes. No entanto, no ambiente da Internet, os efeitos das latências relativamente altas, do desempenho de saída baixo e cargas de servidor altas, podem superar as vantagens oferecidas pelo SO. Podemos acrescentar a isso o fenômeno da desconexão e reconexão na rede, o qual pode ser considerado o causador de uma comunicação com latência extremamente alta. Os computadores móveis dos usuários não estão conectados à rede o tempo todo. Mesmo que tenham acesso remoto sem fio (por exemplo, usando comunicação por telefone celular), os usuários podem ser desconectados peremptoriamente quando, por exemplo, o trem em que estão entra em um túnel.

Uma técnica comum para anular as latências altas é a operação assíncrona, que surge em dois modelos de programação: invocações concorrentes e invocações assíncronas. Esses modelos dizem respeito, basicamente, ao escopo de *middlewares*, em vez do projeto de núcleo de sistema operacional, mas é interessante considerá-los aqui, embora estejamos examinando o assunto do desempenho de uma invocação.

Fazendo invocações concorrentes • No primeiro modelo, o *middleware* fornece apenas invocações bloqueantes, mas o aplicativo gera múltiplas *threads* para realizá-las concorrentemente.

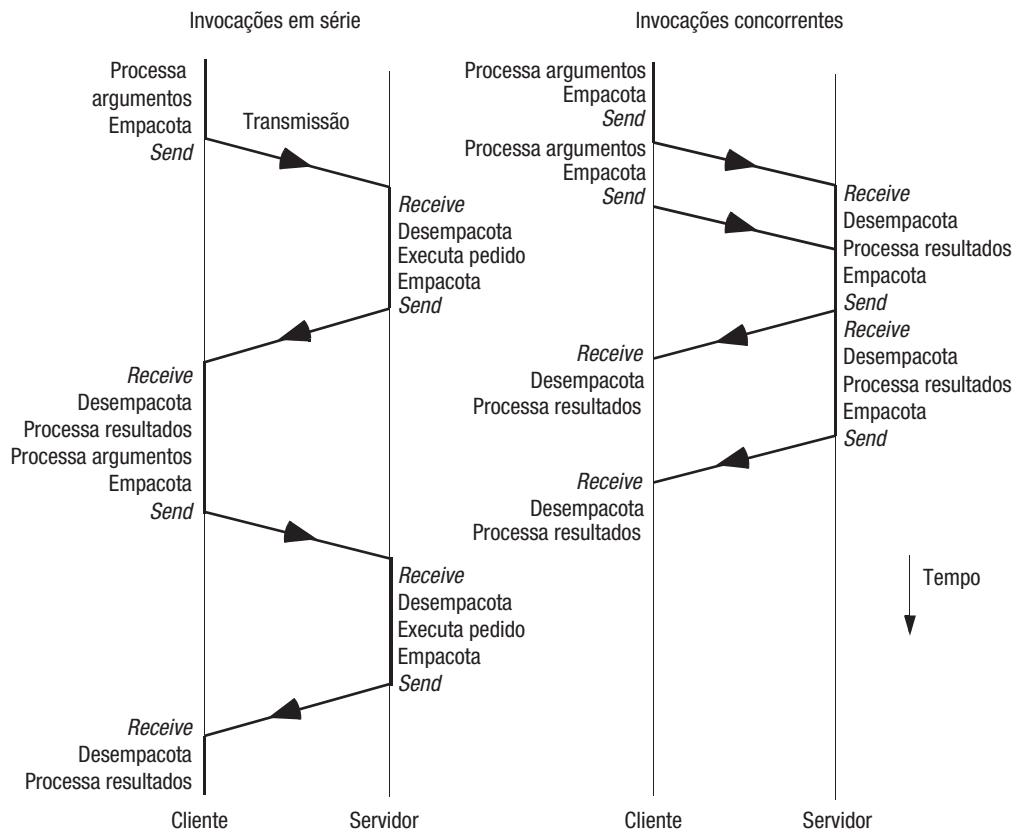


Figura 7.14 Tempos de invocações em série e concorrentes.

Um bom exemplo de tal aplicativo é um navegador Web. Normalmente, uma página Web pode conter muitas. O navegador não precisa obter as imagens em uma sequência específica, de modo que faz várias requisições concorrentes por vez. Deste modo, o tempo gasto para concluir todas as requisições de imagem normalmente é menor do que o atraso resultante de se fazer as requisições em série. Não apenas o atraso total da comunicação é menor, em geral, como também o navegador pode sobrepor a computação da renderização da imagem com a comunicação.

A Figura 7.14 mostra as vantagens em potencial do fato de entrelaçar invocações (como as requisições HTTP) entre um cliente e um único servidor, em uma máquina com um só processador. No caso em série, o cliente empacota os argumentos, chama a operação *send* e espera até que a resposta do servidor chegue – depois disso, executa a operação *receive*, desempacota e, em seguida, processa os resultados. Após disso, ele pode fazer a segunda invocação.

No caso concorrente, a primeira *thread* cliente empacota os argumentos e chama a operação *Send*. Então, a segunda *thread* faz imediatamente a segunda invocação. Cada *thread* espera para receber seus resultados. O tempo total gasto provavelmente é menor do que no caso em série, como mostra a figura. Vantagens semelhantes se aplicam, caso as *threads* clientes façam pedidos concorrentes para vários servidores; e se o cliente for

executado em um multiprocessador, então uma taxa de rendimento ainda maior é totalmente possível, pois o processamento das duas *threads* também pode ser sobreposto.

Voltando ao caso particular do protocolo HTTP, o estudo de Nielson *et al.* [1997], a que nos referimos acima, também mediou os efeitos das invocações HTTP 1.1 concorrentes entrelaçadas (as quais eles chamam de *pipeline*) sobre conexões persistentes. Eles descobriram que o uso de *pipeline* reduziu o tráfego da rede e que pode trazer vantagens de desempenho para os clientes, desde que o sistema operacional forneça uma interface conveniente para esvaziar os *buffers* e, assim, modificar o comportamento padrão do TCP.

Invocações assíncronas • Uma *invocação assíncrona* é aquela que é feita com o auxílio de uma chamada não bloqueante, a qual retorna assim que a mensagem de requisição da invocação tenha sido criada e esteja pronta para envio.

Às vezes, o cliente não exige nenhuma resposta (exceto, talvez, uma indicação de falha, caso o computador de destino não possa ser atingido). Por exemplo, as invocações *sentido único* (*one-way*) do CORBA têm semântica *talvez*. Se necessário, o cliente usa uma chamada separada para receber os resultados da invocação. Por exemplo, o sistema de comunicação Mercury [Liskov e Shrira 1988] suporta invocação assíncrona. Uma operação assíncrona retorna um objeto chamado *promise* (promessa). Quando a invocação tem êxito, ou é considerada falha, o sistema Mercury coloca o *status* e os valores de retorno na promessa. O chamador usa a operação *claim* (reclamação) para obter os resultados da promessa. A operação *claim* bloqueia até que a promessa esteja pronta, quando, então, retorna os resultados ou exceções da chamada. A operação *ready* (pronto) está disponível para testar uma promessa sem bloquear – ela retorna um valor *true* ou *false*, de acordo com a promessa estar pronta ou bloqueada.

Invocações assíncronas persistentes • Os mecanismos de invocação assíncronos tradicionais, como as invocações do Mercury e as invocações *de sentido único* do CORBA, são implementados em fluxos TCP e falham, caso seja desfeito o fluxo, isto é, se o enlace de rede cair ou se o computador destino falhar.

No entanto, uma forma mais desenvolvida do modelo de invocação assíncrona, que chamamos de *invocação assíncrona persistente*, está se tornando cada vez mais relevante por causa da operação em modo desconectado. Esse modelo é semelhante ao Mercury em termos das operações de programação que oferece, mas a diferença está em sua semântica de falhas. Um mecanismo de invocação convencional (síncrono ou assíncrono) é projetado para falhar após a ocorrência de determinado número de *timeouts*. Entretanto, *timeouts* de curto prazo frequentemente não são apropriados quando ocorrem desconexões ou latências muito altas.

Um sistema de invocação assíncrona persistente tenta, indefinidamente, realizar a invocação até obter êxito ou falha, ou até que o aplicativo cancele a invocação. Um exemplo é a QRPC (Queued RPC – RPC enfileirada) do *toolkit Rover* para acesso móvel à informação [Joseph *et al.* 1997].

Conforme seu nome sugere, a QRPC enfileira as requisições de invocação enviadas em um *log* estável, enquanto não há conexão de rede, e agenda seu envio na rede para os servidores, quando há uma conexão. Analogamente, ela enfileira os resultados da invocação, no que podemos considerar a “caixa de correio” de invocação do cliente, até que este reconecte e os colete. As requisições e resultados podem ser compactados ao serem enfileirados, antes de sua transmissão, por uma rede de baixa largura de banda.

A QRPC pode tirar proveito de diferentes enlaces de comunicação para enviar uma requisição de invocação e receber a resposta. Por exemplo, uma requisição po-

deria ser enviada por meio de um enlace de dados de celular, enquanto o usuário estivesse em trânsito, e depois a resposta seria enviada por um enlace Ethernet, quando o usuário conectasse seu dispositivo na intranet corporativa. Em princípio, o sistema de invocação pode armazenar os resultados perto do próximo ponto de conexão esperado do usuário.

O escalonador de rede do cliente funciona de acordo com vários critérios e não envia as invocações necessariamente na ordem FIFO. Os aplicativos podem atribuir prioridades para invocações individuais. Quando uma conexão se torna disponível, a QRPC avalia sua largura de banda e o custo de sua utilização. Ela envia primeiro as invocações de alta prioridade, e pode não enviar todas, caso o enlace seja lento e dispendioso (como uma conexão remota sem fio), supondo que um enlace mais rápido e barato, como um Ethernet, acabará por se tornar disponível. Analogamente, a QRPC leva a prioridade em conta ao buscar resultados das invocações na caixa de correio em um enlace de largura de banda baixa.

A programação com um sistema de invocação assíncrono (persistente ou não) levanta o problema de como os usuários podem continuar usando os aplicativos em seus dispositivos clientes, enquanto os resultados das invocações ainda não são conhecidos. Por exemplo, o usuário pode estar se perguntando se teve êxito na atualização de um parágrafo em um documento compartilhado, ou se alguém fez uma atualização conflitante, como a exclusão do parágrafo. O Capítulo 18 examinará esse problema.

7.6 Arquiteturas de sistemas operacionais

Nesta seção, examinaremos uma arquitetura de núcleo de sistema operacional conveniente para sistemas distribuídos. Adotaremos uma estratégia de obedecer aos primeiros princípios, começando com o requisito de sistema aberto e, tendo isso em mente, examinar, depois, as principais arquiteturas de núcleo que propusemos.

Um sistema distribuído aberto deve tornar possível:

- Em cada computador, executar apenas o *software* necessário para desempenhar sua função particular na arquitetura do sistema distribuído – os requisitos do *software* podem variar entre, por exemplo, telefones móveis e computadores servidores, e carregar módulos redundantes desperdiçaria recursos de memória.
- Permitir que o *software* (e o computador) que está implementando um serviço em particular seja trocado independentemente dos outros recursos.
- Possibilitar que sejam fornecidas alternativas para o mesmo serviço, quando isso é necessário para atender a diferentes usuários ou aplicativos.
- Introduzir novos serviços sem prejudicar a integridade dos já existentes.

A separação dos *mecanismos* de gerenciamento de recursos das *políticas* de gerenciamento de recursos, que variam de um aplicativo para outro e de um serviço para outro, tem sido um princípio fundamental no projeto de sistemas operacionais há muito tempo [Wulf *et al.* 1974]. Por exemplo, dissemos que um sistema de escalonamento ideal forneceria mecanismos que permitiriam um aplicativo multimídia, como a videoconferência, atender a sua demanda de tempo real, e coexistiria com uma aplicação que não usa tempo real, como a navegação Web.

De preferência, o núcleo forneceria apenas os mecanismos mais básicos nos quais as tarefas de gerenciamento de recursos gerais seriam executadas em um nó. Os módulos servidores seriam carregados dinamicamente, conforme necessário, para implementar as

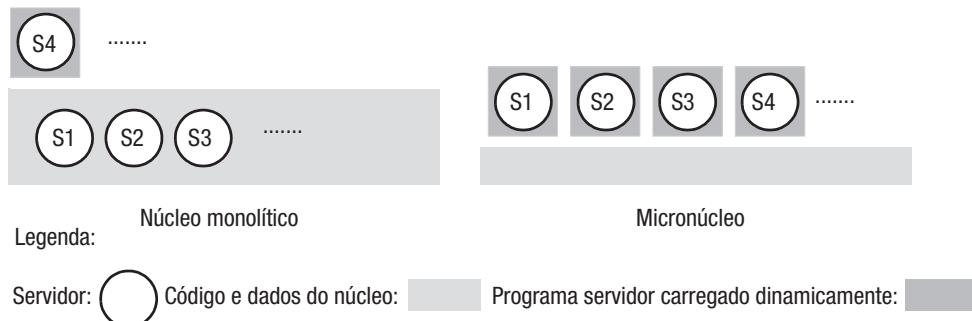


Figura 7.15 Núcleo monolítico e micronúcleo.

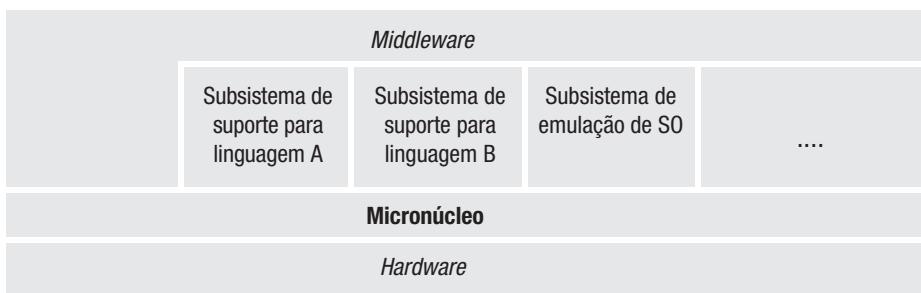
políticas de gerenciamento de recursos necessário aos aplicativos que estivessem correntemente em execução.

Núcleos monolíticos e micronúcleos • Existem dois exemplos importantes de projeto de núcleo de sistemas operacionais: *monolítico* e *micronúcleo*. Esses projetos diferem, principalmente, na decisão sobre qual funcionalidade pertence ao núcleo e qual é deixada para os processos servidores que podem ser carregados dinamicamente para execução. Embora os micronúcleos não sejam amplamente usados, é instrutivo entender suas vantagens e desvantagens comparadas aos núcleos normais encontrados atualmente.

O núcleo do sistema operacional UNIX tem sido chamado de *monolítico* (veja a definição no quadro a seguir). Esse termo se destina a sugerir que o núcleo é *um bloco maciço* – ele executa todas as funções básicas do sistema operacional e ocupa alguns megabytes de código e dados – e que *não é diferenciado* – ele é codificado de maneira não modular. O resultado é que, em grande parte, ele é *intratável*: é difícil alterar qualquer componente de *software* individual para adaptá-lo aos requisitos variáveis. Outro exemplo de núcleo monolítico é o do sistema operacional de rede Sprite [Ousterhout *et al.* 1988]. Um núcleo monolítico pode conter alguns processos servidores que são executados dentro de seu espaço de endereçamento, incluindo servidores de arquivos e alguma capacidade de interligação em rede. O código executado por esses processos faz parte da configuração padrão do núcleo (veja a Figura 7.15).

Em contraste, no caso de um projeto de micronúcleo, o núcleo fornece apenas as abstrações mais básicas, principalmente as de espaços de endereçamento, *threads* e comunicação local entre processos; todos os demais serviços de sistema são fornecidos por servidores carregados dinamicamente, mais precisamente, apenas nos computadores do sistema distribuído que necessitam deles (Figura 7.15). Os clientes acessam esses serviços de sistema usando os mecanismos de invocação baseados em mensagens ao núcleo.

Monolítico • O Chambers 20th Century Dictionary dá a seguinte definição para **monólito** e **monolítico**. **monólito**, s. um pilar ou coluna feita de uma só pedra: tudo que se pareça um monólito em uniformidade, solidez ou em tratamento. – *adj.* monolítico pertencente ou parecido com um monólito: de um estado, uma organização etc., completamente maciço e não diferenciado: intratável por esse motivo.



O micronúcleo suporta *middleware* por meio de subsistemas.

Figura 7.16 A função do micronúcleo.

Dissemos anteriormente que os usuários podem não se interessar por sistemas operacionais que não executam seus aplicativos. No entanto, além da característica extensível, os projetistas de micronúcleo têm outro objetivo: a emulação de sistemas operacionais padrão, como o UNIX [Armand *et al.* 1989, Golub *et al.* 1990, Härtig *et al.* 1997].

O posicionamento do micronúcleo – em sua forma mais geral – no projeto de sistema distribuído global é mostrado na Figura 7.16. O micronúcleo aparece como uma camada entre a camada de *hardware* e a camada composta pelos principais componentes de sistema, chamados de *subsistemas*. Se o desempenho for o principal objetivo, em vez da portabilidade, o *middleware* pode usar diretamente os recursos do micronúcleo. Caso contrário, ele utilizará um subsistema de suporte para uma linguagem em tempo de execução, ou uma interface de sistema operacional de nível mais alto, fornecida por um subsistema de simulação do sistema operacional. Cada um deles, por sua vez, é implementado por uma combinação de procedimentos de biblioteca vinculados aos aplicativos e por um conjunto de serviços executados no micronúcleo.

Pode haver mais de uma interface de chamada de sistema – mais de um “sistema operacional” – apresentada ao programador em uma mesma plataforma. Um exemplo é a implementação do UNIX e do OS/2 no núcleo do sistema operacional distribuído Mach. Note que a simulação do sistema operacional é diferente de máquinas virtuais. (Veja a Seção 7.7.)

Comparação • As principais vantagens de um sistema operacional baseado em micronúcleo são sua capacidade de extensão e sua capacidade de impor modularidade por meio de limites de proteção de memória. Além disso, é mais provável que um núcleo relativamente pequeno esteja menos sujeito a erros do que um maior e mais complexo.

A vantagem de um projeto monolítico é a relativa eficiência com que as operações podem ser invocadas. As chamadas de sistema podem ser mais dispendiosas do que os procedimentos convencionais, mas, mesmo usando as técnicas que examinamos na seção anterior, uma invocação entre espaços de endereçamento em nível de usuário no mesmo nó é ainda mais dispendiosa.

A falta de estrutura nos projetos monolíticos pode ser evitada pelo uso de técnicas de engenharia de *software*, como a disposição em camadas (usada no MULTICS [Organić 1972]) ou o projeto orientado a objetos (usado, por exemplo, no Choices [Campbell *et al.* 1993]). O Windows emprega uma combinação de ambos [Custer 1998], permanece “maciço” e a maior parte de sua funcionalidade não é feita para ser substituída rotineiramente. Mesmo um núcleo grande, modularizado, pode ser difícil de manter; e, além

disso, normalmente, eles fornecem suporte limitado para um sistema distribuído aberto. Quando os módulos são executados dentro de um mesmo espaço de endereçamento, usando-se uma linguagem como C ou C++, que gera código eficiente, mas que permite acessos arbitrários aos dados, é possível que o rigorismo de modularidade seja “quebrado” por programadores que buscam implementações eficientes, e que um erro em um módulo corrompa os dados de outro.

Algumas estratégias mistas • Dois dos micronúcleos originais, Mach [Acetta *et al.* 1986] e Chorus [Rozier *et al.* 1990], iniciaram executando servidores apenas como processos de usuário. A modularidade, então, é imposta pelo *hardware*, por meio de espaços de endereçamento. Quando os servidores necessitam fazer acesso direto ao *hardware*, isso pode ser viabilizado por meio de chamadas de sistema especiais fornecidas para esses processos privilegiados, as quais mapeiam registradores de dispositivo e *buffers* em seus espaços de endereçamentos. O núcleo transforma interrupções em mensagens, permitindo, assim, que os servidores em nível de usuário as tratem.

Devido aos problemas de desempenho, os projetos de micronúcleo Chorus e Mach mudaram para permitir que os servidores fossem carregados dinamicamente no espaço de endereçamento do núcleo ou em nível de usuário. Em cada caso, os clientes interagem com os servidores usando as mesmas chamadas de comunicação entre processos. Assim, um desenvolvedor pode depurar um servidor em nível de usuário e, então, quando o desenvolvimento for julgado concluído, permitir que o servidor seja executado dentro do espaço de endereçamento do núcleo para otimizar o desempenho do sistema. Entretanto, neste caso, tal servidor ameaçará a integridade do sistema, caso se descubra que ele ainda contém erros.

O projeto do sistema operacional SPIN [Bershad *et al.* 1995] refina o problema do compromisso entre eficiência e proteção, empregando recursos de linguagem na proteção. O núcleo e todos os módulos carregados dinamicamente, e enxertados no núcleo, são executados dentro de um único espaço de endereçamento. No entanto, todos são escritos em uma linguagem fortemente tipada (Modula-3), de modo que podem ser mutuamente protegidos. Os domínios de proteção dentro do espaço de endereçamento do núcleo são estabelecidos com o uso de espaços de nomes protegidos. Nenhum módulo enxertado no núcleo pode acessar um recurso, a não ser que tenha sido passada uma referência para ele; e a linguagem Modula-3 impõe a regra de que uma referência só pode ser usada para efetuar as operações permitidas pelo programador.

Em uma tentativa de minimizar as dependências entre os módulos do sistema, os projetistas do SPIN escolheram um modelo baseado em eventos como mecanismo de interação entre os módulos enxertados no espaço de endereçamento do núcleo (veja na Seção 6.3 uma discussão sobre a programação baseada em eventos). O sistema define um conjunto de eventos básicos, como a chegada de um pacote da rede, interrupções de relógio, ocorrências de falta página e alterações do estado das *threads*. Os componentes do sistema funcionam registrando-se como tratadores dos eventos que os afetam. Por exemplo, um escalonador se registrarria para tratar de eventos semelhantes àqueles que estudamos no sistema de ativações do escalonador, na Seção 7.4.

Sistemas operacionais como o Nemesis [Leslie *et al.* 1996] exploram o fato de que, mesmo em nível do *hardware*, um espaço de endereçamento não é necessariamente um único domínio de proteção. O núcleo coexiste em um único espaço de endereçamento com todos os módulos de sistema carregados dinamicamente e com os aplicativos. Quando carrega um aplicativo, o núcleo coloca o código e os dados do aplicativo em regiões escolhidas entre as que estão disponíveis no momento da execução. O advento dos pro-

cessadores com endereçamento de 64 bits tornou os sistemas operacionais de espaço de endereçamento único particularmente atraentes, pois eles suportam espaços de endereçamento muito grandes, que podem acomodar muitos aplicativos.

O núcleo de um sistema operacional de espaço de endereçamento único configura os atributos de proteção em regiões individuais, dentro desse espaço, de forma a restringir o acesso do código em nível de usuário. O código em nível de usuário ainda é executado com o processador em um contexto de proteção em particular (determinado pelas configurações do processador e da unidade de gerenciamento de memória), o qual fornece acesso total a suas próprias regiões e, para outros, somente acesso compartilhado, seletivamente. A economia de um único espaço de endereçamento, comparada ao uso de vários, é que o núcleo nunca precisa esvaziar as caches ao implementar uma transição de domínio.

Alguns projetos de núcleo posteriores, como o L4 [Härtig *et al.* 1997] e o Exonúcleo [Kaashoek *et al.* 1997], adotam a abordagem do que descrevemos como “micronúcleos”, mas ainda contêm políticas demais. O L4 é um projeto de micronúcleo de “segunda-geração” que obriga os módulos de sistema carregados dinamicamente a serem executados em espaços de endereçamento em nível de usuário, mas otimiza a comunicação entre processos para compensar os custos envolvidos nisso. Ele diminui grande parte da complexidade do núcleo delegando o gerenciamento de espaços de endereçamento para servidores em nível de usuário. O Exonúcleo adota uma estratégia bastante diferente, empregando bibliotecas em nível de usuário, em vez de servidores, para fornecer extensões adicionais. Ele fornece alocação protegida de recursos de nível extremamente baixo, como blocos de disco, e considera que todas as outras funcionalidades de gerenciamento de recursos – até o sistema de arquivos – sejam vinculadas aos aplicativos como bibliotecas.

Nas palavras de um projetista de micronúcleo [Liedtke 1996], “a história do micronúcleo está repleta de boas ideias e becos escuros”. Conforme veremos na próxima seção, a necessidade de suportar vários subsistemas, e também de impor proteção entre eles, é agora satisfeita pelo conceito de virtualização, que tem substituído as estratégias de micronúcleo como principal inovação no projeto de sistemas operacionais.

7.7 Virtualização em nível de sistema operacional

A virtualização é um conceito importante nos sistemas distribuídos. Já vimos uma aplicação da virtualização no contexto da interligação em rede, na forma de redes de sobreposição (veja a Seção 4.5), oferecendo suporte para classes específicas de aplicações distribuídas. A virtualização também é aplicada no contexto dos sistemas operacionais; na verdade, é nesse contexto que ela tem maior impacto. Nesta seção, examinaremos o que significa aplicar virtualização em nível de sistema operacional (virtualização de sistema) e também apresentaremos um estudo de caso do Xen, um importante exemplo de virtualização em nível de sistema.

7.7.1 Virtualização

O objetivo da virtualização é fornecer várias máquinas virtuais (imagens virtuais do *hardware*) sobre a arquitetura de máquina física subjacente, com cada máquina virtual executando uma instância separada do sistema operacional. O conceito se origina da observação de que as arquiteturas de computador modernas têm o desempenho necessário para suportar uma grande quantidade de máquinas virtuais e recursos multiplexados entre elas. Várias instâncias de um mesmo sistema operacional ou diversos sistemas operacionais diferentes podem ser executadas nas máquinas virtuais. O sistema de virtualização aloca o

processador (ou processadores) físico e outros recursos de uma máquina física entre todas as máquinas virtuais que suporta.

Historicamente, processos eram usados para compartilhar o processador e outros recursos entre várias tarefas, executando em nome de um ou de vários usuários. A virtualização surgiu mais recentemente e agora é comumente utilizada para esse propósito. Ela oferece vantagens com relação à segurança e à separação clara de tarefas, e com relação à alocação e à cobrança de cada usuário, utilizando recursos de uma forma mais precisa do que pode ser obtida com processos executando em um único sistema.

Para se entender completamente a motivação da virtualização, é útil considerar diferentes casos de uso da tecnologia:

- Em computadores servidores, uma organização atribui cada serviço que oferece a uma máquina virtual e, então, aloca da melhor forma as máquinas virtuais a servidores físicos. Ao contrário dos processos, as máquinas virtuais podem ser migradas para outras máquinas físicas de forma bastante simples, proporcionando flexibilidade ao gerenciamento da infraestrutura de servidores. Essa estratégia tem o potencial de reduzir o investimento em computadores servidores e de reduzir o consumo de energia, uma questão importante para grandes fazendas de servidores (*server farms*).
- A virtualização é bastante relevante no fornecimento de *computação em nuvem*. Conforme descrito no Capítulo 1, a computação em nuvem adota um modelo em que armazenamento, computação e objetos de nível mais alto construídos sobre eles são oferecidos como serviço. Os serviços oferecidos variam desde aspectos de baixo nível, como a infraestrutura física (chamada de IaaS, *Infrastructure as a service*, ou *Infraestrutura como serviço*), até Plataformas de *software*, como o Google App Engine, apresentado no Capítulo 21, (PaaS, *Platform as a service*, ou *Plataforma como serviço*) e serviços arbitrários em nível de aplicativo (SaaS, *Software as a service*, ou *Software como serviço*). De fato, o primeiro é possibilitado diretamente pela virtualização, permitindo aos usuários da nuvem ter uma ou mais máquinas virtuais para uso próprio.
- Os desenvolvedores de soluções de virtualização também são motivados pela necessidade das aplicações distribuídas de criar e destruir máquinas virtuais prontamente e com pouca sobrecarga. Isso é exigido em aplicações que precisam processar recursos dinamicamente, como nos jogos *online* para vários jogadores ou em aplicações multimídia distribuídas, conforme apresentado no Capítulo 1 [Whitaker *et al.* 2002]. O suporte para essas aplicações pode ser melhorado com a adoção de políticas de alocação de recursos apropriadas para atender aos requisitos da qualidade do serviço das máquinas virtuais.
- Um caso muito diferente surge no fornecimento de acesso conveniente a vários ambientes de sistema operacional distintos em um único computador *desktop*. A virtualização pode ser usada para fornecer vários tipos de sistema operacional em uma única arquitetura física. Por exemplo, em um computador Macintosh OS X, o monitor de máquina virtual Parallels *Desktop* permite que um sistema Windows ou Linux seja instalado e coexista com o OS X, compartilhando os recursos físicos subjacentes.

A virtualização é implementada por uma camada fina de *software* sobre a arquitetura de máquina física subjacente; essa camada é chamada de *monitor de máquina virtual* ou *hipervisor*. Esse monitor de máquina virtual fornece uma interface rigorosamente baseada na arquitetura física subjacente. Mais precisamente, na *virtualização total*, ou *completa*, o monitor de máquina virtual oferece uma interface idêntica à arquitetura física subjacente. Isso traz a vantagem de os sistemas operacionais existentes poderem ser executados de forma transparente e sem modificação no monitor de máquina virtual.

Contudo, a experiência tem mostrado que, em muitas arquiteturas de computador – incluindo a família de processadores x86 –, pode ser difícil realizar a virtualização total com desempenho satisfatório e que o desempenho pode ser melhorado com o fornecimento de uma interface modificada (com o inconveniente de que, então, os sistemas operacionais precisam ser portados para essa interface modificada). Essa técnica é conhecida como *paravirtualização* e será considerada com mais detalhes no estudo de caso a seguir.

Note que a virtualização é muito diferente da estratégia de micronúcleo discutida na Seção 7.6. Embora os micronúcleos aceitem a coexistência de vários sistemas operacionais, isso é conseguido pela simulação do sistema operacional sobre os blocos de construção reutilizáveis oferecidos pelo micronúcleo. Em contraste, na virtualização, um sistema operacional é executado diretamente (ou com pequenas modificações) no *hardware* virtualizado. A principal vantagem da virtualização e o principal motivo de sua predominância em relação aos micronúcleos é que os aplicativos podem ser executados nos ambientes virtualizados sem serem reescritos ou recompilados.

A virtualização começou com a arquitetura do IBM 370, cujo sistema operacional VM pode apresentar várias máquinas virtuais completas para diferentes programas em execução no mesmo computador. Portanto, a técnica remonta aos anos 1970. Mais recentemente, houve um enorme aumento no interesse pela virtualização, com vários projetos de pesquisa e sistemas comerciais fornecendo soluções de virtualização para PCs, servidores e infraestrutura de nuvem. Exemplos de importantes soluções de virtualização incluem o Xen [Barham *et al.* 2003a], o Denali [Whitaker *et al.* 2002], o VMWare, o Parallels e o Microsoft Virtual Server. A seguir, apresentaremos, como estudo de caso, o Xen.

7.7.2 Estudo de caso: a estratégia Xen para virtualização de sistemas

O Xen é um exemplo importante de virtualização, desenvolvido inicialmente como parte do projeto Xenoserver no Laboratório de Computação da Universidade de Cambridge e agora mantido por uma comunidade de código-fonte aberto [www.xen.org]. Uma empresa coligada, a XenSource, foi adquirida pela Citrix Systems, em 2007, e atualmente a Citrix oferece soluções empresariais baseadas na tecnologia Xen, incluindo o XenServer e ferramentas de gerenciamento e automação associadas. A descrição do Xen fornecida a seguir é baseada no artigo de Barham *et al.* [2003a] e nos relatórios internos do XenoServer associados [Barham *et al.* 2003b, Fraser *et al.* 2003], junto a um livro excelente e abrangente sobre o funcionamento interno do Xen Hypervisor [Chisnall 2007].

O objetivo global do projeto XenoServer [Fraser *et al.* 2003] é fornecer uma infraestrutura pública para computação distribuída de longa distância. Assim, esse é um exemplo primitivo de *computação em nuvem*, enfocando a infraestrutura como serviço. Na visão do XenoServer, o mundo é povoado de XenoServers capazes de executar código em nome de clientes, os quais são, então, cobrados pelos recursos que utilizam.

Os dois principais produtos do projeto são o monitor de máquina virtual Xen e a Plataforma Aberta XenoServer, discutidos com mais detalhes a seguir.

O monitor de máquina virtual Xen • O Xen é um monitor de máquina virtual projetado inicialmente para suportar a implementação de XenoServers, mas que evoluiu para uma solução independente para virtualização de sistemas. O objetivo do Xen é permitir que várias instâncias do sistema operacional sejam executadas de forma completamente isolada em *hardware* convencional, com sobrecarga de desempenho mínima. O Xen é projetado para se adaptar a números muito grandes de instâncias do sistema operacional (até várias centenas de máquinas virtuais em um único computador) e tratar da heterogeneidade, procurando suportar a maioria dos sistemas operacionais importantes, incluindo

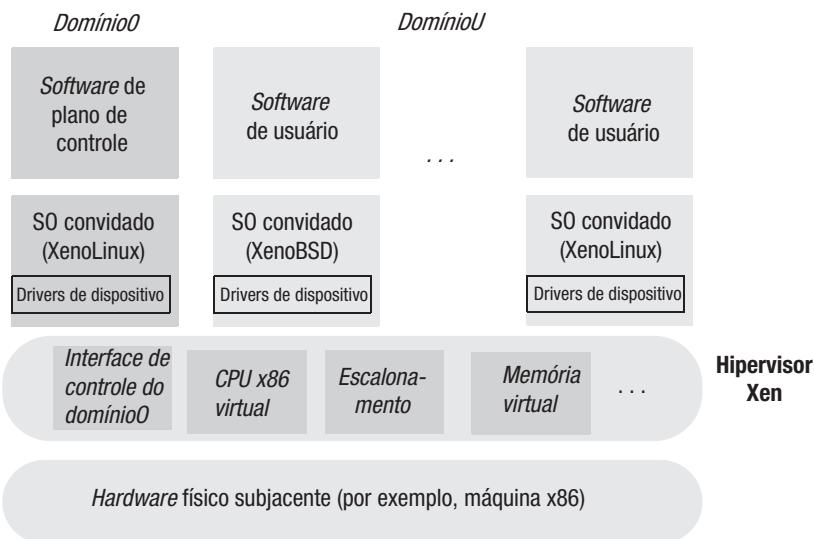


Figura 7.17 A arquitetura do Xen.

Windows, Linux, Solaris e NetBSD. O Xen também funciona em diversas plataformas de *hardware*, incluindo arquiteturas x86 de 32 e 64 bits e CPUs PowerPC e IA64.

A arquitetura do Xen: a arquitetura global do Xen está mostrada na Figura 7.17. O monitor de máquina virtual Xen (conhecido como *hipervisor*) é fundamental nessa arquitetura, suportando a virtualização dos recursos físicos subjacentes, especificamente a CPU e seu conjunto de instruções, o escalonamento da CPU e a gerência da memória física. O objetivo global do hipervisor é fornecer máquinas virtuais com virtualização de *hardware*, determinando a aparência que cada máquina virtual tem de sua própria máquina física (virtualizada) e multiplexando os recursos virtuais nos recursos físicos subjacentes. Para conseguir isso, ele também precisa garantir uma forte proteção entre as diferentes máquinas virtuais que suporta.

O hipervisor segue o projeto do Exonúcleo (apresentado na Seção 7.6), implementando um conjunto mínimo de mecanismos para gerenciamento de recursos e isolamento, deixando a política de nível mais alto para outras partes da arquitetura dos sistemas – em particular, os domínios, conforme discutido a seguir. Além disso, o hipervisor não tem nenhum conhecimento de dispositivos ou de seu gerenciamento; em vez disso, fornece apenas um canal para interagir com os dispositivos (novamente, como discutido a seguir). Esse projeto mínimo é importante por dois motivos principais:

- O Xen ocupa-se principalmente do *isolamento* – incluindo o isolamento de falhas – e, mesmo assim, uma falha no hipervisor pode travar o sistema inteiro. Portanto, é importante que o hipervisor seja mínimo, completamente testado e livre de erros.
- O hipervisor representa uma sobrecarga inevitável, referente à execução no *hardware* básico, e é importante para o desempenho do sistema que isso seja o mais leve possível (como veremos a seguir, a paravirtualização também ajuda a minimizar essa sobrecarga, por ignorar o hipervisor, quando possível).

A função do hipervisor é suportar um número potencialmente grande de instâncias de máquina virtual (denominadas *domínios* no Xen), todas executando sistemas operacionais *convidados*. Os sistemas operacionais convidados são executados em um conjunto de

domínios chamados coletivamente de *domínioU* (ou domínio não-privilegiado, *unprivileged*), referindo-se à sua falta de privilégios em termos de acesso aos recursos físicos (em oposição aos virtuais). Em outras palavras, todo acesso aos recursos é cuidadosamente controlado pelo Xen. O Xen também suporta um domínio especial, chamado *domínio0*, que tem acesso privilegiado aos recursos de *hardware* e atua como plano de controle para a arquitetura Xen, proporcionando uma separação clara entre mecanismo e política no sistema (vamos ver exemplos de utilização do domínio0, a seguir). O domínio0 é configurado para executar uma versão do Linux para o Xen (XenLinux), enquanto outros domínios podem executar qualquer sistema operacional convidado. Note que a arquitetura Xen permite que privilégios selecionados sejam concedidos ao domínioU, especificamente a capacidade de acessar dispositivos de *hardware* diretamente ou de criar novos domínios. Na prática, contudo, a configuração mais comum é o domínio0 manter esses privilégios.

Para continuarmos nosso estudo do Xen, consideremos a implementação das funções básicas do hipervisor – a saber, a virtualização do *hardware* subjacente (incluindo o uso de paravirtualização), o escalonamento e o gerenciamento de memória virtual – antes de mostrarmos como o Xen suporta o gerenciamento de dispositivos. Concluiremos considerando o que é necessário para portar determinado sistema operacional para o Xen.

Virtualização da CPU: a principal função do hipervisor é fornecer, a cada domínio, uma virtualização da CPU; isto é, fornecer a aparência que cada domínio tem de sua própria CPU (virtual) e do conjunto de instruções associado. A complexidade dessa etapa depende inteiramente da arquitetura da CPU física. Nesta seção, abordaremos particularmente a virtualização aplicada na arquitetura x86, a família de processadores mais utilizada atualmente.

Popek e Goldberg [1974], em um artigo clássico sobre requisitos para virtualização, enfocam todas as instruções que podem alterar o estado da máquina de modo a ter impacto sobre outros processos (*instruções sensíveis*), subdividindo essas instruções em:

- *instruções sensíveis de controle*, que tentam mudar a configuração de recursos no sistema; por exemplo, alterando mapeamentos de memória virtual;
- *instruções sensíveis comportamentais*, que leem estado privilegiado e, por meio disso, revelam recursos físicos, em vez de virtuais, violando, com isso, a virtualização.

Então, eles dizem que uma condição para a virtualização é que todas as instruções sensíveis (sensíveis ao controle e ao comportamento) devem ser interceptadas pelo hipervisor (ou mecanismo do núcleo equivalente). Mais especificamente, isso é conseguido pela captura no hipervisor, suportada pelo conceito de *instruções privilegiadas* em uma arquitetura de máquina – isto é, instruções executadas no modo privilegiado ou que gerem uma interrupção de *software* (a qual pode, então, levá-las para o modo privilegiado). Isso leva à seguinte afirmação precisa da condição de Popek e Goldberg:

Condição para virtualização: uma arquitetura de processador serve para virtualização se todas as instruções sensíveis são instruções privilegiadas.

Infelizmente, na família x86 de processadores, esse não é o caso: é possível identificar 17 instruções que são sensíveis, mas não privilegiadas. Por exemplo, as instruções *LAR* (*Load Access Rights* – carregar direitos de acesso) e *LSL* (*Load Segment Limit* – carregar limite de segmento) caem nesta categoria. Elas precisam ser capturadas pelo hipervisor para garantir a virtualização correta, mas não existe um mecanismo para fazer isso, pois elas não são privilegiadas.

Uma solução é fornecer uma camada de simulação para todas as instruções do conjunto de instruções. Assim, é possível gerenciar as instruções sensíveis dentro dessa

camada. Isso é o que é feito na virtualização completa e essa estratégia tem a vantagem de os sistemas operacionais convidados poderem ser executados sem alteração alguma no ambiente virtualizado. Contudo, essa estratégia pode ser dispendiosa, aumentando o custo em cada chamada de instrução afetada. Em contraste, a paravirtualização adota a visão de que muitas instruções podem ser executadas diretamente no *hardware* básico, sem simulação, e que as instruções privilegiadas podem ser capturadas e tratadas pelo hipervisor. Isso leva, então, a instruções sensíveis que não são privilegiadas; uma solução de paravirtualização reconhece que tais instruções podem levar a problemas em potencial, mas deixa que isso seja tratado no sistema operacional convidado. Em outras palavras, o sistema operacional convidado deve ser reescrito para tolerar ou tratar de quaisquer efeitos colaterais dessas instruções. Uma estratégia, por exemplo, é reescrever partes do código para evitar a utilização de instruções problemáticas. Essa estratégia de paravirtualização aumenta muito o desempenho da virtualização, mas à custa de exigir que o sistema operacional convidado seja portado para o ambiente virtualizado.

Para se entender melhor a implementação da paravirtualização, é útil examinar os níveis (ou anéis) de privilégio nos processadores modernos. Por exemplo, a família x86 suporta quatro níveis de privilégio, sendo 0 o mais privilegiado, com o anel 1 sendo o próximo mais privilegiado e assim por diante, até o anel 3, o menos privilegiado. Em um ambiente de sistema operacional tradicional, o núcleo é executado no anel 0 e os aplicativos, no anel 3 – os anéis 1 e 2 não são utilizados. As interrupções de *software* assumem o fluxo de controle do aplicativo para o núcleo e permitem que atividades privilegiadas ocorram. No Xen, o hipervisor é executado no anel 0, e esse é o único anel que pode executar instruções privilegiadas. Então, os sistemas operacionais convidados são executados no anel 1 e os aplicativos são executados no anel 3. As instruções privilegiadas são reescritas como *hiperchamadas* (*hypercalls*) capturadas pelo hipervisor, permitindo que o hipervisor controle a execução das operações potencialmente sensíveis. Todas as outras instruções sensíveis devem ser gerenciadas pelo sistema operacional convidado, conforme discutido anteriormente.

Essa distinção entre sistemas operacionais baseados em núcleo e o Xen está resumida na Figura 7.18.

As hiperchamadas são assíncronas e, assim, representam notificações de que as instruções correspondentes devem ser executadas (não há bloqueio no sistema operacional convidado à espera de um resultado). A comunicação entre o hipervisor e o sistema operacional convidado também é assíncrona e é suportada por um mecanismo de *evento* simples, oferecido pelo hipervisor do Xen. Ele é usado, por exemplo, para tratar de interrupções de dispositivo. O hipervisor mapeia tais interrupções de *hardware* em eventos de *software*, disparados no sistema operacional convidado correto. Portanto, o hipervisor do Xen é completamente *baseado em eventos*.

Escalonamento: vimos, na Seção 7.4, que muitos ambientes de sistema operacional suportam dois níveis de escalonamento – isto é, o escalonamento de processos e o escalonamento de *threads* em nível de usuário dentro de processos. O Xen vai um passo além, introduzindo um nível de escalonamento extra que se preocupa com a execução de sistemas operacionais convidados em particular. Isso é conseguido pela introdução do conceito de CPU virtual (VCPU), com cada VCPU suportando um sistema convidado. Portanto, o escalonamento envolve as seguintes etapas:

- O hipervisor escalona VCPUs nas CPU(s) físicas subjacentes, fornecendo, com isso, uma parte do tempo de processamento físico para cada convidado.

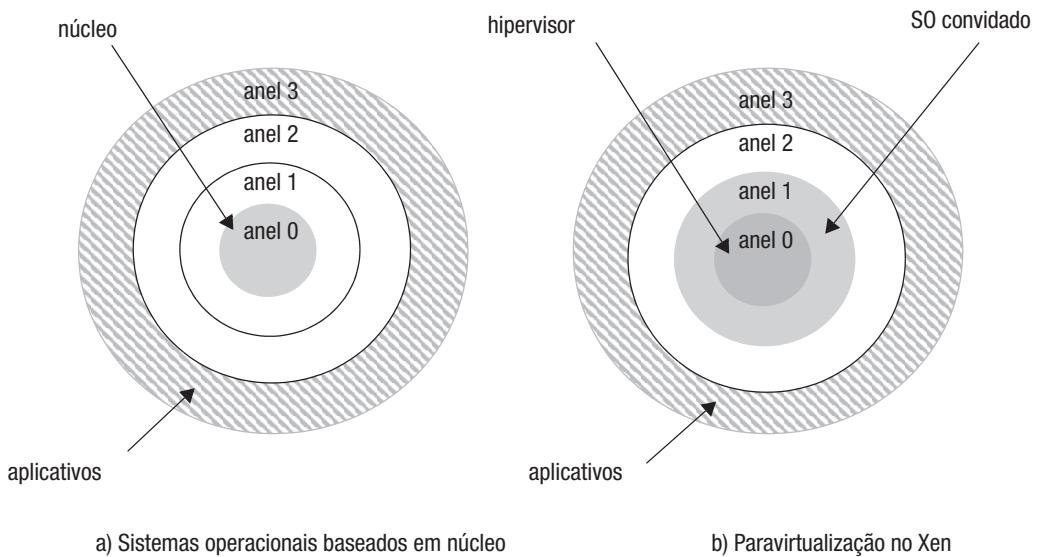


Figura 7.18 Uso de anéis de privilégio.

- Os sistemas operacionais convidados escalonam *threads* em nível de núcleo para suas VCPU(s) alocadas.
- Onde aplicável, bibliotecas de *threads* no espaço do usuário escalonam *threads* em nível de usuário nas *threads* em nível de núcleo disponíveis.

O principal requisito no Xen é que o projeto do escalonador do hipervisor seja previsível, pois escalonadores de nível mais alto farão suposições sobre o comportamento desse escalonador e é fundamental que essas suposições sejam satisfeitas.

O Xen suporta dois escalonadores, *Simple EDF* e *Credit Scheduler*:

Simple Earliest Deadline First (SEDF) Scheduler do Xen: esse escalonador funciona selecionando a VCPU que tem o prazo final mais próximo, com os prazos finais calculados de acordo com dois parâmetros: n (a fatia) e m (o período). Por exemplo, um domínio pode requisitar 10 ms (n) a cada 100 ms (m). O prazo final é definido como o último momento em que esse domínio pode ser executado para satisfazer seu prazo final. Voltando ao nosso exemplo, no ponto de partida do sistema, essa VCPU pode ser escalonada para até 90 ms no período de 100 ms e ainda satisfazer seu prazo final. O escalonador funciona escolhendo o mais antecipado dos prazos finais correntes, examinando o conjunto de VCPUs que podem executar.

Credit Scheduler do Xen: para este escalonador, cada domínio (VCPU) é especificado em termos de duas propriedades: o *peso* (*weight*) e o *limite* (*cap*). O peso determina a cota da CPU que deve ser dada a essa VCPU. Por exemplo, se uma VCPU tem peso 64 e outra tem peso 32, a primeira VCPU deve receber o dobro da cota da segunda. Os pesos válidos vão de 1 a 65.535, com o padrão sendo 256. Esse comportamento é modificado pelo *limite*, que expressa a porcentagem total da CPU que deve ser dada à VCPU correspondente. Esse limite pode não existir. O escalonador transforma o peso associado a uma VCPU em créditos e, quando essa VCPU é executada, consome os créditos. A VCPU é considerada *under* (abaixo)

se tem créditos restando; caso contrário, é considerada *over (acima)*. Para cada CPU, o escalonador mantém uma fila de VCPUs executáveis, com todas as VCPUs *under* primeiro, seguidas das VCPUs *over*. Quando uma VCPU não é escalonada, é colocada nessa fila, no final da categoria apropriada (dependendo de estar *under* ou *over* em relação ao crédito no momento). Então, o escalonador escolhe o primeiro elemento da fila para executar em seguida. Como uma forma de balanceamento de carga, se determinada CPU não tiver VCPUs *under*, ela pesquisará as filas de outras CPUs em busca de uma possível candidata ao escalonamento nessa CPU.

Esses escalonadores substituem os anteriores do Xen, inclusive um escalonador *round robin* simples, um baseado em *tempo virtual emprestado* (projeto para fornecer uma cota proporcional da CPU física, de acordo com a configuração de diferentes pesos de domínio) e o *Atropos* (projeto para suportar escalonamento flexível em tempo real). Mais detalhes sobre esses escalonadores podem ser encontrados em Chisnall [2007].

Também é possível adicionar novos escalonadores no hipervisor do Xen, mas isso que deve ser feito com cuidado e após testes extensivos, em razão dos requisitos do hipervisor, conforme discutido anteriormente. Chisnall [2007] fornece um guia passo a passo sobre como implementar um escalonador simples no Xen.

A interação entre sistemas operacionais convidados e o escalonador subjacente é feita por meio de diversas hiperchamadas específicas do escalonador, incluindo operações para *liberação* voluntária da CPU (mas permanecendo executável), para *blockear* um domínio em particular até que um evento tenha ocorrido ou para *encerrar* o domínio por um motivo especificado.

Gerenciamento de memória virtual: o gerenciamento de memória virtual é o aspecto mais complicado da virtualização, parcialmente devido à complexidade das soluções de *hardware* subjacentes para gerenciamento de memória e parcialmente devido à necessidade de injetar níveis de proteção extras para fornecer isolamento entre os diferentes domínios. Fornecemos, a seguir, alguns princípios gerais de gerenciamento de memória no Xen. O leitor está convidado a estudar a descrição detalhada do gerenciamento de memória virtual no Xen, fornecida por Chisnall [2007].

A estratégia global de virtualização de gerenciamento de memória no Xen aparece na Figura 7.19. Assim como no escalonamento, o Xen adota uma arquitetura de três níveis, com o hipervisor gerenciando a memória física, o núcleo do sistema operacional convidado fornecendo memória pseudofísica e os aplicativos dentro desse sistema operacional usando memória virtual, como seria de se esperar em qualquer sistema operacional subjacente. O conceito de *memória pseudofísica* é fundamental para se entender essa arquitetura e está descrito com mais detalhes a seguir.

A principal decisão de projeto na arquitetura de gerenciamento de memória virtual é manter a funcionalidade do hipervisor em um nível mínimo. Efetivamente, o hipervisor tem apenas duas funções – a alocação e o subsequente gerenciamento da memória física, na forma de páginas:

- Em termos de *alocação de memória*, o hipervisor mantém uma pequena parte da memória física para suas próprias necessidades e, então, aloca páginas para domínios de acordo com a demanda. Por exemplo, quando um novo domínio é criado, ele recebe um conjunto de páginas, de acordo com suas necessidades declaradas. Na prática, esse conjunto de páginas será fragmentado pelo espaço de endereçamento físico e isso pode estar em conflito com as expectativas do sistema operacional convidado (o qual pode esperar um espaço de endereçamento contíguo). A função da memória pseudofísica é

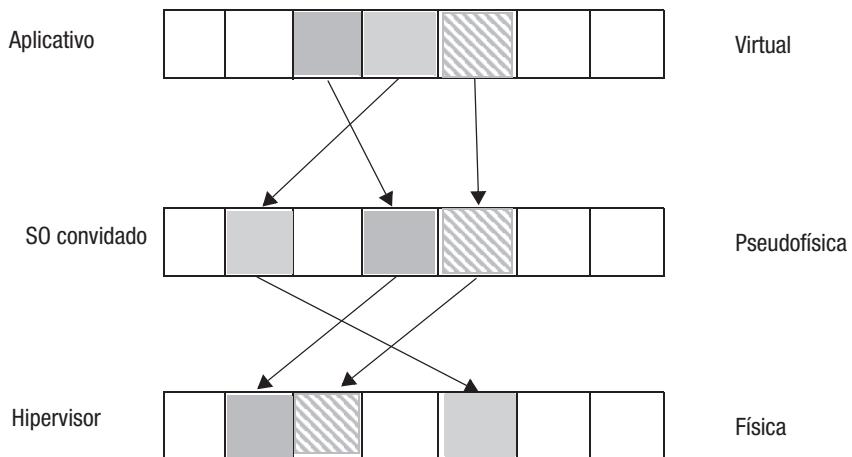


Figura 7.19 Virtualização de gerenciamento de memória.

fornecer essa abstração, oferecendo um espaço de endereçamento pseudofísico contíguo e, então, mantendo um mapeamento desse espaço de endereçamento para os endereços físicos reais. Fundamentalmente, esse mapeamento deve ser gerenciado pelo sistema operacional convidado e não pelo hipervisor, a fim de manter a natureza leve do hipervisor (mais especificamente, a composição das duas funções mostradas na Figura 7.19 é realizada no convidado). Essa estratégia permite ao sistema operacional convidado interpretar o mapeamento em seu próprio contexto (por exemplo, para alguns sistemas operacionais convidados, onde não são esperados endereços contíguos, esse mapeamento pode ser eliminado) e também torna mais fácil migrar um domínio para um espaço de endereçamento diferente, por exemplo, em outra máquina. O mesmo mecanismo também é usado para suportar suspensão e reinício de sistemas operacionais convidados. Na suspensão, o estado do domínio é serializado no disco e, no reinício, esse estado é restaurado, mas em um local físico diferente. Isso é suportado pelo nível de indireção extra na arquitetura de gerenciamento de memória.

- Em termos de gerenciamento da memória física, o hipervisor exporta um pequeno conjunto de hiperchamadas para manipular as tabelas de páginas subjacentes. Como ilustração, a hiperchamada `pt_update(lista de requisições)` é usada por um sistema operacional convidado para requisitar um lote de atualizações incrementais para uma tabela de páginas. Isso permite ao hipervisor validar todas as requisições e executar apenas as atualizações consideradas seguras, por exemplo, para impor o isolamento.

O resultado global é uma estratégia de gerenciamento de memória virtual flexível, que permite aos sistemas operacionais convidados otimizar suas implementações para diferentes famílias de processador.

Gerenciamento de dispositivos: a estratégia do Xen para gerenciamento de dispositivos conta com o conceito de *drivers de dispositivo divididos*, como mostrado na Figura 7.20. Como pode-se ver nessa figura, o acesso ao dispositivo físico é controlado exclusivamente pelo domínio0, que também contém um *driver real* para esse dispositivo. Como o domínio0 executa XenoLinux, esse vai ser um *driver Linux* de dispositivo disponível. É importante enfatizar que, embora alguns *drivers de dispositivo* tenham bom suporte para multiplexação, outros não o têm; assim, é importante

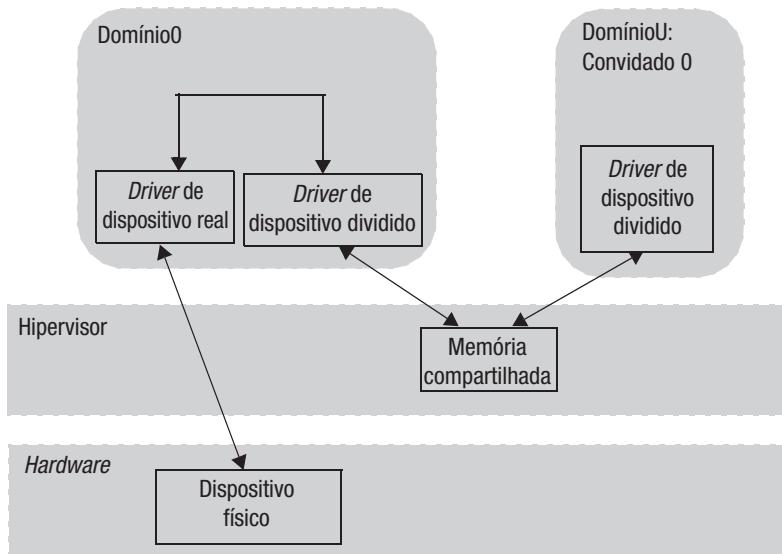


Figura 7.20 Drivers de dispositivo divididos.

que o Xen forneça uma abstração por meio da qual cada sistema operacional convidado possa ter a aparência de ter seu próprio *dispositivo virtual*. Isso é obtido com a estrutura de *driver dividido*, envolvendo um *driver de dispositivo de back-end* em execução no domínio0 e um *driver de front-end* em execução no sistema operacional convidado. Então, os dois se comunicam para fornecer o acesso a dispositivo necessário para o sistema operacional convidado. As respectivas funções das partes *back-end* e *front-end* do *driver* são as seguintes:

- O *back-end* tem duas funções importantes a executar na arquitetura. Primeiro, ele precisa gerenciar a multiplexação (em particular, o acesso de vários sistemas operacionais convidados), especialmente onde não é fornecido suporte no *driver* Linux subjacente. Segundo, ele fornece uma interface genérica que captura as funções essenciais do dispositivo e é neutra para os diferentes sistemas operacionais convidados que vão utilizá-la. Isso se torna mais fácil porque os sistemas operacionais já oferecem várias abstrações que efetivamente fornecem a multiplexação necessária de maneira neutra, por exemplo, lendo e escrevendo blocos nos dispositivos de armazenamento persistente. Interfaces de nível mais alto (por exemplo, soquetes) seriam inadequadas, pois seriam influenciadas demais por determinadas abstrações do sistema operacional.
- Em contraste, o *front-end* é muito simples e atua como um *proxy* para o dispositivo no ambiente do sistema operacional convidado, aceitando comandos e se comunicando com o *back-end*, como descrito a seguir.

A comunicação entre o *front-end* e o *back-end* na estrutura de dispositivo dividido é facilitada pela criação de uma página de memória compartilhada entre os dois componentes. A região de memória compartilhada é estabelecida por meio de um mecanismo de *tabela de concessões*, suportada pelo hipervisor. Uma tabela de concessões é um vetor de estruturas (entradas de concessão) que suporta operações para conceder permissões, a fim de

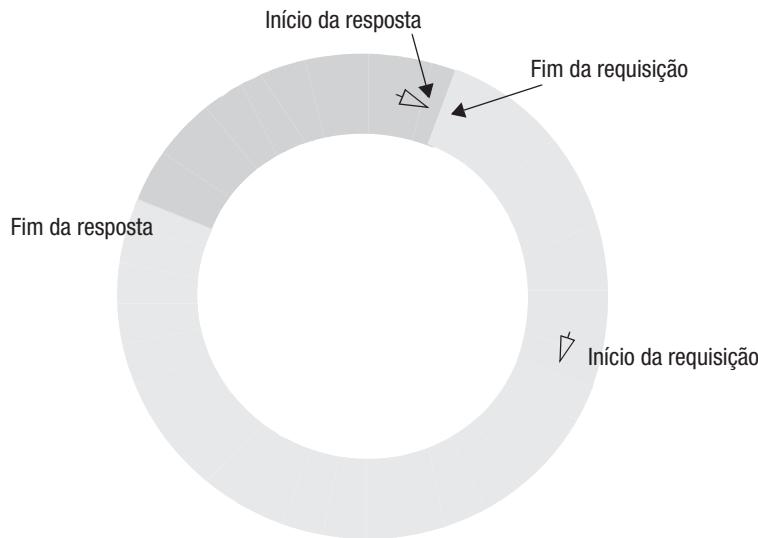


Figura 7.21 Anéis de E/S.

garantir o acesso externo a uma reserva de memória ou para acessar outras reservas de memória por meio de referências de concessão. Pode ser concedido acesso a leitura ou escrita na região de memória compartilhada. Esse mecanismo oferece uma maneira leve e de alto desempenho para diferentes domínios se comunicarem no Xen.

O mecanismo normal para se comunicar é usar uma estrutura de dados (conhecida como *anel de E/S*) nessa região de memória compartilhada, a qual suporta comunicação assíncrona bilateral entre as duas partes do *driver* de dispositivo dividido. A estrutura de um anel de E/S aparece na Figura 7.21. Os domínios se comunicam por meio de requisições e respostas. Em particular, um domínio escreve sua requisição no sentido horário, começando no indicador de início de requisição (supondo que haja espaço suficiente) e movendo o ponteiro correspondente. Então, a outra extremidade pode ler os dados a partir do fim, novamente movendo o ponteiro associado. O mesmo procedimento ocorre para as respostas. Para dispositivos que transferem grandes volumes de dados continuamente, os pontos extremos correspondentes executarão consultas a essa estrutura de dados (*polling*). Para transferências menos frequentes, os anéis de E/S podem ser complementados pelo uso do mecanismo de eventos do Xen, para notificar o destinatário de que dados estão prontos para consumo. O mecanismo de descoberta de dispositivo é realizado por meio de um espaço de informações compartilhado, chamado *XenStore*, acessível a todos os domínios. O *XenStore* em si é implementado como um dispositivo, usando a arquitetura de dispositivo dividido, a qual os *drivers* de dispositivo utilizam para anunciar seus serviços. As informações fornecidas incluem a referência de concessão dos anéis de E/S associados ao dispositivo e também (onde apropriado) quaisquer canais de evento associados ao dispositivo. Os diversos recursos de comunicação usados pelos *drivers* de dispositivo (anéis de E/S, eventos e *XenStore*) são coletivamente referidos como *XenBus*.

Uma instalação de Xen pode fornecer diferentes configurações de *drivers* de dispositivo. Contudo, espera-se que a maioria das implementações de Xen forneça dois *drivers* genéricos:

- O primeiro é o *driver de dispositivo de bloco*, oferecendo uma abstração comum para dispositivos de bloco (mais comumente, dispositivos de armazenamento). A interface para isso é muito simples, suportando três operações: *ler* ou *escrever* um bloco e implementar uma *barreira de escrita*, garantindo que todas as escritas pendentes sejam concluídas.
- O segundo é o *Virtual Interface Network Driver* do Xen, que oferece uma interface comum para interagir com dispositivos de rede. Ele usa dois anéis de E/S para transmitir e receber dados para/da rede. Rigorosamente falando, os anéis são usados para fluxo de controle, e áreas de memória compartilhadas separadas são usadas para os dados associados (o que ajuda em termos de minimizar as cópias e reutilizar regiões da memória).

Note que a maior parte dessa arquitetura é implementada acima do hipervisor, no domínio0 e nos outros sistemas operacionais convidados. A função do hipervisor é simplesmente facilitar a comunicação entre domínios, por exemplo por meio do mecanismo de tabela de concessões, e o restante é construído sobre essa base mínima. Isso ajuda consideravelmente, em termos de manter o hipervisor pequeno e eficiente.

Porte de um sistema operacional convidado: a partir das descrições anteriores, agora é possível ver o que é necessário para portar um sistema operacional para o ambiente Xen. Isso envolve vários estágios importantes:

- substituir todas as instruções privilegiadas utilizadas pelo sistema operacional por hiperchamadas correspondentes;
- pegar todas as instruções sensíveis e reimplementá-las de uma maneira que preserve a semântica das operações associadas desejadas;
- portar o subsistema de memória virtual;
- desenvolver *drivers* de dispositivo divididos para o conjunto de dispositivos exigido, reutilizando a funcionalidade de *driver* de dispositivo já fornecida no domínio0, junto às interfaces de *driver* de dispositivo genéricas, onde for apropriado.

Essa lista abrange as principais tarefas, mas existem outras, mais específicas, que precisam ser executadas. Por exemplo, o Xen oferece sua própria arquitetura de tempo, reconhecendo a diferença entre o tempo real e a passagem de tempo observada pelos sistemas operacionais convidados individuais.

Em mais detalhes, o hipervisor fornece suporte para várias abstrações de tempo – especificamente, um *tempo de contador de ciclos* subjacente, baseado no relógio do processador físico e usado para extrapolar outras referências de tempo; *tempo de domínio virtual*, que avança na mesma velocidade do tempo de contador de ciclos, mas somente quando um domínio em particular é escalonado; *tempo de sistema*, que reflete precisamente a passagem de tempo real no sistema; e *tempo do relógio de parede*, que fornece a hora real do dia. Supõe-se que as instâncias de sistema operacional em execução nos domínios forneçam abstrações do tempo real e do tempo virtual sobre esses valores, exigindo mais trabalho ao portar. Curiosamente, tanto o tempo de sistema como o tempo do relógio de parede são corrigidos automaticamente com as variações de relógio, explorando uma única instância de um cliente NTP (descrito no Capítulo 14) em execução no domínio0. Esse é apenas um exemplo das otimizações permitidas pelo domínio0 compartilhado.

A Plataforma Aberta XenoServer • Conforme mencionado anteriormente, o Xen foi desenvolvido inicialmente como parte do projeto XenoServer, que investigava infraestrutura de *software* para computação distribuída de longa distância. Vamos descrever, agora, a

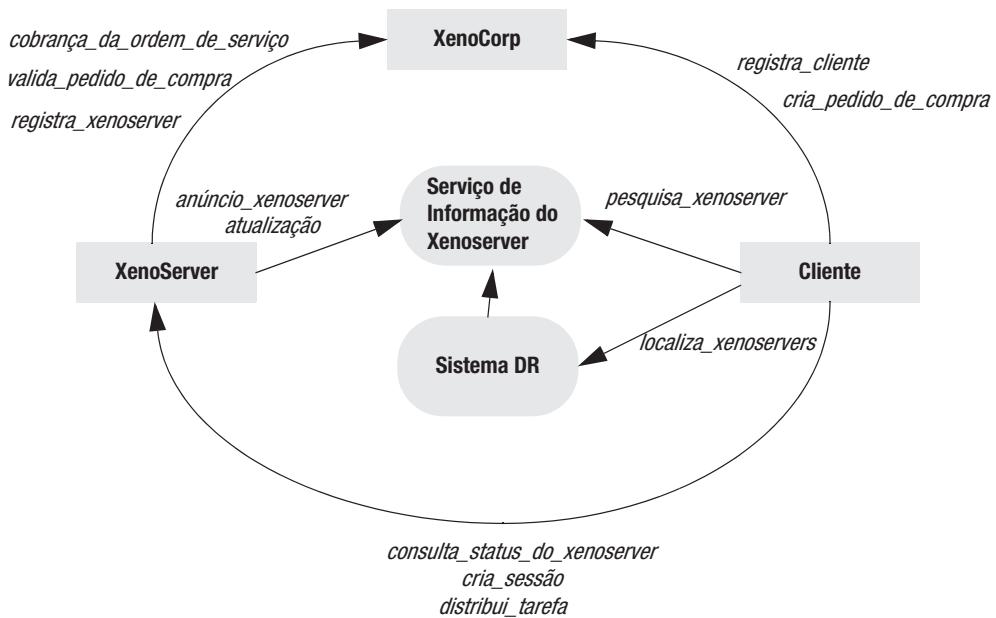


Figura 7.22 A Arquitetura da Plataforma Aberta XenoServer.

arquitetura global da Plataforma Aberta XenoServer associada [Hand *et al.* 2003]. Nessa arquitetura, que aparece na Figura 7.22, para utilizar o sistema, os clientes são registrados em um entidade conhecida como XenoCorp. Os desenvolvedores da Arquitetura Aberta XenoServer preveem diversas instâncias concorrentes de XenoCorps existindo em determinado sistema, oferecendo diferentes regimes de pagamento e diferentes níveis de qualidade de serviço (por exemplo, variado suporte para privacidade). Mais formalmente, a função de determinado XenoCorp é oferecer serviços de autenticação, auditoria, cobrança e pagamento e manter uma relação contratual com os clientes e com as organizações que oferecem XenoServers. Isso é suportado por um processo de registro por meio do qual a identidade é estabelecida e pela criação de ordens de compra, as quais representam o compromisso do cliente (autenticado) de bancar determinada sessão.

Na arquitetura global, os XenoServers competem entre si para oferecer serviços. Então, a função do Serviço de Informação do XenoServer é permitir que os XenoServers anunciem seus serviços e que os clientes localizem XenoServers apropriados com base em seus requisitos especificados. Os anúncios são especificados usando-se XML e incluem cláusulas que abrangem funcionalidade, disponibilidade de recursos e preço.

O Serviço de Informação é relativamente primitivo, oferecendo mecanismos de pesquisa básicos no conjunto de anúncios. Para complementar isso, a arquitetura da plataforma também oferece um sistema de descoberta de recursos (DR) que suporta consultas mais complexas, como:

- Localizar um XenoServer com enlace de baixa latência para o cliente e que satisfaça certos requisitos de recurso para determinado preço.
- Localizar um agrupamento (*cluster*) de XenoServers que estejam interconectados por enlaces de baixa latência, suportem comunicação segura e satisfaçam certos requisitos de recurso.

A principal inovação no projeto XenoServer está em como ele acopla a arquitetura anterior com a virtualização – cada XenoServer executa o monitor de máquina virtual do Xen, permitindo que os clientes ofereçam recursos virtuais (em vez de físicos) e que o sistema gerencie o conjunto de recursos de forma mais eficaz, graças a essa virtualização. Essa é uma ilustração clara da natureza complementar da computação em nuvem e da virtualização, conforme discutido anteriormente.

7.8 Resumo

Este capítulo descreveu como o sistema operacional suporta a camada de *middleware* no fornecimento de invocações a recursos compartilhados. O sistema operacional (SO) fornece um conjunto de mecanismos nos quais diversas políticas de gerenciamento de recursos podem ser implementadas para atender aos requisitos locais e para tirar proveito dos aprimoramentos tecnológicos. O SO permite que os servidores encapsulem e protejam os recursos, enquanto permite que os clientes os compartilhem de forma concorrente. Ele fornece os mecanismos necessários para os clientes invocarem operações em recursos.

Um processo consiste em um ambiente de execução e *threads*: um ambiente de execução é formado por um espaço de endereçamento, interfaces de comunicação e outros recursos locais, como os semáforos; uma *thread* é uma abstração de atividade executada dentro de um ambiente de execução. Os espaços de endereçamento precisam ser grandes e esparsos para suportarem compartilhamento e acesso mapeado a objetos como, por exemplo, arquivos. Novos espaços de endereçamento podem ser criados a partir de regiões herdadas de processos pais. Uma técnica importante para copiar regiões é a cópia em escrita.

Os processos podem ter múltiplas *threads*, as quais compartilham o ambiente de execução. Os processos *multithreadeds* nos permitem obter concorrência com um custo relativamente baixo e explorar o paralelismo real disponível nos multiprocessadores. Eles são úteis tanto para clientes como para servidores. As implementações recentes de *threads* permitem o escalonamento em duas camadas: o núcleo dá acesso a vários processadores, enquanto o código em nível de usuário trata dos detalhes das políticas de escalonamento.

O sistema operacional fornece primitivas básicas para passagem de mensagem e mecanismos para comunicação por meio de memória compartilhada. A maioria dos núcleos inclui comunicação de rede como um recurso básico; outros fornecem apenas comunicação local e deixam a comunicação de rede para os servidores, os quais podem implementar uma variedade de protocolos de comunicação. Esse é um compromisso de desempenho em relação à flexibilidade.

Discutimos as invocações remotas e levamos em conta a diferença entre sobrecargas decorrentes diretamente da rede e as ocasionadas pela execução do código do sistema operacional. Descobrimos que a proporção do tempo total relacionado ao *software* é relativamente grande para uma invocação nula, mas diminui proporcionalmente com o tamanho dos argumentos da requisição. As principais sobrecargas envolvidas em uma invocação, que são candidatas à otimização, são o empacotamento, a cópia de dados, a inicialização de pacotes, o escalonamento e o chaveamento de contexto de *threads* e o protocolo de controle de fluxo utilizado. A invocação entre espaços de endereçamento em um mesmo computador é um caso especial importante, e descrevemos as técnicas de gerenciamento de *threads* e passagem de parâmetros usadas na RPC leve.

Existem duas estratégias principais de arquitetura do núcleo: núcleos monolíticos e micronúcleos. A principal diferença entre elas reside em onde é traçada a linha entre o

gerenciamento de recursos pelo núcleo e o gerenciamento de recursos realizado por servidores carregados dinamicamente (e, normalmente, em nível de usuário). Um micronúcleo precisa suportar pelo menos a noção de processo e de comunicação entre processos. Ele suporta subsistemas de simulação do sistema operacional, assim como subsistemas de suporte à linguagem e a outros serviços, como os para processamento em tempo real. A virtualização oferece uma atraente alternativa a esse estilo, fornecendo simulação de *hardware* e, então, permitindo que várias máquinas virtuais (e, portanto, vários sistemas operacionais) coexistam no mesmo computador.

Exercícios

- 7.1 Discuta as tarefas de encapsulamento, processamento concorrente, proteção, transformação de nomes, comunicação de parâmetros e resultados e escalonamento, no caso do serviço de arquivos do UNIX (ou de outro núcleo com que você esteja familiarizado). *página 282*
- 7.2 Por que algumas interfaces de sistema são implementadas por chamadas de sistema dedicadas (ao núcleo) e outras por chamadas de sistema baseadas em mensagens? *página 282*
- 7.3 Smith decide que qualquer *thread* em seus processos deve ter sua própria pilha protegida – todas as outras regiões em um processo seriam totalmente compartilhadas. Isso faz sentido? *página 286*
- 7.4 Os tratadores de um sinal (interrupção de *software*) devem pertencer a um processo ou a uma *thread*? *página 286*
- 7.5 Discuta o problema da atribuição de nomes aplicada às regiões de memória compartilhada. *página 288*
- 7.6 Sugira um esquema para balancear a carga em um conjunto de computadores. Você deve discutir:
 - i) quais requisitos de usuário ou sistema são atendidos por tal esquema;
 - ii) para quais categorias de aplicativos ele é conveniente;
 - iii) como medir a carga e com que precisão;
 - iv) como monitorar a carga e escolher a localização de um novo processo. Suponha que os processos não podem migrar.Como seu projeto seria afetado se os processos pudessem migrar entre computadores? Você esperaria que a migração de processo tivesse um custo significativo? *página 289*
- 7.7 Explique a vantagem da cópia de região com *cópia na escrita* para o UNIX, em que uma chamada *fork* normalmente é acompanhada de uma chamada *exec*. O que deve acontecer se uma região que foi copiada usando *cópia na escrita* for, ela própria, copiada? *página 291*
- 7.8 Um servidor de arquivos usa cache e obtém uma taxa de acertos de 80%. As operações no servidor custam 5 ms de tempo da CPU, quando o servidor encontra o bloco solicitado na cache, e levam mais 15 ms de tempo de E/S de disco, quando ele não encontra. Explicando todas as suposições que fizer, faça uma estimativa da capacidade de *throughput* (pedidos médios/seg), se ele:
 - i) *for single-threaded*;
 - ii) usar duas *threads* executando em um único processador;
 - iii) usar duas *threads* executando em um computador com dois processadores. *página 292*
- 7.9 Compare a arquitetura *multi-threaded* de conjunto de trabalhadores, com a arquitetura de uma *thread* por pedido. *página 293*

- 7.10 Quais operações de *thread* têm custo mais significativo? **página 295**
- 7.11 Um *spin lock* (veja Bacon [2002]) é uma variável booleana acessada por meio de uma instrução *test-and-set* atômica, que é usada para obter exclusão mútua. Você usaria *spin lock* para obter exclusão mútua entre *threads* em um computador com um só processador? **página 298**
- 7.12 Explique o que o núcleo deve fornecer para uma implementação de *threads* em nível de usuário, como o Java no UNIX. **página 300**
- 7.13 A falta de página representa um problema para implementações de *threads* em nível de usuário? **página 300**
- 7.14 Explique os fatores que motivam a estratégia de escalonamento mista do projeto de “ativações do escalonador” (em vez de escalonamento em nível de usuário ou em nível de núcleo pura). **página 301**
- 7.15 Por que um pacote de *threads* deve estar interessado nos eventos que bloqueiam ou desbloqueiam uma *thread*? Por que ele deve estar interessado no evento de preempção de um processador virtual? (Dica: outros processadores virtuais podem continuar a ser alocados.) **página 302**
- 7.16 O tempo de transmissão na rede é responsável por 20% de uma RPC nula e por 80% de uma RPC que transmite 1.024 bytes de usuário (menor do que o tamanho de um pacote de rede). Qual será a porcentagem dos tempos dessas duas operações se a rede for migrada de 10 megarbytes/segundo para 100 megabits/segundo? **página 305**
- 7.17 Uma RMI “nula”, que não recebe parâmetros, chama um procedimento vazio e não retorna valores, “travando” o chamador por 2,0 milissegundos. Explique o que contribui para esse tempo. No mesmo sistema RMI, cada 1K de dados do usuário adiciona mais 1,5 milissegundos. Um cliente deseja buscar 32K de dados de um servidor de arquivos. Ele deve usar uma RMI de 32K ou 32 RMIs de 1K? **página 305**
- 7.18 Quais fatores identificados no custo de uma invocação remota também aparecem na passagem de mensagens? **página 307**
- 7.19 Explique como uma região compartilhada poderia ser usada por um processo para ler dados gravados pelo núcleo. Inclua em sua explicação o que seria necessário para a sincronização. **página 308**
- 7.20 i) Um servidor ativado por chamadas de procedimento leves pode controlar o grau de concorrência dentro dele?
ii) Explique por que e como um cliente é impedido de chamar código arbitrário dentro de um servidor na RPC leve.
iii) A LRPC expõe os clientes e servidores a maiores riscos de interferência mútua do que a RPC convencional (dado o compartilhamento da memória)? **página 309**
- 7.21 Um cliente faz RMIs em um servidor. O cliente leva 5 ms para computar os argumentos de cada requisição, e o servidor leva 10 ms para processar cada requisição. O tempo de processamento do SO local para cada operação *send* ou *receive* é de 0,5 ms e o tempo da rede para transmitir cada requisição ou resposta é de 3 ms. O empacotamento ou desempacotamento leva 0,5 ms por mensagem.
- Faça uma estimativa do tempo que leva para o cliente gerar e retornar 2 pedidos (i) se for *single-threaded* e (ii) se tiver duas *threads* que podem fazer pedidos concorrentemente em um único processador. Há necessidade de RMI assíncrona se os processos forem *multithreaded*? **página 311**

- 7.22 Explique o que é política de segurança e quais são os mecanismos correspondentes no caso de um sistema operacional multiusuário como o UNIX. *página 314*
- 7.23 Explique os requisitos de ligação de programa (*linking*) que devem ser satisfeitos se um servidor precisa ser carregado dinamicamente no espaço de endereçamento do núcleo e como eles diferem do caso da execução de um servidor em nível de usuário. *página 315*
- 7.24 Como uma interrupção poderia ser comunicada a um servidor em nível de usuário? *página 317*
- 7.25 Em certo computador, estimamos que, independentemente do SO que execute, o escalonamento de *threads* custa cerca de 50 µs, uma chamada de procedimento nula custa 1 ms, uma troca de contexto para o núcleo custa 20 µs e uma transição de domínio custa 40 µs. Para o Mach e para o SPIN, faça uma estimativa do custo para um cliente chamar um procedimento nulo carregado dinamicamente. *página 317*
- 7.26 Qual é a diferença entre a estratégia de virtualização defendida pelo Xen e o estilo de micronúcleo defendido pelo projeto do Exonúcleo? Em sua resposta, destaque duas coisas que elas têm em comum e duas características diferenciadas entre as estratégias. *páginas 317, 320*
- 7.27 Esboce, em pseudocódigo, como você adicionaria um escalonador *round robin* simples no hipervisor do Xen, usando a estrutura discutida na Seção 7.7.2. *página 323*
- 7.28 A partir de seu entendimento da estratégia de virtualização do Xen, discuta características específicas do Xen que podem suportar a arquitetura do XenoServer, ilustrando, assim, a sinergia entre a virtualização e a computação em nuvem. *páginas 320, 330*

8

Objetos e Componentes Distribuídos

- 8.1 Introdução
- 8.2 Objetos distribuídos
- 8.3 Estudo de caso: CORBA
- 8.4 De objetos a componentes
- 8.5 Estudos de caso: Enterprise JavaBeans e Fractal
- 8.6 Resumo

Uma solução de *middleware* completa deve apresentar uma abstração de programação de nível mais alto e abstrair as complexidades subjacentes envolvidas nos sistemas distribuídos. Este capítulo examina duas das abstrações de programação mais importantes – objetos distribuídos e componentes – e examina as plataformas de *middleware* associadas, incluindo CORBA, Enterprise JavaBeans e Fractal.

O CORBA é um *middleware* que permite às aplicações se comunicaremumas com as outras independentemente de suas linguagens de programação, de suas plataformas de *hardware* e *software*, das redes pelas quais se comunicam e de seus desenvolvedores. As aplicações são construídas a partir de objetos CORBA, os quais implementam interfaces determinadas pela linguagem de definição de interface, IDL (Interface Description Language), do CORBA. Assim como RMI Java, o CORBA suporta invocação transparente de métodos em objetos remotos. O componente de *middleware* que suporta RMI é chamado de Object Request Broker ou ORB.

O *middleware* baseado em componentes surgiu como uma evolução natural dos objetos distribuídos, fornecendo suporte para gerenciamento de dependências entre componentes, ocultando os detalhes de baixo nível associados ao *middleware*, gerenciando as complexidades das aplicações distribuídas com propriedades não funcionais apropriadas (como a segurança) e suportando estratégias de distribuição adequadas. As principais tecnologias nessa área incluem o Enterprise JavaBeans e o Fractal.

8.1 Introdução

Os capítulos anteriores apresentaram os blocos de construção subjacentes fundamentais dos sistemas distribuídos, em termos de comunicação e de sistema operacional. Este capítulo se concentra nas soluções de *middleware* completas, apresentando os objetos distribuídos e componentes como dois dos estilos de *middleware* mais importantes em uso atualmente. Essa discussão é seguida, nos Capítulos 9 e 10, pela consideração de estratégias alternativas, baseadas em serviços Web e soluções *peer-to-peer*.

Conforme discutido no Capítulo 2, a tarefa do *middleware* é fornecer uma abstração de programação de nível mais alto para o desenvolvimento de sistemas distribuídos e, por meio de camadas, abstrair a heterogeneidade da infraestrutura subjacente a fim de promover a operação conjunta e a portabilidade.

Middleware de objetos distribuídos • A principal característica dos objetos distribuídos é que eles permitem adotar um modelo de programação orientada a objetos para o desenvolvimento de sistemas distribuídos e, por meio disso, ocultar a complexidade da programação distribuída. Nessa estratégia, as entidades que se comunicam são representadas por objetos. Os objetos se comunicam usando, principalmente, invocação de método remoto, mas possivelmente usando também um paradigma de comunicação alternativo (como os eventos distribuídos). Essa estratégia relativamente simples tem diversas vantagens importantes, incluindo:

- O *encapsulamento* inerente às soluções baseadas em objetos é apropriado à programação distribuída.
- A propriedade relacionada da *abstração de dados* oferece uma clara separação entre a especificação de um objeto e sua implementação, permitindo aos programadores lidar exclusivamente com as interfaces e não se preocupar com os detalhes da implementação, como a linguagem de programação e o sistema operacional usados.
- Essa estratégia também proporciona soluções mais *dinâmicas* e *extensíveis*, permitindo, por exemplo, a introdução de novos objetos ou a substituição de um objeto por outro (compatível).

Estão disponíveis diversas soluções de *middleware* baseadas em objetos distribuídos, incluindo RMI Java e CORBA. Resumimos as principais características dos objetos distribuídos na Seção 8.2 e fornecemos um estudo de caso detalhado do CORBA na Seção 8.3.

Middleware baseado em componentes • As soluções baseadas em componentes foram desenvolvidas para superar diversas limitações observadas pelos desenvolvedores de aplicativo que trabalham com *middleware* de objeto distribuído:

Dependências implícitas: as interfaces de objeto não descrevem as dependências da implementação de um objeto, tornando difícil desenvolver sistemas baseados em objetos (especialmente por parte de desenvolvedores externos) e, subsequentemente, gerenciá-los.

Complexidade de programação: a programação de *middleware* de objeto distribuído leva à necessidade de dominar muitos detalhes de baixo nível associados às implementações de *middleware*.

Falta de separação de aspectos relacionados à distribuição: os desenvolvedores de aplicativo são obrigados a considerar detalhes relacionados a segurança, falha de tratamento e concorrência, que são predominantemente semelhantes de um aplicativo para outro.

Ausência de suporte para implantação: o *middleware* baseado em objeto fornece pouco ou nenhum suporte para a implantação e definição de configurações (potencialmente complexas) de objetos.

As soluções baseadas em componentes podem ser mais bem entendidas como uma evolução natural dos objetos, complementando a forte herança desse trabalho anterior. A Seção 8.4 discutirá essa argumentação com mais detalhes e apresentará as principais características de uma estratégia baseada em componentes. Em seguida, a Seção 8.5 apresentará dois estudos de caso contrastantes de soluções baseadas em componentes, Enterprise JavaBeans e Fractal, com o primeiro oferecendo uma solução abrangente que abstrai muitos dos principais problemas do desenvolvimento de aplicações distribuídas e o segundo representando uma solução mais leve, frequentemente usada na construção de tecnologias de *middleware* mais complexas.

8.2 Objetos distribuídos

O *middleware* baseado em objetos distribuídos é projetado para fornecer um modelo de programação a partir de princípios orientados a objetos e, portanto, para trazer as vantagens da estratégia orientada a objetos para a programação distribuída.

Emmerich [2000] vê esses objetos distribuídos como uma evolução natural de três áreas de atividade:

- *Sistemas distribuídos:* os primeiros *middlewares* eram baseados no modelo cliente-servidor e havia o desejo por abstrações de programação mais sofisticadas.
- *Linguagens de programação:* o trabalho anterior em linguagens orientadas a objetos, como Simula-67 e Smalltalk, levou ao surgimento de outras, mais atuais e muito utilizadas, como Java e C++ (linguagens extensivamente usadas em sistemas distribuídos).
- *Engenharia de software:* foi feito um progresso significativo no desenvolvimento de métodos de projeto orientados a objetos, levando ao surgimento da UML (Unified Modelling Language) como notação padrão do setor para especificar sistemas de *software* orientados a objetos (potencialmente distribuídos).

Em outras palavras, por meio da adoção de uma estratégia orientada a objetos, os desenvolvedores de sistemas distribuídos não somente receberam abstrações de programação mais ricas (usando linguagens de programação conhecidas, como C++ e Java), como também puderam utilizar princípios de projeto, ferramentas e técnicas orientados a objetos (incluindo a UML) no desenvolvimento de *software* de sistemas distribuídos. Isso representa um grande avanço em uma área em que, anteriormente, não havia tais técnicas de projeto. É interessante notar que o OMG (Object Management Group), a organização que desenvolveu o CORBA (veja a Seção 8.3) também gerencia a padronização da UML.

O *middleware de objeto distribuído* oferece uma abstração de programação baseada em princípios orientados a objetos. Os principais exemplos de *middleware* de objeto distribuído incluem RMI Java (discutida na Seção 5.5) e CORBA (examinado em profundidade na Seção 8.3, a seguir). Embora RMI Java e CORBA tenham muito em comum, há uma diferença importante: o uso de RMI Java é restrito ao desenvolvimento baseado em Java, enquanto o CORBA é uma solução multilinguagem que permite a operação conjunta de objetos escritos em diversas linguagens. (Existe suporte para C++, Java, Python e várias outras linguagens.)

<i>Objetos</i>	<i>Objetos distribuídos</i>	<i>Descrição de objeto distribuído</i>
Referências de objeto	Referências de objeto remoto	Referências globalmente exclusivas para um objeto distribuído; podem ser passadas como parâmetro.
Interfaces	Interfaces remotas	Fornecem uma especificação abstrata dos métodos que podem ser invocados no objeto remoto; especificadas usando-se uma linguagem de definição de interface (IDL, Interface Definition Language).
Ações	Ações distribuídas	Iniciadas por uma invocação de método, potencialmente resultando em encadeamentos de invocação; as invocações remotas usam RMI.
Exceções	Exceções distribuídas	Exceções adicionais geradas a partir da natureza distribuída do sistema, incluindo perda de mensagens ou falha de processos.
Coleta de lixo	Coleta de lixo distribuída	Esquema ampliado para garantir que um objeto continue a existir, caso exista pelo menos uma referência de objeto ou referência de objeto remoto para esse objeto; caso contrário, ele deve ser removido. Exige um algoritmo de coleta de lixo distribuída.

Figura 8.1 Objetos distribuídos.

Deve-se enfatizar que a programação com objetos distribuídos é diferente e significativamente mais complexa que a programação orientada a objetos padrão, conforme resumido a seguir.

As diferenças: as principais diferenças entre objetos e objetos distribuídos já foram abordadas na Seção 5.4.1, no contexto de RMI. Por conveniência, essa discussão está repetida aqui de forma resumida (veja a Figura 8.1). Outras diferenças surgirão quando examinarmos o CORBA em detalhes, na Seção 8.3. Elas incluem:

- *Classe* é um conceito fundamental nas linguagens orientadas a objetos, mas não é uma característica muito destacada no *middleware* de objeto distribuído. Conforme observado no estudo de caso do CORBA, é difícil concordar com uma interpretação comum de classe em um ambiente heterogêneo, em que várias linguagens coexistem. De uma forma mais geral, no mundo orientado a objetos, a noção de classe tem várias interpretações, incluindo a descrição do comportamento associado a um grupo de objetos (o modelo usado para criar um objeto a partir da classe), o lugar para se instanciar um objeto com determinado comportamento (a fábrica associada) ou mesmo o grupo de objetos que obedece a esse comportamento. Embora o termo “classe” seja evitado, termos mais específicos, como “fábrica” e “modelo”, são usados sem dificuldade (uma fábrica é um objeto que instanciará um novo objeto a partir de determinado modelo).

- O estilo de *herança* é significantivamente diferente do oferecido na maioria das linguagens orientadas a objetos. Em particular, o *middleware* de objeto distribuído oferece *herança de interface*, que é uma relação entre interfaces por meio da qual a nova interface herda assinaturas de método da interface original e pode adicionar outras. Em contraste, linguagens orientadas a objetos, como a Smalltalk, oferecem *herança de implementação* como uma relação entre implementações, por meio da qual a nova classe (neste caso) herda a implementação (e, portanto, o comportamento) da classe original e pode adicionar comportamento extra. É muito mais difícil implementar herança de implementação, particularmente em sistemas distribuídos, devido à necessidade de solucionar o comportamento executável correto em tempo de execução. Considere, por exemplo, o nível de heterogeneidade que pode existir em um sistema distribuído, junto à necessidade de implementar soluções altamente escaláveis.

Wegner, em seu influente artigo sobre linguagens orientadas a objetos [Wegner 1987], define orientação a objetos como objeto + classe + herança. Nos sistemas distribuídos, a interpretação é claramente um pouco diferente, com classe e herança evitadas ou adaptadas. O que resta é um forte enfoque no encapsulamento e na abstração de dados, junto a poderosos vínculos com metodologias de projeto, conforme enfatizado anteriormente.

Complexidades adicionais: devido a complexidades adicionais envolvidas, o *middleware* de objeto distribuído associado deve fornecer mais funcionalidades, conforme resumido a seguir:

Comunicação entre objetos: uma estrutura de *middleware* de objeto distribuído deve fornecer um ou mais mecanismos para os objetos se comunicarem no ambiente distribuído. Normalmente, isso é fornecido pela invocação de método remoto, embora frequentemente o *middleware* de objeto distribuído complementa isso com outros paradigmas de comunicação (por exemplo, estratégias indiretas, como os eventos distribuídos). O CORBA fornece um serviço de eventos e um serviço de notificação associado, ambos implementados como serviços sobre o *middleware* básico (veja a Seção 8.3.4).

Gerenciamento de ciclo de vida: gerenciamento de ciclo de vida relaciona-se à criação, migração e exclusão de objetos, com cada etapa tendo de lidar com a natureza distribuída do ambiente subjacente.

Ativação e desativação: nas implementações não distribuídas, frequentemente pode-se supor que os objetos estão ativos o tempo todo, enquanto o processo que os contém é executado. Contudo, nos sistemas distribuídos, isso não pode ser presumido, pois os números de objetos podem ser muito grandes e, assim, seria desperdício de recursos ter todos os objetos disponíveis a todo momento. Além disso, nós que contêm objetos podem ficar indisponíveis por períodos de tempo. Ativação é o processo de tornar um objeto ativo no ambiente distribuído, fornecendo os recursos necessários para que ele processe as invocações recebidas – efetivamente, localizando o objeto na memória virtual e dando a ele as *threads* necessárias para executar. Desativação é o processo oposto, tornando um objeto temporariamente incapaz de processar invocações.

Persistência: normalmente, os objetos têm estado, e é importante manter esse estado entre os possíveis ciclos de ativação e desativação e em caso de falha do sistema. Portanto, o *middleware* de objeto distribuído deve oferecer gerenciamento de persistência para objetos com estado.

Serviços adicionais: uma estrutura de *middleware* de objeto distribuído abrangente também deve fornecer suporte para os diversos serviços de sistema distribuído considerados neste livro, incluindo serviços de atribuição de nomes, segurança e transação.

8.3 Estudo de caso: CORBA

O OMG (Object Management Group) foi formado em 1989, com o objetivo de estimular a adoção de sistemas de objetos distribuídos para usufruir as vantagens da programação orientada a objetos no desenvolvimento de *software* para sistemas distribuídos, os quais estavam se tornando muito difundidos. Para atingir seus objetivos, o OMG defendeu o uso de sistemas abertos baseados em interfaces orientadas a objetos padrão. Esses sistemas seriam construídos a partir de *hardware*, redes de computadores, sistemas operacionais e linguagens de programação heterogêneos.

Uma motivação importante foi permitir que os objetos distribuídos fossem implementados em qualquer linguagem de programação e pudessem se comunicar uns com os outros. Portanto, foi projetada uma linguagem de interface independente de qualquer linguagem de implementação específica.

Foi introduzida uma metáfora, o ORB (ou object request broker – agente de requisição de objeto), cuja função é ajudar um cliente a invocar um método em um objeto (segundo o estilo RMI, conforme apresentado no Capítulo 5). Essa função envolve localizar o objeto, invocá-lo se necessário e, então, comunicar a requisição do cliente para o objeto, o qual o executa e responde.

Esta seção apresenta um estudo de caso do CORBA (Common Object Request Broker Architecture) do OMG, complementando esse conceito de agente de requisição de objeto. A apresentação se concentra na especificação CORBA 2 (a principal inovação em seu sucessor, CORBA 3, é a introdução de um modelo de componente, conforme discutido na Seção 8.4).

Os principais componentes do *framework* de RMI CORBA independente de linguagem são:

- Uma linguagem de definição de interface conhecida como IDL, que será descrita com detalhes na Seção 8.3.1.
- Uma arquitetura, que será discutida na Seção 8.3.2.
- Uma representação externa de dados, chamada CDR, que foi descrita na Seção 4.3.1. Ela também define formatos específicos para as mensagens em um protocolo de requisição-resposta. Além das mensagens de requisição-resposta, ela especifica mensagens para fazer perguntas sobre a localização de um objeto, cancelar requisições e relatar erros.
- Uma forma padrão para referências de objeto remoto, que será descrita na Seção 8.3.3.

A arquitetura CORBA também admite serviços CORBA – um conjunto de serviços genéricos úteis para aplicações distribuídas. Eles serão apresentados brevemente na Seção 8.3.4 (uma versão mais completa deste estudo de caso, incluindo uma consideração detalhada dos serviços do CORBA, pode ser encontrada no *site* que acompanha o livro [www.cdk5.net] – em inglês).

A Seção 8.3.5 também contém um exemplo de desenvolvimento de cliente-servidor usando CORBA.

Para um conjunto interessante de artigos sobre CORBA, consulte a edição especial do *CACM* [Seetharaman 1998].

8.3.1 RMI CORBA

Programar em um sistema RMI de multilinguagens, como RMI CORBA, exige mais do programador se comparado com o desenvolvimento em um sistema RMI de linguagem única, como RMI Java. Os seguintes novos conceitos precisam ser aprendidos:

- o modelo de objeto oferecido pelo CORBA;
- a linguagem de definição de interface;
- seu mapeamento para a linguagem de implementação.

Outros aspectos da programação em CORBA são semelhantes aos discutidos no Capítulo 5. Em particular, o programador define interfaces remotas para os objetos remotos e depois usa um compilador de interface para produzir os *proxies* e esqueletos correspondentes. No entanto, no CORBA, os *proxies* são gerados na linguagem do cliente e os esqueletos, na linguagem do servidor.

Modelo de objeto do CORBA • O modelo de objeto do CORBA é semelhante àquele descrito na Seção 5.4.1, mas os clientes não são necessariamente objetos – um cliente pode ser qualquer programa que envie mensagens de requisição para objetos remotos e receba respostas. O termo *objeto CORBA* é usado para se referir aos objetos remotos. Assim, um objeto CORBA implementa uma interface IDL, tem uma referência de objeto remoto e é capaz de responder às invocações de métodos em sua interface IDL. Um objeto CORBA pode, por exemplo, ser implementado por uma linguagem não orientada a objeto, sem o conceito de classe. Como as linguagens de implementação terão diferentes noções de classe, ou mesmo nenhuma noção, o conceito de classe não existe no CORBA (veja também a discussão na Seção 8.2). Portanto, classes não podem ser definidas na IDL do CORBA, o que significa que instâncias de classes não podem ser passadas como argumentos. Entretanto, estruturas de dados de vários tipos e complexidade arbitrária podem ser passadas como argumentos.

IDL CORBA • A IDL do CORBA especifica um nome e um conjunto de métodos que os clientes podem solicitar. Como um exemplo inicial, a Figura 8.2 mostra duas interfaces, chamadas *Shape* (linha 3) e *ShapeList* (linha 5), que são versões de IDL das interfaces definidas na Figura 5.16. Elas são precedidas por definições de duas estruturas (*structs*), as quais são usadas como tipos de parâmetro na definição dos métodos. Note, em particular, que *GraphicalObject* é definido como *struct*, embora fosse uma classe no exemplo de RMI em Java. Um componente cujo tipo é *struct* tem um conjunto de campos contendo valores de diferentes tipos, como as variáveis de instância de um objeto, mas não tem métodos.

Mais detalhadamente, a linguagem de definição de interface – ou simplesmente IDL – CORBA fornece recursos para a definição de módulos, interfaces, tipos, atributos e assinaturas de método. Podemos ver exemplos de tudo isso, fora os módulos, nas Figuras 5.8 e 8.2. A IDL CORBA tem as mesmas regras léxicas da linguagem C++, mas possui palavras-chave adicionais para suportar distribuição, por exemplo, *interface*, *any*, *attribute*, *in*, *out*, *inout*, *readonly* e *raises*. Ela também permite recursos de pré-processamento padrão da linguagem C++. Veja, por exemplo, o *typedef* para *All* na Figura 8.3.

A gramática da IDL é um subconjunto da linguagem C++ ANSI, com construções adicionais para suportar assinaturas de método. Fornecemos, aqui, apenas um

```

struct Rectangle{
    long width;
    long height;
    long x;
    long y;
}
struct GraphicalObject {
    string type;
    Rectangle enclosing;
    boolean isFilled;
}
interface Shape {
    long getVersion();
    GraphicalObject getAllState(); // retorna o estado do objeto GraphicalObject
}
typedef sequence <Shape, 100> All;
interface ShapeList {
    exception FullException();
    Shape newShape(in GraphicalObject g) raises (FullException);
    All allShapes(); // retorna a sequência de referências de objeto remoto
    long getVersion();
}

```

Figura 8.2 Interfaces IDL *Shape* e *ShapeList*.

breve panorama da IDL. Uma visão geral útil e exemplos são dados em Baker [1997] e Henning e Vinoski [1999]. A especificação completa está disponível no *site* do OMG [[OMG 2002a](#)].

Módulos IDL: a construção do módulo permite que as interfaces e outras definições de tipo da IDL sejam agrupadas em unidades lógicas. Um *módulo* define um escopo de atribuição de nomes, o qual impede que os nomes definidos dentro dele colidam com os nomes definidos fora dele. Por exemplo, as definições das interfaces *Shape* e *ShapeList* poderiam pertencer a um módulo chamado *Whiteboard*, como mostrado na Figura 8.3.

Interfaces IDL: conforme já vimos, uma interface IDL descreve os métodos que estão disponíveis nos objetos CORBA que implementam essa interface. Os clientes de um objeto CORBA podem ser desenvolvidos apenas a partir do conhecimento de sua interface IDL. A partir do estudo de nossos exemplos, os leitores verão que uma interface define um conjunto de operações e atributos e, geralmente, depende de um conjunto de tipos definidos com ela. Por exemplo, a interface *PersonList*, na Figura 5.8, define um atributo e três métodos e depende do tipo *Person*.

Métodos IDL: a forma geral de uma assinatura de método é a seguinte:

[oneway] <tipo_retorno> <nome_método> (parâmetro1,...,parâmetroL)
[raises (exceção1,..., exceçãoN)] [contexto (nome1,..., nomeM)]

onde as expressões entre colchetes são opcionais. Para um exemplo de assinatura de método que contenha apenas as partes exigidas, considere:

void getPerson(in string name, out Person p);

Os parâmetros são rotulados como *in*, *out* ou *inout*, onde o valor de um parâmetro *in* é passado do cliente para o objeto CORBA invocado e o valor de um parâmetro *out* é pas-

```

module Whiteboard {
    struct Rectangle{
        ...};
    struct GraphicalObject {
        ...};
    interface Shape {
        ...};
    typedef sequence <Shape, 100> All;
    interface ShapeList {
        ...};
}

```

Figura 8.3 Módulo de IDL *Whiteboard*.

sado de volta do objeto CORBA invocado para o cliente. Raramente são usados parâmetros rotulados como *inout*, mas eles indicam que o valor do parâmetro pode ser passado nas duas direções. O tipo de retorno pode ser especificado como *void*, caso nenhum valor deva ser retornado. A Figura 5.8 ilustra um exemplo simples do uso dessas palavras-chaves. Na Figura 8.2, linha 7, o parâmetro de *newShape* é um parâmetro *in*, para indicar que o argumento deve ser passado do cliente para o servidor na mensagem de requisição. O valor de retorno de um método fornece um parâmetro *out* adicional – ele pode ser indicado como *void*, caso não exista nenhum parâmetro *out*.

Os parâmetros podem ser qualquer um dos tipos primitivos, como *long* e *boolean*, ou um dos tipos construídos, como *struct* ou *array* (mais informações sobre tipos IDL primitivos e estruturados podem ser encontradas a seguir). Nossa exemplo mostra as definições de dois tipos *struct*, nas linhas 1 e 2. Sequências e vetores são definidos em *typedefs*, como se vê na linha 4, que mostra uma sequência de elementos de tipo *Shape* de tamanho 100. A semântica da passagem de parâmetros é a seguinte:

Passagem de objetos CORBA: qualquer parâmetro cujo tipo seja especificado pelo nome de uma interface IDL, como o valor de retorno *Shape* na linha 7, é uma referência para um objeto CORBA, e é passado o valor de uma referência de objeto remoto.

Passagem de tipos primitivos e construídos CORBA: os argumentos de tipos primitivos e construídos são copiados e passados por valor. Na chegada, um novo valor é criado no processo do destinatário. Por exemplo, o *struct GraphicalObject* passado como argumento (na linha 7) produz uma nova cópia desse *struct* no servidor.

Essas duas formas de passagem de parâmetro são combinadas no método *allShapes* (na linha 8), cujo tipo de retorno é um vetor de tipo *Shape* – isto é, um vetor de referências de objeto remoto. O valor de retorno é uma cópia do vetor na qual cada um dos elementos é uma referência de objeto remoto.

Semântica de invocação. A invocação remota no CORBA tem como padrão a semântica de chamada *no máximo uma vez*. Entretanto, a IDL pode especificar que a invocação de um método em particular tenha semântica *talvez*, usando a palavra-chave *oneway*. O cliente não é bloqueado em requisições *oneway*, que só podem ser usadas por métodos sem resultados. Para um exemplo de uma requisição *oneway*, veja o que lida com *callbacks* no final da Seção 8.3.5.

Exceções na IDL CORBA: a IDL CORBA permite que exceções sejam definidas nas interfaces e disparadas por seus métodos. A expressão opcional *raises* indica as exceções definidas pelo usuário que podem ser disparadas para terminar uma execução do método. Considere o exemplo a seguir da Figura 8.2:

```
exception FullException{ };
Shape newShape(in GraphicalObject g) raises (FullException);
```

Com a expressão *raises*, o método *newShape* especifica que ele pode disparar uma exceção chamada *FullException*, que está definida dentro da interface *ShapeList*. Em nosso exemplo, a exceção não contém variáveis. Entretanto, as exceções podem ser definidas para conter variáveis, por exemplo:

```
exception FullException {GraphicalObject g};
```

Quando uma exceção que contém variáveis é disparada, o servidor pode usar as variáveis para retornar informações para o cliente sobre o contexto da exceção.

O CORBA também pode produzir exceções de sistema relacionadas a problemas nos servidores (como o fato de estarem ocupados demais ou não poderem ativar objetos), problemas com a comunicação e problemas no lado do cliente. Os programas clientes devem tratar de exceções definidas pelo usuário e de sistema. A expressão opcional *context* é usada para fornecer mapeamentos de nomes de *string* para valores de *string*. Consulte Baker [1997] para uma explicação de *context*.

Tipos da IDL: a IDL suporta 15 tipos primitivos, os quais incluem *short* (16 bits), *long* (32 bits), *unsigned short*, *unsigned long*, *float* (32 bits), *double* (64 bits), *char*, *boolean* (TRUE, FALSE), *octet* (8 bits) e *any* (que pode representar qualquer tipo primitivo ou construído). Constantes da maioria dos tipos primitivos e *strings* podem ser declaradas usando-se a palavra-chave *const*. A IDL fornece um tipo especial chamado *Object*, cujos valores são referências de objeto remoto. Se um parâmetro ou resultado é de tipo *Object*, então o argumento correspondente pode se referir a qualquer objeto CORBA.

Os tipos construídos da IDL estão descritos na Figura 8.4, todos os quais são passados por valor em argumentos e resultados. Todos os vetores ou sequências usados como argumentos devem ser definidos em *typedefs*. Nenhum dos tipos de dados primitivos ou construídos pode conter referências.

O CORBA também suporta passagem por valor de objetos que não são CORBA [OMG 2002c]. Esses objetos que não são CORBA são parecidos com objetos, pois possuem atributos e métodos. Entretanto, eles são puramente objetos locais, pois suas operações não podem ser invocadas de forma remota. O recurso de passagem por valor fornece a capacidade de passar uma cópia de um objeto que não é CORBA entre cliente e servidor.

Isso é obtido pela adição, na IDL, de um tipo chamado *valuetype* para representar objetos que não são CORBA. Um *valuetype* é uma *struct* com assinaturas de método adicionais (como as de uma interface) e os argumentos e resultados *valuetype* são passados por valor. Isto é, o estado é passado para o *site* remoto e usado para produzir um novo objeto no destino.

Os métodos desse novo objeto podem ser invocados de forma local, fazendo seu estado divergir do estado do objeto original. Passar a implementação dos métodos não é tão simples assim, pois o cliente e o servidor podem usar linguagens diferentes. Entretanto, se o cliente e o servidor forem implementados em Java, o código poderá ser baixado

<i>Tipo</i>	<i>Exemplos</i>	<i>Uso</i>
<i>sequence</i>	<i>typedef sequence <Shape, 100> All;</i> <i>typedef sequence <Shape> All;</i> sequências limitadas e não limitadas de <i>Shapes</i>	Define um tipo para uma sequência de elementos de tamanho variável de um tipo IDL especificado. Pode ser especificado um limite superior para o tamanho.
<i>string</i>	<i>string name; typedef string<8> SmallString;</i> sequências limitadas e não limitadas de caracteres	Define uma sequência de caracteres, terminada pelo caractere nulo. Pode ser especificado um limite superior para o comprimento.
<i>array</i>	<i>typedef octet uniqueId[12];</i> <i>typedef GraphicalObject GO[10][8];</i>	Define um tipo para uma sequência multidimensional de elementos de tamanho fixo de um tipo IDL especificado.
<i>record</i>	<i>struct GraphicalObject {</i> <i>string type;</i> <i>Rectangle enclosing;</i> <i>boolean isFilled;</i> <i>};</i>	Define um tipo para um registro contendo um grupo de entidades relacionadas. <i>Structs</i> são passadas por valor em argumentos e resultados.
<i>enumerated</i>	<i>enum Rand</i> <i>(Exp, Number, Name);</i>	O tipo enumerado na IDL faz o mapeamento de um nome de tipo para um conjunto de valores inteiros.
<i>union</i>	<i>union Exp switch (Rand) {</i> <i>case Exp: string vote;</i> <i>case Number: long n;</i> <i>case Name: string s;</i> <i>};</i>	A união da IDL permite que um tipo de determinado conjunto de tipos seja passado como argumento. O cabeçalho é parametrizado por um <i>enum</i> , que especifica qual membro é usado.

Figura 8.4 Tipos construídos da IDL.

por *download*. Para uma implementação em C++ comum, o código necessário precisaria estar presente no cliente e no servidor.

Esse recurso é útil quando for vantajoso colocar uma cópia de um objeto no processo cliente para permitir que ele receba invocações locais. Entretanto, ele nem chega perto da passagem por valor de objetos CORBA.

Atributos: as interfaces IDL podem ter atributos, assim como métodos. Os atributos são como os campos de classe pública em Java. Eles podem ser definidos como *readonly*, onde for apropriado. Os atributos são privativos dos objetos CORBA, mas, para cada atributo declarado, dois métodos de acesso são gerados automaticamente pelo compilador de IDL: um para recuperar o valor do atributo e o outro para configurá-lo. Para atributos *readonly* é fornecido apenas o método de obtenção (*getter*). Por exemplo, a interface *PersonList* definida na Figura 5.2 inclui a seguinte definição de atributo:

readonly attribute string listname;

Herança: as interfaces IDL podem ser estendidas por meio de herança, conforme definido na Seção 8.2. Por exemplo, se a interface *B* estende a interface *A*, isso significa que ela pode adicionar novos tipos, constantes, exceções, métodos e atributos naqueles de *A*. Uma interface estendida pode redefinir tipos, constantes e exceções, mas não pode redefinir métodos. O valor de um tipo estendido é válido como valor de um parâmetro ou resultado do tipo ascendente. Por exemplo, o tipo *B* é válido como valor de um parâmetro

ou resultado do tipo *A*. Além disso, uma interface IDL pode estender mais de uma interface. Por exemplo, aqui, a interface *Z* estende *B* e *C*:

```
interface A {};
interface B: A {};
interface C {};
interface Z: B, C {};
```

Isso significa que *Z* tem todos os componentes de *B* e de *C* (além daqueles que ela redefine), assim como os que define como extensões.

Quando uma interface como *Z* estende mais de uma interface, existe a possibilidade de que ela possa herdar um tipo, constante ou exceção com o mesmo nome, de duas interfaces diferentes. Por exemplo, suponha que tanto *B* como *C* definam um tipo chamado *Q*; então, o uso de *Q* na interface *Z* será ambíguo, a menos que seja dado um nome com escopo, como *B::Q* ou *C::Q*. A IDL não permite que uma interface herde métodos ou atributos com nomes comuns de duas interfaces diferentes.

Todas as interfaces IDL herdam do tipo *Object*, o que implica que todas as interfaces IDL são compatíveis com o tipo *Object* – o que inclui referências de objeto remoto. Isso torna possível definir operações de IDL que podem receber como argumento ou retornar como resultado uma referência de objeto remoto de qualquer tipo. As operações *bind* e *resolve* no serviço de nomes (Naming Service) são exemplos disso.

Identificadores de tipo IDL: conforme veremos na Seção 8.3.2, os identificadores de tipo são gerados pelo compilador de IDL para cada tipo em uma interface IDL. Por exemplo, o tipo IDL para a interface *Shape* (Figura 8.3) poderia ser:

IDL:Whiteboard/Shape:1.0

Esse exemplo mostra que um nome de tipo da IDL tem três partes – o prefixo IDL, um nome de tipo e um número de versão. Como identificadores de interface são usados como chaves para acessar definições de interface no repositório de interfaces (descrito na Seção 8.3.2), os programadores devem garantir o fornecimento de um mapeamento único para as próprias interfaces. Os programadores podem usar o prefixo IDL *pragma* para prefixar um *string* adicional no nome de tipo, para distinguir seus próprios tipos dos de outros.

Diretivas *pragma* da IDL: elas permitem que propriedades adicionais que não pertencem à IDL sejam especificadas para componentes em uma interface IDL (consulte Henning e Vinoski [1999]). Essas propriedades incluem, por exemplo, especificar que uma interface será usada apenas de forma local ou fornecer o valor de um ID de repositório de interfaces. Cada *pragma* é introduzido por *#pragma* e especifica seu tipo, por exemplo:

#pragma version Whiteboard 2.3

Mapeamentos de linguagem CORBA • O mapeamento dos tipos na IDL para tipos de determinada linguagem de programação é muito simples. Por exemplo, em Java, os tipos primitivos na IDL são mapeados nos tipos primitivos correspondentes dessa linguagem. *Structs*, *enums* e *unions* são mapeados em classes Java; sequências e *arrays* na IDL são mapeados em vetores em Java. Uma exceção da IDL é mapeada em uma classe Java que fornece variáveis de instância para os campos da exceção e para os construtores. Os mapeamentos em C++ são semelhantemente simples.

Entretanto, surgem algumas dificuldades no mapeamento da semântica da passagem de parâmetros da IDL para a linguagem Java. Em particular, a IDL permite que os

métodos retornem diversos valores separados por meio de parâmetros de saída, enquanto a linguagem Java só pode ter um único resultado. As classes *Holder* são fornecidas para resolver essa diferença, mas isso exige que o programador as utilize, o que não é nada simples. Por exemplo, o método *getPerson* da Figura 5.8 é definido na IDL como segue:

```
void getPerson(in string name, out Person p);
```

Em Java, o método equivalente seria definido como:

```
void getPerson(String name, PersonHolder p);
```

e o cliente teria de fornecer uma instância de *PersonHolder* como argumento de sua invocação. A classe portadora tem uma variável de instância que contém o valor do argumento para o cliente acessar por RMI quando a invocação retornar. Ela também tem métodos para transmitir o argumento entre servidor e cliente.

Embora as implementações em C++ do CORBA possam manipular parâmetros *out* e *inout* de forma bastante natural, os programadores de C++ sofrem com um conjunto de problemas diferentes nos parâmetros, relacionados ao gerenciamento do armazenamento. Essas dificuldades surgem quando referências de objeto e entidades de comprimento variável, como *strings* ou sequências, são passadas como argumentos.

Por exemplo, no Orbix [Baker 1997], o ORB mantém contagens de referência para objetos remotos e *proxies* e as libera quando elas não são mais necessárias. Ele fornece aos programadores métodos para liberá-las ou duplicá-las. Quando um método de servidor termina de executar, os argumentos e resultados *out* são liberados, e o programador deve duplicá-los, caso ainda sejam necessários. Por exemplo, um servente em C++ implementando a interface *ShapeList* precisará duplicar as referências retornadas pelo método *allShapes*. As referências de objeto passadas para os clientes devem ser liberadas quando não forem mais necessárias. Regras semelhantes se aplicam aos parâmetros de tamanho variável.

Em geral, os programadores que usam a IDL não apenas precisam aprender a própria notação da IDL como devem entender como seus parâmetros são mapeados nos parâmetros da linguagem de implementação.

RMI assíncrona • O CORBA suporta uma forma de RMI assíncrona que permite aos clientes fazerem invocações não bloqueantes nos objetos CORBA [OMG 2004e]. Ela se destina a ser implementada no cliente. Portanto, um servidor geralmente não sabe se ela é invocada de forma síncrona ou de forma assíncrona. (Uma exceção é o *Transaction Service*, que precisaria saber da diferença.)

A RMI assíncrona adiciona duas novas variantes na semântica de invocação das RMIs:

- *callback*, na qual um cliente usa um parâmetro extra para passar uma referência para uma *callback* com cada invocação, de modo que o servidor possa chamar de volta com os resultados;
- *consulta sequencial*, na qual o servidor retorna um objeto *valuetype*, que pode ser usado para fazer uma consulta sequencial ou esperar pela resposta.

A arquitetura da RMI assíncrona permite que um agente intermediário seja implantado para garantir que a requisição seja executada e, se necessário, armazene a resposta. Assim, ela é apropriada para uso em ambientes onde os clientes podem ser temporariamente desconectados – como, por exemplo, um cliente usando um *notebook* em um trem.

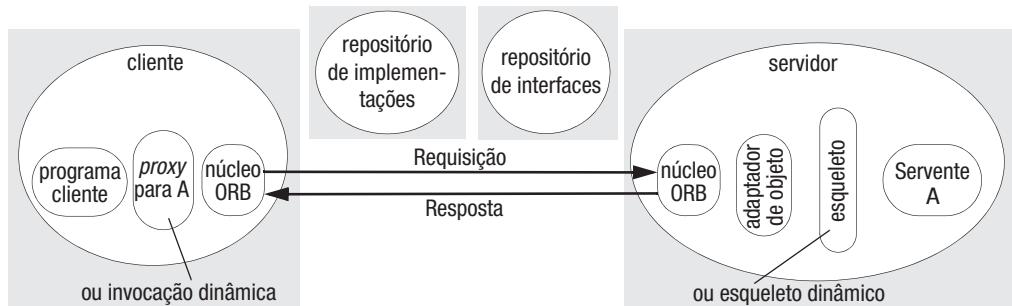


Figura 8.5 Os principais componentes da arquitetura CORBA.

8.3.2 A arquitetura CORBA

A arquitetura é projetada para suportar a função de um agente de requisição de objeto (ORB, Object Request Broker) que permite aos clientes invocar métodos em objetos remotos, em que clientes e servidores podem ser implementados em uma variedade de linguagens de programação. Os principais componentes da arquitetura CORBA estão ilustrados na Figura 8.5.

Essa figura deve ser comparada com a Figura 5.15, para que se note que a arquitetura CORBA contém três componentes adicionais: o adaptador de objeto, o repositório de implementações e o repositório de interfaces.

O CORBA fornece invocações estáticas e dinâmicas. As invocações estáticas são usadas quando a interface remota do objeto CORBA é conhecida no momento da compilação, permitindo o uso de *stubs* de cliente e esqueletos de servidor. Se a interface remota não for conhecida no momento da compilação, a invocação dinâmica deverá ser usada. A maioria dos programadores prefere usar invocação estática, pois ela fornece um modelo de programação mais natural.

Discutiremos, agora, os componentes da arquitetura, deixando as preocupações com a invocação dinâmica por último.

Núcleo ORB • A função do núcleo ORB inclui toda a funcionalidade do módulo de comunicação da Figura 5.15. Além disso, um núcleo ORB fornece uma interface com o seguinte:

- operações que o permitem ser iniciado e parado;
- operações para fazer a conversão entre referências de objeto remoto e *strings*;
- operações para fornecer listas de argumentos para requisições usando invocação dinâmica.

Adaptador de objeto • A função de um *adaptador de objeto* é fazer a ligação entre objetos CORBA com interfaces IDL e as interfaces da linguagem de programação das classes serventes correspondentes. Essa função também inclui a dos módulos de referência remota e despachante na Figura 5.15. Um adaptador de objeto tem as seguintes tarefas:

- Ele cria referências de objeto remoto para objetos CORBA (veja a Seção 8.3.3).
- Ele envia cada RMI, por meio de um esqueleto, para o servente apropriado.
- Ele ativa e desativa serventes.

Um adaptador de objeto fornece para cada objeto CORBA um *nome de objeto* exclusivo, o qual faz parte de sua referência de objeto remoto. O mesmo nome é usado sempre que um objeto é ativado. O nome do objeto pode ser especificado pelo programa aplicativo ou gerado pelo adaptador de objeto. Cada objeto CORBA é registrado em seu adaptador de objeto, o qual mantém uma tabela de objetos remotos que faz o mapeamento dos nomes de objetos CORBA para seus serventes.

Cada adaptador de objeto tem seu próprio nome, o qual faz parte das referências de objeto remoto de todos os objetos CORBA que gerencia. Esse nome pode ser especificado pelo programa aplicativo ou gerado automaticamente.

Adaptador de objeto portável • O padrão CORBA para adaptador de objetos é chamado de *Portable Object Adapter* (POA). Ele é chamado portável porque permite que aplicações e serventes sejam executados em ORBs produzidos por diferentes desenvolvedores [Vinoski 1998]. Isso é obtido por meio da padronização das classes de esqueleto e das interações entre o POA e os serventes.

O POA suporta objetos CORBA com dois tipos diferentes de tempos de vida:

- aqueles cujos tempos de vida são restritos ao processo em que seus serventes são instanciados;
- aqueles cujos tempos de vida podem abranger as instanciações de serventes em vários processos.

Os primeiros têm referências de objeto transientes e os últimos têm referências de objeto persistentes (veja a Seção 8.3.3).

O POA permite que objetos CORBA sejam instanciados de forma transparente e, além disso, separa a criação de objetos CORBA da criação dos *serventes* que implementam esses objetos. Aplicações servidoras, como bancos de dados, com grandes números de objetos CORBA, podem criar serventes sob demanda, apenas quando os objetos são acessados. Neste caso, elas podem usar chaves de banco de dados para os nomes de objeto ou podem usar um único servente para suportar todos esses objetos.

Além disso, é possível especificar políticas para o POA; por exemplo, se ele deve fornecer uma *thread* para cada invocação, se as referências de objeto devem ser persistentes ou transientes e se deve haver um servente separado para cada objeto CORBA. O padrão é que um único servente possa representar todos os objetos CORBA para seu POA.

Note que as implementações de CORBA fornecem interfaces para a funcionalidade do POA e do núcleo ORB por meio de *pseudo-objetos*, os quais recebem esse nome porque não podem ser usados como objetos CORBA normais; por exemplo, eles não podem ser passados como argumentos em RMIs. Contudo, eles podem ter interfaces IDL e são implementados como bibliotecas. O pseudo-objeto POA inclui, por exemplo, um método para ativar um *POAmanager* e outro, *servant_to_reference*, para registrar um objeto CORBA. O pseudo-objeto ORB inclui o método *init*, que deve ser chamado para inicializar o ORB, o método *resolve_initial_references*, que é usado para localizar serviços como o serviço de nomes e o POA raiz, e outros métodos, que possibilitam fazer conversões entre referências de objeto remoto e *strings*.

Esqueletos • As classes de esqueleto são geradas na linguagem do servidor por um compilador de IDL. Como descrito na Seção 5.4.2, as invocações de método remotas são enviadas por meio do esqueleto apropriado para um servente em particular, e o esqueleto desempacota os argumentos nas mensagens de requisição e empacota exceções e resultados nas mensagens de resposta.

Stubs/proxies de cliente • Eles são escritos na linguagem do cliente. A classe de um *proxy* (para linguagens orientadas a objetos) ou um conjunto de procedimentos de *stub* (para linguagens procedurais) é gerada a partir de uma interface IDL por um compilador de IDL para a linguagem do cliente. Como antes, os *stubs/proxies* de cliente empacotam os argumentos nas requisições de invocação e desempacotam exceções e resultados nas respostas.

Repositório de implementações • Um repositório de implementações é responsável por ativar servidores registrados sob demanda e por localizar os servidores que estão correntemente em execução. O nome do adaptador de objeto é usado para se referir aos servidores ao registrá-los e ativá-los.

Um repositório de implementações armazena um mapeamento dos nomes de adaptadores de objeto para nomes de caminho de arquivos contendo implementações de objeto. As implementações de objeto e os nomes de adaptadores de objeto geralmente são registrados no repositório de implementações quando os programas servidores são instalados. Quando as implementações de objeto são ativadas nos servidores, o nome de computador servidor (*hostname*) e o seu número da porta são adicionados no mapeamento:

Entrada do repositório de implementações:

nome do adaptador de objeto	nome de caminho da implementação de objeto	<i>hostname</i> e número da porta do servidor
-----------------------------	--	---

Nem todos os objetos CORBA precisam ser ativados sob demanda. Alguns deles, por exemplo, objetos de *callback* criados pelos clientes, são executados uma vez e deixam de existir quando não são mais necessários. Eles não usam o repositório de implementações.

Geralmente, um repositório de implementações permite que sejam armazenadas informações extras sobre cada servidor, como informações de controle de acesso sobre quem pode ativá-lo ou executar suas operações. É possível replicar informações nos repositórios de implementação para fornecer disponibilidade ou tolerância a falhas.

Repositório de interfaces • A função do repositório de interfaces é fornecer informações sobre as interfaces IDL registradas para clientes e servidores que as exigirem. Para uma interface de determinado tipo, ele pode fornecer os nomes dos métodos e, para cada método, os nomes e tipos dos argumentos e exceções. Assim, o repositório de interfaces adiciona um recurso de reflexão no CORBA. Suponha que um programa cliente receba uma referência remota para um novo objeto CORBA. Se o cliente não tem *proxy* para ele, pode perguntar ao repositório de interfaces sobre os métodos do objeto e os tipos de parâmetro que cada um deles exige.

Quando um compilador de IDL processa uma interface, ele atribui um identificador de tipo para cada tipo de IDL que encontra. Para cada interface registrada nele, o repositório de interfaces fornece um mapeamento entre o identificador de tipo dessa interface e a interface em si. Assim, o identificador de tipo de uma interface é também chamado de *ID de repositório*, pois pode ser usado como uma chave para as interfaces IDL registradas no repositório de interfaces.

Toda referência de objeto remoto CORBA inclui uma entrada (*slot*) que contém o identificador de tipo de sua interface, permitindo que os clientes que o possuem perguntem sobre seu tipo no repositório de interfaces. Essas aplicações que usam invocação estática (normal) com *proxies* clientes e esqueletos de IDL não exigem repositório de interfaces. Nem todos os ORBs fornecem repositório de interfaces.

Interface de invocação dinâmica • Conforme sugerido na Seção 5.5, em algumas aplicações pode ser necessário construir um programa cliente sem conhecer todas as classes proxy de que ele precisará no futuro. Por exemplo, talvez um navegador de objeto precise exibir informações sobre todos os objetos CORBA disponíveis nos vários servidores em um sistema distribuído. Não é exequível que tal programa inclua *proxies* para todos esses objetos, particularmente porque, à medida que o tempo passa, novos objetos podem ser adicionados no sistema. O CORBA não permite baixar classes de *proxies* por *download* em tempo de execução, como RMI Java. A interface de invocação dinâmica é a alternativa do CORBA.

A interface de invocação dinâmica permite que os clientes façam invocações dinâmicas em objetos remotos CORBA. Ela é usada quando não é possível empregar *proxies*. O cliente pode obter do repositório de interfaces as informações necessárias sobre os métodos disponíveis para determinado objeto CORBA e pode usar essas informações para construir uma invocação com argumentos convenientes e enviá-la para o servidor.

Esqueletos dinâmicos • Novamente, conforme explicado na Seção 5.5, pode ser necessário adicionar a um servidor um objeto CORBA cuja interface era desconhecida quando o servidor foi compilado. Se um servidor usa esqueletos dinâmicos, ele pode aceitar invocações na interface de um objeto CORBA para as quais não possui esqueleto. Quando um esqueleto dinâmico recebe uma invocação, ele inspeciona o conteúdo da requisição para descobrir seu objeto de destino, o método a ser invocado e os argumentos. Então, ele ativa o destino.

Código legado • O termo *código legado* se refere a código existente que não foi projetado tendo-se em mente os objetos distribuídos. Um código legado pode ser transformado em um objeto CORBA definindo-se uma interface IDL para ele e fornecendo-se a implementação de um adaptador de objeto apropriado e os esqueletos necessários.

8.3.3 Referências de objeto remoto CORBA

O CORBA especifica um formato para referências de objeto remoto conveniente para uso, seja o objeto remoto invocado por um repositório de implementações ou não. As referências que usam esse formato são chamadas de *referências de objeto interoperáveis (IORs, Interoperable Object References)*. A figura a seguir é baseada em Henning [1998], que contém um relato mais detalhado das IORs:

Formato IOR

ID de tipo de interface IDL	Protocolo e detalhes do endereço			Chave de objeto	
identificador de repositório de interfaces	IIOP	nome de domínio do host	número da porta	nome do adaptador	nome do objeto

Seguindo a sequência dos vários campos:

- O primeiro campo de uma IOR especifica o identificador de tipo da interface IDL do objeto CORBA. Note que, se o ORB tiver um repositório de interfaces, esse nome de tipo também será o identificador do repositório de interfaces da interface IDL, o qual permite recuperar a definição de IDL da interface em tempo de execução.
- O segundo campo especifica o protocolo de transporte e os detalhes exigidos por esse protocolo de transporte específico para identificar o servidor. Em particular, o protocolo IIOP (Internet Inter-ORB) usa TCP, no qual o endereço do servidor

consiste em um nome de domínio de computador (*host*) e um número de porta [OMG 2004a].

- O terceiro campo é usado pelo ORB para identificar um objeto CORBA. Ele consiste no nome de um adaptador de objeto no servidor e no nome de um objeto CORBA, especificado pelo adaptador de objeto.

As *IORs transientes* dos objetos CORBA duram apenas tanto quanto o processo que contém esses objetos, enquanto as *IORs persistentes* duram entre as invocações dos objetos CORBA. Uma IOR transiente contém os detalhes do endereço do servidor que contém o objeto CORBA, enquanto uma IOR persistente contém os detalhes do endereço do repositório de implementações no qual está registrada. Nos dois casos, o cliente ORB envia a mensagem de requisição para o servidor cujos detalhes do endereço são dados na IOR. Aqui está como a IOR é usada para localizar o servente que representa o objeto CORBA nos dois casos:

IORs transientes: o núcleo ORB servidor recebe a mensagem de requisição contendo o nome do adaptador de objeto e o nome do objeto do destino. Ele usa o nome do adaptador de objeto para localizar o adaptador de objeto, o qual utiliza o nome do objeto para localizar o servente.

IORs persistentes: um repositório de implementações recebe a requisição. Ele extrai o nome do adaptador de objeto da IOR presente na requisição. Desde que o nome do adaptador de objeto esteja em sua tabela, ele tenta, se necessário, ativar o objeto CORBA no endereço de *host* especificado em sua tabela. Uma vez que o objeto CORBA tenha sido ativado, o repositório de implementações retorna seus detalhes de endereço para o cliente ORB, o qual os utiliza como destino das mensagens de requisição RMI, que incluem o nome do adaptador de objeto e o nome do objeto. Isso permite que o núcleo ORB servidor localize o adaptador de objeto, o qual, como antes, utiliza o nome do objeto para localizar o servente.

O segundo campo de uma IOR pode ser repetido, de modo a especificar o nome de domínio do *host* e o número da porta de mais de um destino, para permitir que um objeto ou um repositório de implementações seja replicado em vários locais diferentes.

A mensagem de resposta no protocolo de requisição-resposta inclui informações de cabeçalho que permitem a execução do procedimento das IORs persistentes. Em particular, ela inclui uma entrada de *status* que pode indicar se a requisição deve ser encaminhada para um servidor diferente, no caso em que o corpo da resposta inclui uma IOR que contém o endereço do servidor do objeto recentemente ativado.

8.3.4 Serviços CORBA

O CORBA inclui especificações de serviços que podem ser exigidos por objetos distribuídos. Em particular, o serviço de nomes (*Naming Service*) é uma adição essencial em qualquer ORB, como veremos em nosso exemplo de programação na Seção 8.3.5. Um índice para documentação sobre todos os serviços pode ser encontrado no *site* do OMG, no endereço [www.omg.org]. Muitos dos serviços CORBA estão descritos em Orfali *et al.* [1996, 1997]. Um resumo dos principais serviços CORBA aparece na Figura 8.6. Mais detalhes sobre esses serviços podem ser encontrados no *site* que acompanha o livro [www.cdk5.net/corba] (em inglês).

<i>Serviço CORBA</i>	<i>Função</i>	<i>Mais detalhes</i>
<i>Serviço de nomes</i>	Oferece atribuição de nomes no CORBA, em particular, realiza o mapeamento de nomes em referências de objeto remoto dentro de determinado contexto de atribuição de nomes (veja o Capítulo 9).	[OMG 2004b]
<i>Serviço de negócio</i>	Enquanto o serviço de nomes permite localizar objetos pelo nome, o <i>Trading Service</i> (serviço de negócio) permite localizá-los pelo atributo; isto é, ele é um serviço de diretório. O banco de dados subjacente gerencia um mapeamento dos tipos de serviço e seus atributos associados para referências de objeto remoto.	[OMG 2000a, Henning e Vinoski 1999]
<i>Serviço de evento</i>	Permite que objetos de interesse transmitam notificações para os assinantes, usando invocações de método remoto CORBA (veja o Capítulo 6 para mais sobre serviços de evento em geral).	[Farley 1998, OMG 2004c]
<i>Serviço de notificação</i>	Amplia o serviço de evento, com os recursos adicionais incluindo a capacidade de definir filtros expressando eventos de interesse e de definir propriedades de confiabilidade e ordenação do canal de eventos subjacente.	[OMG 2004d]
<i>Serviço de segurança</i>	Superta diversos mecanismos de segurança, incluindo autenticação, controle de acesso, comunicação segura, auditoria e não repúdio (veja o Capítulo 11).	[Blakely 1999, Baker 1997, OMG 2002b]
<i>Serviço de transação</i>	Suporta a criação de transações planas ou aninhadas (conforme definidas nos Capítulos 16 e 17).	[OMG 2003]
<i>Serviço de controle de concorrência</i>	Usa travas para aplicar controle de concorrência no acesso de objetos CORBA (pode ser usado por meio do serviço de transação ou como um serviço independente).	[OMG 2000b]
<i>Serviço de estado persistente</i>	Oferece um repositório de objetos persistentes para o CORBA, usado para salvar e restaurar o estado de objetos CORBA (as implementações são recuperadas a partir do repositório de implementações).	[OMG 2002d]
<i>Serviço de ciclo de vida</i>	Define convenções para criar, excluir, copiar e mover objetos CORBA; por exemplo, como usar fábricas para criar objetos.	[OMG 2002e]

Figura 8.6 Serviços CORBA.

```
public interface ShapeListOperations {
    Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException;
    Shape[ ] allShapes( );
    int getVersion( );
}

public interface ShapeList extends ShapeListOperations, org.omg.CORBA.Object,
org.omg.CORBA.portable.IDLEntity {}
```

Figura 8.7 Interfaces Java geradas pelo *idlj* a partir da interface CORBA *ShapeList*.

8.3.5 Exemplo de cliente e servidor CORBA

Esta seção descreve, em linhas gerais, os passos necessários para produzir programas cliente e servidor que usam as interfaces IDL *Shape* e *ShapeList* mostradas na Figura 8.2. A seguir, apresentaremos uma discussão sobre *callbacks* no CORBA. Usamos Java como linguagem de programação tanto para o cliente quanto para o servidor, mas a estratégia é semelhante para outras linguagens. O compilador de interface *idlj* pode ser aplicado nas interfaces CORBA para gerar os seguintes itens:

- As interfaces Java equivalentes – duas por interface IDL. O nome da primeira interface Java termina com *Operations* – essa interface define apenas as operações na interface IDL. A segunda interface Java tem o mesmo nome da interface IDL e implementa as operações da primeira interface, assim como as de uma interface conveniente para um objeto CORBA. Por exemplo, a interface IDL *ShapeList* resulta em duas interfaces Java *ShapeListOperations* e *ShapeList*, como mostrado na Figura 8.7.
- Os esqueletos de servidor para cada interface *idl*. Os nomes das classes de esqueleto terminam com *POA* – por exemplo, *ShapeListPOA*.
- As classes *proxy* ou *stubs* de cliente, uma para cada interface IDL. Os nomes dessas classes terminam com *Stub* – por exemplo, *_ShapeListStub*.
- Uma classe Java para corresponder a cada uma das *structs* definidas com as interfaces IDL. Em nosso exemplo, são geradas as classes *Rectangle* e *GraphicalObject*. Cada uma dessas classes contém uma declaração de uma variável de instância para cada campo na *struct* correspondente e dois construtores, mas nenhum outro método.
- Classes chamadas auxiliares (*helpers*) e portadoras (*holders*), uma para cada um dos tipos definidos na interface IDL. Uma classe auxiliar contém o método *narrow*, que é usado para converter determinada referência de objeto para a classe a que pertence e que está mais abaixo na hierarquia de classes. Por exemplo, o método *narrow* em *ShapeHelper* é convertido para a classe *Shape*. As classes portadoras lidam com argumentos *out* e *inout*, os quais não podem ser mapeados diretamente para Java. Veja no Exercício 8.9 um exemplo do uso de classes portadoras.

Programa servidor • O programa servidor deve conter implementações de uma ou mais interfaces IDL. Para um servidor escrito em uma linguagem orientada a objetos, como Java ou C++, essas interfaces são implementadas como classes serventes. Os objetos CORBA são instâncias de classes serventes.

```

import org.omg.CORBA.*;
import org.omg.PortableServer.POA;
class ShapeListServant extends ShapeListPOA {
    private POA theRootpoa;
    private Shape theList[];
    private int version;
    private static int n=0;
    public ShapeListServant(POA rootpoa){           1
        theRootpoa = rootpoa;
        // inicializa as outras variáveis de instância
    }
    public Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException {      2
        version++;
        Shape s = null;
        ShapeServant shapeRef = new ShapeServant( g, version);
        try{
            org.omg.CORBA.Object ref = theRootpoa.servant_to_reference(shapeRef);
            s = ShapeHelper.narrow(ref);
        } catch (Exception e) {
            if(n >=100) throw new ShapeListPackage.FullException();
            theList[n++] = s;
        }
        return s;
    }
    public Shape[ ] allShapes( ){ ... }
    public int getVersion( ){ ... }
}

```

Figura 8.8 Classe *ShapeListServant* do programa servidor em Java da interface CORBA *ShapeList*.

Quando um servidor cria uma instância de uma classe servente, ele deve registrá-la no POA, o qual transforma a instância em um objeto CORBA e fornece a ele uma referência de objeto remoto. Se isso não for feito, o objeto CORBA não poderá receber invocações remotas. Os leitores que estudaram cuidadosamente o Capítulo 5 poderão perceber que o registro do objeto no POA o faz ser registrado no equivalente CORBA da tabela de objetos remotos.

Em nosso exemplo, o servidor contém implementações das interfaces *Shape* e *ShapeList* na forma de duas classes serventes, junto a uma classe servidora que contém uma seção *initialization* (veja a Seção 5.4.2) em seu método *main*.

As *classes serventes*: cada classe servente estende a classe esqueleto correspondente e implementa os métodos de uma interface IDL usando as assinaturas de método definidas na interface Java equivalente. A classe servente que implementa a interface *ShapeList* é chamada de *ShapeListServant*, embora qualquer outro nome pudesse ser escolhido. Seu esboço aparece na Figura 8.8. Considere o método *newShape*, na linha 1, que é um método de fábrica, pois cria objetos *Shape*. Para transformar um objeto *Shape* em um objeto CORBA, ele é registrado no POA por intermédio de seu método *servant_to_reference*, como mostrado na linha 2, que faz uso da referência ao POA raiz, que foi passado por meio do construtor quando a classe servente foi criada. Versões completas da interface IDL e das classes cliente e servidor deste exemplo estão disponíveis na página do livro em www.grupoa.com.br ou em www.cdk5.net/corba (em inglês).

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
public class ShapeListServer {
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null);
            POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            rootpoa.the_POAManager().activate();
            ShapeListServant SLSRef = new ShapeListServant(rootpoa);
            org.omg.CORBA.Object ref = root.poa.servant_to_reference(SLSRef);
            ShapeList SLRef = ShapeListHelper.narrow(ref);
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ShapeList", "");
            NameComponent path[] = {nc};
            ncRef.rebind(path, SLRef);
            orb.run();
        } catch (Exception e) { ... }
    }
}
```

Figura 8.9 Classe Java *ShapeListServer*.

O servidor: o método *main* na classe servidora *ShapeListServer* aparece na Figura 8.9. Primeiramente, ele cria e inicializa o ORB (linha 1) e, então, obtém uma referência para o POA raiz e ativa o POAManager (linhas 2 e 3). Em seguida, ele cria uma instância de *ShapeListServant*, que é apenas um objeto Java (linha 4) e, ao fazer isso, passa uma referência para o POA raiz. Então, ele o transforma em um objeto CORBA, registrando-o no POA (linha 5). Depois disso, ele registra o servidor no serviço de nomes. Então, ele espera receber requisições de clientes (linha 10).

Um servidor que usa o serviço de nomes, primeiro obtém um contexto de atribuição de nomes raiz usando *NamingContextHelper* (linha 6). O contexto de atribuição de nomes define o escopo dentro do qual um conjunto de nomes se aplica (cada um dos nomes deve ser exclusivo dentro de um contexto). Depois, o servidor faz um *NameComponent* (linha 7), sendo *NameComponent* um objeto que representa um nome no CORBA. Isso tem duas partes: um *nome* e um *tipo*. O campo de *tipo* é puramente descritivo (o campo é usado pelas aplicações e não é interpretado pelo serviço de nomes). Os nomes podem ser compostos e representados por um *caminho* para o objeto no grafo de nomes. Neste exemplo, não são usados nomes compostos; em vez disso, o caminho consiste em um único nome, definido na linha 8. Finalmente, o servidor usa o método *rebind* do serviço de nomes (linha 9), o qual registra o par nome e referência de objeto remoto no contexto apropriado. Os clientes executam os mesmos passos, mas utilizam o método *resolve*, como mostrado na Figura 8.10, linha 2.

O programa cliente • Um exemplo de programa cliente aparece na Figura 8.10. Ele cria e inicializa um ORB (linha 1) e, depois, entra em contato com o serviço de nomes para

```

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class ShapeListClient{
    public static void main(String args[ ]) {
        try{
            ORB orb = ORB.init(args, null);
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ShapeList", "");
            NameComponent path [] = { nc };
            ShapeList shapeListRef =
            ShapeListHelper.narrow(ncRef.resolve(path));
            1
            Shape[ ] sList = shapeListRef.allShapes();
            2
            GraphicalObject g = sList[0].getAllState();
            3
            GraphicalObject path[] = { g };
            4
            } catch(org.omg.CORBA.SystemException e) {...}
            5
        }
    }
}

```

Figura 8.10 Programa cliente em Java para as interfaces CORBA *Shape* e *ShapeList*.

obter uma referência para o objeto remoto *ShapeList*, usando seu método *resolve* (linha 2). Depois disso, ele invoca seu método *allShapes* (linha 3) para obter uma sequência de referências de objeto remoto para todos os objetos *Shape* correntemente mantidos no servidor. Em seguida, ele invoca o método *getAllState* (linha 4), fornecendo como argumento a primeira referência de objeto remoto da sequência retornada; o resultado é fornecido como uma instância da classe *GraphicalObject*.

O método *getAllState* parece contradizer nossa afirmação anterior de que objetos não podem ser passados por valor no CORBA, pois tanto o cliente como o servidor transacionam em instâncias da classe *GraphicalObject*. Entretanto, não há contradição: o objeto CORBA retorna uma *struct*, e os clientes que estiverem usando uma linguagem diferente poderão vê-la de forma diferente. Por exemplo, na linguagem C++, o cliente a veria como uma *struct*. Mesmo em Java, a classe *GraphicalObject* gerada é mais parecida com uma *struct*, pois não tem métodos.

Os programas clientes sempre devem capturar exceções CORBA *SystemExceptions*, as quais relatam erros decorrentes da distribuição (veja a linha 5). Os programas clientes também devem capturar as exceções definidas na interface IDL, como a *FullException* disparada pelo método *newShape*.

Esse exemplo ilustra o uso da operação *narrow*: a operação *resolve* do serviço de nomes retorna um valor de tipo *Object*; esse tipo é reduzido para estar de acordo com o tipo em particular exigido – (*ShapeList*).

Callbacks • *Callbacks* podem ser implementadas no CORBA de uma maneira semelhante àquela descrita para RMI Java, na Seção 5.5.1. Por exemplo, a interface *WhiteboardCallback* pode ser definida como segue:

```

interface WhiteboardCallback {
    oneway void callback(in int version);
};

```

Essa interface é implementada como um objeto CORBA pelo cliente, permitindo que o servidor envie ao cliente um número de versão quando novos objetos são adicionados. No entanto, antes que o servidor possa fazer isso, o cliente precisa informá-lo da referência de objeto remoto de seu objeto. Para tornar isso possível, a interface *ShapeList* exige métodos adicionais, como *register* e *deregister*, conforme segue:

```
int register(in WhiteboardCallback callback);
void deregister(in int callbackId);
```

Após um cliente ter obtido uma referência para o objeto *ShapeList* e criado uma instância de *WhiteboardCallback*, ele usa o método *register* de *ShapeList* para informar o servidor de que está interessado em receber *callbacks*. O objeto *ShapeList* no servidor é responsável por manter uma lista de clientes interessados e notificar a todos eles sempre que seu número de versão aumentar, quando um novo objeto for adicionado. O método *callback* é declarado como *oneway* para que o servidor possa usar chamadas assíncronas para evitar atraso ao notificar cada cliente.

8.4 De objetos a componentes

O *middleware* de objeto distribuído tem sido expressivamente implementado em uma grande variedade de aplicações, incluindo as áreas apresentadas no Capítulo 1: finanças e comércio, assistência médica, educação, transporte e logística e assim por diante. As técnicas incorporadas no CORBA e as plataformas relacionadas se mostraram bem-sucedidas em enfrentar muitos dos principais problemas associados à programação distribuída, especialmente os relacionados a solucionar a heterogeneidade e a permitir a portabilidade e a operação conjunta de *software* de sistemas distribuídos. A variedade de serviços que acompanha essas plataformas também estimula o desenvolvimento de *software* seguro e confiável.

No entanto, foram identificadas várias deficiências. Isso levou ao surgimento do que definimos como estratégias baseadas em componentes, como uma evolução natural da computação de objeto distribuído. Esta seção apresenta essa transição, discutindo os requisitos que levaram às estratégias baseadas em componentes e fornece uma definição de componentes, antes de examinar com mais profundidade os estilos de estratégias baseadas em componentes adotadas nos sistemas distribuídos. A seguir, a Seção 8.5, apresentará dois estudos de caso de tecnologias de componente contrastantes: Enterprise JavaBeans e Fractal.

Problemas do middleware orientado a objetos • Conforme mencionado anteriormente, as estratégias baseadas em componentes surgiram para enfrentar os problemas identificados na computação de objeto distribuído. Os problemas foram listados na Seção 8.1 e serão discutidos com mais detalhes a seguir.

Dependências implícitas: um objeto distribuído oferece um *contrato* para o mundo externo em termos da interface (ou interfaces) que apresenta para o ambiente distribuído. O contrato representa um acordo de compromisso entre o provedor do objeto e os usuários desse objeto, em termos de seu comportamento esperado. Frequentemente, supõe-se que essas interfaces forneçam um contrato completo, no sentido de distribuir e usar esse objeto. Contudo, esse não é o caso. O problema surge do fato de o comportamento interno (encapsulado) de um objeto ser oculto. Por exemplo, um objeto pode se comunicar com outro objeto ou com o serviço de sistema distribuído associado por meio de uma

invocação de método remoto ou de outro paradigma de comunicação. Se olharmos os programas servidor e cliente CORBA mostrados nas Figuras 8.9 e 8.10, respectivamente, veremos que o servidor emite uma *callback* para o cliente e isso não fica aparente na interface definida no servidor. Além disso, embora tanto o cliente como o servidor se comuniquem com o serviço de nomes, novamente isso não é visível a partir da observação externa desse objeto (conforme oferecida pelas interfaces).

Mais geralmente, determinado objeto pode fazer chamadas arbitrárias para outros objetos em nível de aplicação ou para serviços do sistema distribuído que oferecem atribuição de nomes, persistência, controle de concorrência, transações, segurança, etc., e isso não é capturado na visão externa da configuração. As dependências implícitas na configuração distribuída tornam difícil garantir a composição segura de uma configuração, substituir um objeto por outro e a implementação de um elemento em particular em uma configuração distribuída por parte de desenvolvedores externos.

Requisito: a partir disso, há a necessidade clara de não apenas especificar as interfaces oferecidas por um objeto, mas também as dependências desse objeto em relação a outros na configuração distribuída.

Interação com o middleware: apesar dos objetivos da transparência, é claro que, ao usar *middleware* de objeto distribuído, o programador fica exposto a muitos detalhes de nível relativamente baixo, associados à arquitetura do *middleware*. Novamente, o exemplo de cliente-servidor mostrado nas Figuras 8.9 e 8.10 fornecem uma ilustração disso. Apesar de ser uma aplicação muito simples, existem muitas chamadas relacionadas ao CORBA que são absolutamente fundamentais para o funcionamento da aplicação. Isso inclui chamadas associadas aos nomes (conforme mencionado anteriormente), ao POA e ao núcleo ORB. Em exemplos mais complexos, isso poderia incluir código arbitrariamente sofisticado em termos da criação e do gerenciamento de referências de objeto, gerenciamento de ciclos de vida de objetos, políticas de ativação e passivação, gerenciamento de estado persistente e políticas para mapeamentos nos recursos da plataforma subjacente, como as *threads*. Tudo isso pode se tornar, muito rapidamente, um desvio do principal objetivo do código, que é criar um aplicativo. Isso fica muito evidente no exemplo citado anteriormente, em que o código realmente relacionado ao aplicativo de quadro negro (*whiteboard*) é mínimo e intercalado com código relacionado às preocupações com os sistemas distribuídos.

Requisito: há uma necessidade evidente de simplificar a programação de aplicações distribuídas para apresentar uma clara separação de preocupações entre código relacionado à operação em uma estrutura de *middleware* e código associado à aplicação, e de permitir que o programador se concentre exclusivamente neste último.

Falta de separação de aspectos relacionados à distribuição: os programadores que usam *middleware* de objeto distribuído também precisam lidar explicitamente com preocupações não funcionais relacionadas a problemas como segurança, transações, coordenação e replicação. Em tecnologias como CORBA e RMI, isso é conseguido pela inserção de chamadas apropriadas nos serviços do sistema distribuído associados dentro dos objetos. Isso tem duas repercussões:

- Os programadores precisam ter profundo conhecimento dos detalhes completos de todos os serviços do sistema distribuído associados,
- A implementação de determinado objeto conterá código de aplicação junto a chamadas para serviços do sistema distribuído e para as interfaces do *middleware* sub-

jacente (conforme mencionado anteriormente). O emaranhado de possibilidades aumenta ainda mais a complexidade da programação de sistemas distribuídos.

Requisito: a separação de aspectos mencionada anteriormente também deve abranger o tratamento dos diversos serviços do sistema distribuído e, quando possível, as complexidades do tratamento de tais serviços devem ficar ocultas do programador.

Falta de suporte para distribuição: embora tecnologias como CORBA e RMI Java tornem possível desenvolver configurações distribuídas arbitrárias de objetos, não há suporte para a distribuição dessas configurações. Em vez disso, os objetos precisam ser distribuídos manualmente em máquinas individuais. Esse processo pode ser cansativo e propenso a erros, particularmente em distribuições em larga escala, que consistem em muitos objetos, em uma arquitetura física com um grande número de nós computacionais (potencialmente heterogêneos). Além de ser necessário posicioná-los fisicamente, os objetos também precisam ser ativados, e vínculos apropriados para outros objetos precisam ser criados. Por causa dessa falta de suporte para distribuição, inevitavelmente os desenvolvedores contam com estratégias de distribuição *ad hoc*, as quais não são portáveis para outros ambientes.

Requisito: as plataformas de *middleware* devem fornecer suporte intrínseco para a distribuição, para que o *software* distribuído possa ser instalado e implantado da mesma maneira que um *software* para uma única máquina, com as complexidades da distribuição oculta do usuário.

Esses quatro requisitos levaram ao surgimento de estratégias baseadas em componentes para o desenvolvimento de sistemas distribuídos, junto ao surgimento de *middleware baseado em componentes*, incluindo o estilo de *middleware* chamado de *servidores de aplicação*.

Note que, embora as estratégias baseadas em componentes só tenham ganhado importância nos últimos anos, suas raízes remontam aos primeiros projetos que tratavam de reconfiguração em sistemas distribuídos (como o Projeto Conic do Imperial College London [Magee e Sloman 1989]).

Essência dos componentes • Para os propósitos desta discussão, adotamos a definição de componentes fornecida por Szyperski em seu livro sobre *software* de componentes [Szyperski 2002]:

Componentes: um componente de *software* é uma unidade de composição com interfaces especificadas por contratos e somente dependências contextuais explícitas.

Nessa definição clássica, a palavra “somente” se refere ao fato de que quaisquer dependências contextuais devem ser explícitas – isto é, não estão presentes quaisquer dependências implícitas.

Os componentes de *software* são como os objetos distribuídos, no sentido de serem unidades de composição encapsuladas, mas determinado componente especifica suas interfaces fornecidas para o mundo externo e suas dependências de outros componentes no ambiente distribuído. As dependências também são representadas como interfaces. Mais especificamente, um componente é especificado em termos de um *contrato*, o qual inclui:

- um conjunto de *interfaces fornecidas* – isto é, as interfaces que o componente oferece como serviços para outros componentes;

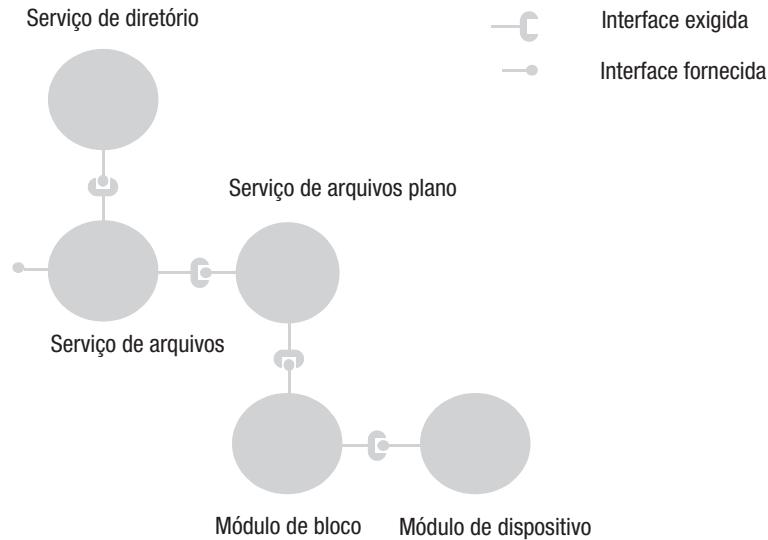


Figura 8.11 Um exemplo de arquitetura de software.

- um conjunto de *interfaces exigidas* – isto é, as dependências que esse componente tem em termos de outros componentes que devem estar presentes e conectados a ele para que ele funcione corretamente.

Em determinada configuração de componente, toda interface exigida deve estar ligada a uma interface fornecida de outro componente. Isso também é chamado de *arquitetura de software*, consistindo em componentes, interfaces e conexões entre interfaces. Podemos ver um exemplo dessa configuração na Figura 8.11. Esse exemplo mostra a arquitetura de um sistema de arquivos simples fornecendo uma interface para outros usuários e, por sua vez, exigindo conexão com um componente de serviço de diretório e um componente de serviço de arquivo plano (*flat file service*). A figura também mostra conexões adicionais para módulos de bloco e de dispositivo, capturando a arquitetura global desse sistema de arquivos em particular. (Investigaremos as arquiteturas reais de sistemas de arquivos distribuídos no Capítulo 12.)

As interfaces podem ter diferentes estilos. Em particular, muitas estratégias baseadas em componentes oferecem dois estilos de interfaces:

- interfaces que suportam invocação de método remoto, como em CORBA e RMI Java;
- interfaces que suportam eventos distribuídos (conforme discutido no Capítulo 6).

Veremos exemplos dos dois estilos de interface quando estudarmos o Enterprise JavaBeans, na Seção 8.5.1.

A programação em sistemas baseados em componentes se preocupa com o desenvolvimento de componentes e com sua *composição*. O objetivo é suportar um estilo de desenvolvimento de *software* que se compare ao desenvolvimento de *hardware* no uso de componentes padrão e compô-los para desenvolver serviços mais sofisticados: uma mudança de desenvolvimento de *software* para montagem de *software*. Portanto,

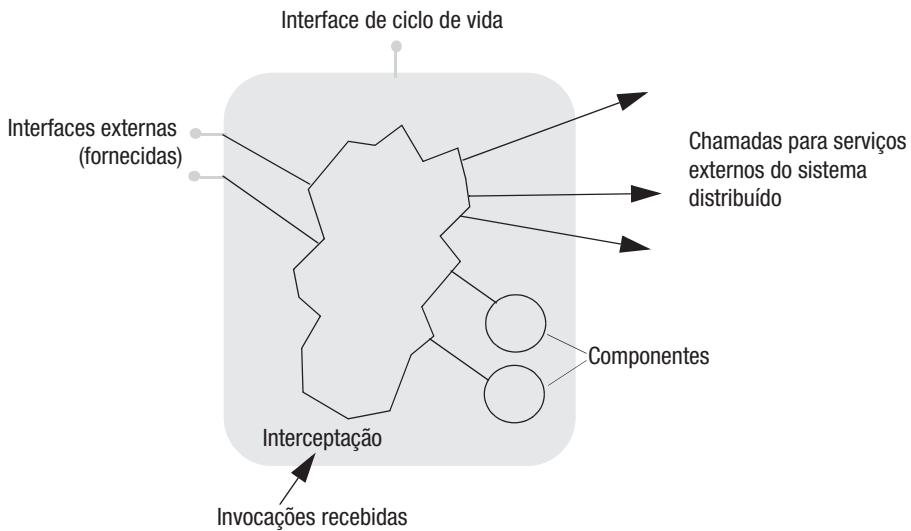


Figura 8.12 A estrutura de um contêiner.

essa estratégia suporta desenvolvimento de componentes de *software* terceirizado e torna mais fácil adaptar configurações de sistema em tempo de execução, substituindo um componente por outro precisamente correspondente em termos das interfaces fornecidas e exigidas.

Note que os defensores das estratégias baseadas em componentes dão ênfase significativa a esse uso de composição e veem isso como a estratégia mais limpa para a construção de sistemas de *software* complexos. Em particular, eles defendem a composição em detrimento da herança, vendo a herança como a criação de formas adicionais de dependência implícita (desta vez, entre classes). Isso pode levar a questões como o problema da classe base frágil, em que alterações em uma classe base podem ter repercuções imprevistas nos objetos que estão mais abaixo na hierarquia de herança [Szyperski 2002].

Até aqui, tratamos do primeiro requisito destacado anteriormente (em termos de tornar todas as dependências explícitas), mas não dos três seguintes, que se referem à simplificação do desenvolvimento e da implementação de aplicações distribuídas. Esse nível adicional de suporte vai se tornar aparente quando examinarmos como as estratégias baseadas em componentes têm evoluído na comunidade de sistemas distribuídos.

Componentes e sistemas distribuídos • Diversas tecnologias de *middleware* baseado em componentes vêm surgindo, incluindo o Enterprise JavaBeans (discutido na Seção 8.5.1, a seguir) e o CCM (CORBA Component Model – modelo de componente CORBA) [Wang *et al.* 2001], uma evolução do CORBA de uma plataforma baseada em objetos para uma plataforma baseada em componentes. O *middleware* baseado em componentes complementa a filosofia adotada anteriormente, mas adiciona suporte significativo para o desenvolvimento e a implementação de sistemas distribuídos, conforme discutido a seguir.

<i>Tecnologia</i>	<i>Desenvolvida por</i>	<i>Mais detalhes</i>
<i>WebSphere Application Server</i>	IBM	[www.ibm.com]
<i>Enterprise JavaBeans</i>	SUN	[java.sun.com XII]
<i>Spring Framework</i>	SpringSource (uma divisão da VMware)	[www.springsource.org]
<i>JBoss</i>	JBoss Community	[www.jboss.org]
<i>CORBA Component Model</i>	OMG	[Wang <i>et al.</i> 2001]
<i>JOnAS</i>	OW2 Consortium	[jonas.ow2.org]
<i>GlassFish</i>	SUN	[glassfish.dev.java.net]

Figura 8.13 Servidores de aplicação.

Contêineres: a noção de *contêineres* é absolutamente fundamental para o *middleware* baseado em componentes. Os contêineres suportam um padrão comum, frequentemente encontrado em aplicações distribuídas, que consiste em:

- um cliente de *front-end* (talvez baseado na Web);
- um contêiner contendo um ou mais componentes que implementam a lógica da aplicação ou do negócio;
- serviços de sistema que gerenciam os dados associados no armazenamento persistente.

(Isso é análogo ao modelo de três camadas físicas, descrito na Seção 2.3.2.)

As tarefas do contêiner são: fornecer um ambiente hospedeiro no lado do servidor *gerenciado* para os componentes e fornecer a separação de aspectos, mencionada anteriormente, na qual os componentes tratam das necessidades da aplicação e o contêiner trata de problemas dos sistemas distribuídos e do *middleware*, garantindo a obtenção de propriedades não funcionais. A estrutura global de um contêiner está apresentada na Figura 8.12. Ela mostra vários componentes encapsulados dentro de um contêiner; o contêiner não fornece acesso direto aos componentes, mas intercepta as invocações recebidas e, então, executa as ações apropriadas para garantir que as propriedades desejadas da aplicação distribuída sejam mantidas. Se pegarmos o caso do CORBA, por exemplo, isso incluiria:

- gerenciar a interação com o núcleo ORB e a funcionalidade do POA subjacentes e ocultá-las totalmente dos desenvolvedores de aplicações;
- gerenciar as chamadas para os serviços do sistema distribuído apropriados, inclusive serviços de segurança e transação, para fornecer as propriedades não funcionais exigidas da aplicação – novamente, de forma transparente para o programador.

Considerados em conjunto, isso pode simplificar significativamente o desenvolvimento de aplicações distribuídas, permitindo que o desenvolvedor de componente se concentre exclusivamente nas preocupações em nível de aplicação. Por exemplo, com uma estratégia de contêiner, todas as chamadas relacionadas ao POA apresentadas nas Figuras 8.8 e 8.9 seriam feitas pelo contêiner e não pelo componente. Analogamente, por intermédio do mecanismo de interceptação, o contêiner pode fazer uma sequência de chamadas potencialmente complexas para serviços apropriados do sistema distribuído, para implementar as propriedades não funcionais exigidas. Como ilustração deste último

ponto, considere a implementação de uma política de gerenciamento simples para tratar do acesso concorrente a um componente. Isso pode ser implementado de maneira transparente para o componente, interceptando-se a invocação recebida na interface externa, adquirindo-se uma trava associada ao componente subjacente e, então, fazendo-se a chamada na operação, no próprio componente, garantindo que a trava seja liberada quando a invocação terminar (vamos falar mais sobre travas na Seção 16.4, mas, por enquanto, um entendimento geral é suficiente).

O *middleware* que suporta o padrão contêiner e a separação de aspectos acarretada por esse padrão é conhecido como *servidor de aplicação*. Esse estilo de programação distribuída está sendo amplamente utilizado no setor. Atualmente, existem diversos servidores de aplicativo; um resumo das principais estratégias está na Figura 8.13. A Seção 8.5.1 examinará a especificação Enterprise JavaBeans como um exemplo de servidor de aplicação.

Suporte para distribuição: o *middleware* baseado em componentes fornece suporte para a distribuição de *configurações* de componente; as versões de *software* são empacotadas como arquiteturas de *software* (componentes e suas interconexões), junto a *descritores de distribuição* que descrevem completamente como a configuração deve ser implantada em um ambiente distribuído.

Note que os componentes são distribuídos em contêineres, sendo que os descritores de distribuição são interpretados pelos contêineres para estabelecer as políticas exigidas pelo *middleware* subjacente e pelos serviços do sistema distribuído. Portanto, determinando contêiner inclui vários componentes que exigem a mesma configuração em termos de suporte para sistema distribuído.

Normalmente, os descritores de distribuição são escritos em XML e incluem informações suficientes para garantir que:

- os componentes sejam corretamente conectados, usando os protocolos apropriados e o suporte de *middleware* associado;
- o *middleware* e a plataforma subjacentes sejam configurados de forma a fornecer o nível correto de suporte para a configuração de componente (por exemplo, no CORBA, isso incluiria configurar o POA);
- os serviços do sistema distribuído associado sejam configurados de forma a fornecer o nível de segurança correto, suporte para transações, etc.

Também são fornecidas ferramentas para interpretar os descritores de distribuição e para garantir a correta implantação de determinada arquitetura física.

8.5 Estudos de caso: Enterprise JavaBeans e Fractal

A vantagem dos servidores de aplicativo é que eles fornecem amplo suporte para um estilo de programação distribuída – a estratégia de três camadas físicas explicada anteriormente – e a maior parte das complexidades associadas à programação distribuída fica oculta para o usuário. As desvantagens são que a estratégia é prescritiva e pesada – prescritiva no sentido de que a estratégia impõe esse estilo de arquitetura de sistemas em particular, e pesada porque os servidores de aplicativo são sistemas de *software* grandes e complexos, que inevitavelmente apresentam uma sobrecarga em termos de requisitos de desempenho e recursos. A estratégia funciona melhor em máquinas servidoras de alta capacidade.

Para contra-atacar isso, um estilo de programação com componentes mais despojado e mínimo também é adotado nos sistemas distribuídos. Referimos-nos a esse estilo como *modelos de componente leves*, para distingui-los das arquiteturas de servidor de aplicação, muito mais pesadas. Nesta seção, apresentaremos dois estudos de caso de tecnologias de componente: Enterprise JavaBeans, um exemplo importante da estratégia de servidor de aplicação, e Fractal, um exemplo de arquitetura de componente leve.

8.5.1 Enterprise JavaBeans

Enterprise JavaBeans (EJB) [[java.sun.com XII](http://java.sun.com/XII)] é uma especificação de arquitetura de componente gerenciada no lado do servidor e um elemento importante da plataforma Java, Enterprise Edition (Java EE), um conjunto de especificações para programação cliente-servidor. Outras especificações incluem RMI Java e JMS, apresentadas em outras partes, do livro (nos Capítulos 5 e 6, respectivamente).

O EJB é definido como *modelo de componente no lado do servidor*, pois suporta o desenvolvimento do estilo clássico de aplicações, em que números potencialmente grandes de clientes interagem com diversos serviços construídos por meio de componentes ou de configuração de componentes. Os componentes, conhecidos como *beans* no EJB, destinam-se a capturar a lógica da aplicação (ou do negócio), conforme definida no Capítulo 2, com o EJB também suportando a separação entre essa lógica da aplicação e seu armazenamento persistente em um banco de dados de *back-end*. Em outras palavras, o EJB fornece suporte direto para a arquitetura de três camadas físicas apresentada na Seção 2.3.2.

O EJB é *gerenciado*, no sentido de que o padrão contêiner, apresentado anteriormente (Seção 8.4), é usado para fornecer suporte para importantes serviços de sistemas distribuídos, incluindo transações, segurança e ciclo de vida. Normalmente, o contêiner injeta chamadas apropriadas para os serviços associados, a fim de fornecer as propriedades exigidas, sendo que o uso de um gerenciador de transação ou de serviços de segurança fica completamente oculto do desenvolvedor dos *beans* associados (*gerenciados pelo contêiner*). Também é possível ao desenvolvedor de *beans* ter mais controle sobre essas operações (*gerenciadas por beans*).

O objetivo do EJB é manter uma forte separação entre as várias funções envolvidas no desenvolvimento de aplicações distribuídas. A especificação EJB identifica as seguintes funções principais:

- o *provedor de bean*, que desenvolve a lógica da aplicação do componente (ou componentes);
- o *montador de aplicação*, que monta os *beans* nas configurações de aplicação;
- o *distribuidor*, que pega determinada montagem de aplicação e garante que ela seja corretamente distribuída em determinado ambiente operacional;
- o *provedor de serviço*, que é um especialista em serviços de sistema distribuído fundamentais, como gerenciamento de transação, e estabelece o nível de suporte desejado nessas áreas;
- o *provedor de persistência*, que é um especialista em mapear dados persistentes nos bancos de dados subjacentes e em gerenciar essa relação em tempo de execução;
- o *provedor de contêiner*, que complementa as duas funções anteriores e é responsável por configurar os contêineres corretamente, com o nível de suporte para sistemas distribuídos exigido em termos das propriedades não funcionais relacionadas, por exemplo, às transações e à segurança, assim como o suporte para persistência desejado;

- o *administrador de sistema*, que é responsável por monitorar a distribuição em tempo de execução e por fazer ajustes para garantir sua operação correta.

Note que o EJB é uma arquitetura de componente *pesada*, no sentido apresentado anteriormente. Há complexidade de *software* significativa, associada particularmente ao gerenciamento de contêineres. Assim, a estratégia é prescritiva e destinada apenas a certos tipos de aplicação. Conforme mencionado anteriormente, o EJB é particularmente adequado a aplicações que seguem a arquitetura de três camadas físicas, baseada em um banco de dados de *back-end* acessado por meio de uma interface de serviço oferecida pela camada física do meio (a lógica da aplicação). Por exemplo, esse estilo de arquitetura é comum em muitas aplicações de comércio eletrônico (*e-Commerce*), em que o banco de dados mantém informações sobre itens em estoque, preços e disponibilidade, enquanto a camada física do meio oferece interfaces para examinar o estoque e comprar itens selecionados. Normalmente, esses sistemas são grandes e complexos, exigindo suporte em termos de serviços de sistema distribuído; assim, a sobrecarga associada ao gerenciamento do contêiner é plenamente justificada. Como motivação, e para ilustrar o uso do EJB neste cenário, usaremos o exemplo de uma loja eletrônica (*e-Shop*) ao longo de toda esta seção. Outros tipos de aplicação não seguirão esse padrão e, assim, o EJB é uma tecnologia inadequada para tais aplicações. Exemplos incluem estruturas *peer-to-peer*, que simplesmente não seguem esse modelo de camadas físicas, e aplicações mais leves, executadas em dispositivos embarcados, nos quais a sobrecarga do EJB não se justifica.

Nesta seção, abordaremos os recursos do EJB 3.0 [[java.sun.com XII](http://java.sun.com/XII)], lançado em 2006. Um número grande de implementações de EJB 3.0 está disponível, tanto comercialmente como em consórcios de código-fonte aberto. Os principais exemplos incluem Spring, JBoss, JOnAS e Glassfish.

O modelo de componente do EJB • No EJB, um *bean* é um componente que oferece uma ou mais *interfaces do negócio* para clientes em potencial desse componente, em que as interfaces podem ser *remotas*, exigindo o uso de um *middleware* de comunicação apropriado (como RMI ou JMS), ou *local*, no caso em que são possíveis vínculos mais diretos e, assim, mais eficientes. Voltando à terminologia apresentada na Seção 8.4, uma interface do negócio é equivalente a uma *interface fornecida* (vamos ver como o EJB suporta interfaces exigidas a seguir, na subseção sobre injeção de dependência). Determinado *bean* é representado pelo conjunto de interfaces do negócio remotas e locais, junto a uma *classe de bean* associada que implementa as interfaces. Dois estilos principais de *bean* são suportados na especificação EJB 3.0:

Beans de sessão: um *bean* de sessão é um componente que implementa uma tarefa específica dentro da lógica da aplicação de um serviço, por exemplo, fazer uma compra em nossa aplicação *eShop*. Um *bean* de sessão persiste pela duração de um serviço e mantém uma conversa contínua com o cliente enquanto durar a sessão. Os *beans* de sessão podem ser *com estado*, mantendo o estado da conversa associado (como o *status* atual da transação de comércio eletrônico) ou *sem estado*, no caso em que nenhum estado é mantido. Os *beans* de sessão com estado implicam uma conversa com um único cliente e mantêm o estado dessa conversa. Em contraste, os *beans* sem estado podem ter muitas conversas concomitantes com diferentes clientes. O estado associado aos *beans* com estado pode ser persistente ou não, conforme discutiremos a seguir.

Beans baseados em mensagens: os clientes interagem com *beans* de sessão usando invocação local ou remota. Ao longo de todo o livro, temos visto que outros para-

digmas de comunicação também são importantes para o desenvolvimento de sistemas distribuídos, incluindo os paradigmas de comunicação indireta. O conceito de *bean* baseado em mensagens foi introduzido no EJB 2.0 para suportar comunicação indireta e, em particular, a possibilidade de interagir com os componentes usando filas de mensagens ou tópicos, complementando diretamente a funcionalidade oferecida pelo JMS (lembre-se de que filas e tópicos são entidades de primeira classe no JMS, representando intermediários alternativos para mensagens – consulte a Seção 6.4.3). Em um *bean* baseado em mensagens, uma interface do negócio será concretizada como uma interface estilo receptora, refletindo a natureza baseada em eventos do *bean* associado.

POJOs e anotações • A tarefa de programação no EJB foi significativamente simplificada pelo uso de *Enterprise JavaBean POJOs* (*Plain Old Java Objects – objetos Java puros*), junto a anotações *Enterprise JavaBean* da linguagem Java. Um *bean* (que é a implementação das interfaces do negócio do *bean*) é um objeto Java puro: ele consiste na lógica da aplicação escrita simplesmente em Java, sem que nenhum outro código relacionado seja um *bean*. Então, anotações são usadas para garantir o comportamento correto no contexto do EJB. Em outras palavras, um *bean* é um POJO complementado com anotações.

As anotações foram introduzidas no Java 1.5 como um mecanismo para associar metadados a pacotes, classes, métodos, parâmetros e variáveis. Esses metadados podem, então, ser usados por estruturas para garantir que o comportamento ou a interpretação correta seja associada a essa parte do programa. Como exemplo, anotações são usadas para introduzir um *bean* de um estilo específico. A seguir, estão exemplos de definições de *bean* anotados (representando os principais estilos de *bean* no EJB 3.0):

```
@Stateful public class eShop implements Orders {...}
@Stateless public class CalculatorBean implements Calculator {...}
@MessageDriven public class SharePrice implements MessageListener {...}
```

As anotações também são usadas para indicar se as interfaces do negócio são remotas (@*Remote*) ou locais (@*Local*). O exemplo a seguir introduz a interface *Orders* como uma interface remota e a interface *Calculator* de *CalculatorBean* apenas como uma interface local:

```
@Remote public interface Orders {...}
@Local public interface Calculator {...}
```

Conforme ficará claro, as anotações são usadas por todo o EJB, fornecendo uma especificação sobre como um programa deve ser interpretado em um contexto EJB.

Na descrição a seguir, vamos desenvolver o exemplo de loja eletrônica como uma ilustração do uso extensivo de anotações na programação de objetos *bean* (neste caso, um *bean* de sessão).

Contêineres Enterprise JavaBean no EJB • O EJB adota uma estratégia baseada em contêineres, conforme descrito na Seção 8.4. Os *beans* são distribuídos em contêineres, e estes fornecem gerenciamento de sistema distribuído implícito usando interceptação. Assim, o contêiner fornece as políticas necessárias em áreas que incluem gerenciamento de transação, segurança, persistência e gerenciamento de ciclo de vida, permitindo que o desenvolvedor de *bean* se concentre exclusivamente na lógica da aplicação. Portanto, os contêineres devem ser configurados com o nível de suporte necessário. Na versão atual, o EJB é previamente configurado com as políticas padrão comuns, e o desenvolvedor só

precisa tomar medidas se esses padrões forem insuficientes (o que é referido como *configuração por exceção* na especificação [[java.sun.com XII](#)]).

Um número significativo de anotações são definidas para controlar os vários aspectos mencionados anteriormente. Ilustraremos seu uso enfocando o gerenciamento de transações do Enterprise JavaBean e incentivamos o leitor a examinar também as especificações do EJB 3.0 para mais exemplos. As transações serão apresentadas nos Capítulos 16 e 17. Em resumo, contudo, sua função é garantir que todos os objetos (ou, neste contexto, componentes) gerenciados por um único servidor (ou por vários servidores, no caso de transações distribuídas) permaneçam em um estado consistente, apesar do acesso concorrente de vários clientes e no caso de falha do servidor. Elas conseguem fazer isso por permitir que uma sequência de operações seja executada *de forma atômica*, no sentido de que a sequência de operações termine com êxito, de uma maneira que esteja livre da interferência de outro acesso concorrente, ou que a sequência não tenha nenhum efeito na presença de uma falha (como no caso de um colapso do servidor). Voltando ao nosso exemplo de loja eletrônica, um mecanismo de transação garantirá, por exemplo, que duas compras concomitantes não resultem na venda do mesmo item e que um colapso do servidor não permita que o sistema entre em um estado inconsistente, em que um item foi pago, mas não consignado ao comprador.

Os mecanismos para executar transações são relativamente complexos; portanto, dois capítulos são dedicados a esse assunto, posteriormente no livro. Contudo, a ideia geral é relativamente simples, e um entendimento intuitivo será suficiente para se entender como o EJB gerencia transações. O principal ponto a ser lembrado é que as transações se referem a sequências de operações e que as sequências devem ser claramente identificadas para que o serviço de gerenciamento de transação faça seu trabalho. No EJB, as transações se aplicam igualmente a qualquer estilo de *bean*, incluindo *beans* de sessão e *beans* baseados em mensagens.

A primeira coisa a ser declarada é se as transações associadas a um *bean* do negócio devem ser gerenciadas por *bean* ou por contêiner. Isso é conseguido pela associação das anotações a seguir com a classe associada, respectivamente:

```
@TransactionManagement (BEAN)  
@TransactionManagement (CONTAINER)
```

As transações gerenciadas por *beans* são as mais simples de entender. Nesse caso, o desenvolvedor de *bean* é responsável por identificar explicitamente a sequência de operações a ser incluída dentro da transação. Isso é conseguido por se incluir explicitamente dois métodos da interface Java *javax.transaction.UserTransaction* – os métodos *UserTransaction.begin* e *UserTransaction.commit* – dentro do código do *bean*. Isso pode ser usado no lado cliente ou servidor de uma interação. O trecho de código a seguir ilustra o uso dessa estratégia gerenciada por *bean* no exemplo de loja eletrônica:

```
@Stateful  
@TransactionManagement (BEAN)  
public class eShop implements Orders {  
    @Resource javax.transaction.UserTransaction ut;  
  
    public void MakeOrder (...) {  
        ut.begin ();  
        ...  
        ut.commit ();  
    }  
}
```

Atributo	Política
<i>REQUIRED</i>	Se o cliente tem uma transação associada em execução, executa dentro dessa transação; caso contrário, inicia uma nova transação.
<i>REQUIRES_NEW</i>	Sempre inicia uma nova transação para essa invocação.
<i>SUPPORTS</i>	Se o cliente tem uma transação associada, executa o método dentro do contexto dessa transação; se não tiver, a chamada prossegue sem qualquer suporte para transação.
<i>NOT_SUPPORTED</i>	Se o cliente chama o método dentro de uma transação, essa transação é suspensa antes da chamada do método e, posteriormente, retomada – isto é, o método invocado é excluído da transação.
<i>MANDATORY</i>	O método associado deve ser chamado dentro de uma transação de cliente; se não for, uma exceção é disparada.
<i>NEVER</i>	Os métodos associados não devem ser chamados dentro de uma transação de cliente; se isso for tentado, uma exceção será disparada.

Figura 8.14 Atributos de transação no EJB.

Contudo, até certo ponto, isso vai contra o espírito da estratégia de contêiner, pois exige a inclusão de código relacionado à transação dentro do *bean*. A alternativa, transação gerenciada por contêiner, evita a necessidade desse código explícito, permitindo que o contêiner determine quando vai começar e terminar uma transação. Isso é obtido por meio da associação de determinada política de *demarcação* à execução do *bean*. Novamente, isso é conseguido de forma declarada por se associar uma anotação apropriada a determinado método dentro da classe de *bean*. Por exemplo, considere o trecho de código a seguir:

```
@Stateful public class eShop implements Orders {
    ...
    @TransactionAttribute (REQUIRED)
    public void MakeOrder (...) {
        ...
    }
}
```

Isso mostra a associação da política *REQUIRED* ao método *MakeOrder*. Essa política diz que o método associado deve ser executado dentro de uma transação. Para entender essa política, é necessário se dar conta de que uma transação pode ser iniciada pelo chamador ou ser a responsabilidade do próprio *bean*. A política *REQUIRED* inicia uma nova transação, se necessário – isto é, se o chamador não fornecer um contexto de transação indicando que o trabalho já está sendo feito dentro de uma transação. Essa e outras políticas estão resumidas na Figura 8.14.

Note que, por padrão, as transações são gerenciadas por contêiner no EJB.

Injeção de dependência: o exemplo anterior também ilustra mais uma função importante dos contêineres, a *injeção de dependência do Enterprise JavaBean*. A injeção de dependência é um padrão comum na programação, com a qual um agente externo, neste caso um contêiner, é responsável por gerenciar e solucionar as relações entre um componente e suas dependências (as interfaces exigidas, na terminologia da Seção 8.4). Em particular, no EJB 3.0, um componente se refere a uma dependência usando uma anotação, e o

contêiner é responsável por solucionar essa anotação e garantir que, em tempo de execução, o atributo associado se refira ao objeto correto. Normalmente, isso é implementado pelo contêiner usando reflexão.

Por exemplo, no trecho de código anterior, a anotação `@Resource` indica uma dependência desse componente em relação a um objeto que implementa a interface `UserTransaction`. Isso simplesmente deve existir para que a configuração faça sentido. A injeção de dependência tanto sinaliza essa dependência como garante que, quando a configuração de componente correta é implementada, o atributo `ut` se refira ao recurso externo correto.

Interceptação em Enterprise JavaBean • A especificação Enterprise JavaBeans permite ao programador interceptar dois tipos de operação em *beans* para alterar seus comportamentos padrão:

- chamadas de método associadas a uma interface do negócio;
- eventos de ciclo de vida.

Veremos cada um por sua vez, a seguir.

Interceptação de métodos: esse mecanismo é usado quando é necessário associar uma ação em particular ou um conjunto de ações a uma chamada recebida em uma interface do negócio. Isso se aplica igualmente às invocações recebidas em um *bean* de sessão ou a eventos recebidos em um *bean* baseado em mensagens. Conforme já vimos, a interceptação é usada amplamente na arquitetura EJB para fornecer gerenciamento implícito. Isso permite ao desenvolvedor da aplicação estender o uso de interceptação a preocupações mais específicas do domínio, não fornecidas pelo contêiner.

Considere o exemplo da loja eletrônica (*eShop*). Suponha que, dentro da loja eletrônica, haja a necessidade de implementar o registro de todas as operações realizadas no sistema, por exemplo, para propósitos de auditoria. A interceptação permite ao programador introduzir esse serviço sem alterar a lógica da aplicação contida no *bean*. Como um segundo exemplo, o mecanismo de interceptação poderia ser usado para impedir que certos clientes fizessem compras na loja eletrônica (por exemplo, caso estejam em débito com pagamentos anteriores).

Existem várias maneiras de associar interceptadores a determinado *bean*, incluindo associar uma classe de interceptação a determinada classe de *bean* ou método individual (usando a anotação `@Interceptors`), ou associar um método de interceptação a determinada classe (usando a anotação `@AroundInvoke`). Por simplicidade, abordaremos este último mecanismo e voltaremos ao nosso exemplo de loja eletrônica.

```
@Stateful
public class eShop implements Orders {
    public void MakeOrder (...) {
        ...
    }
    @AroundInvoke
    public Object log(InvocationContext ctx) throws Exception {
        System.out.println ("The following method was invoked: " +
                           ctx.getMethod().getName());
        return invocationContext.proceed();
    }
}
```

<i>Assinatura</i>	<i>Uso</i>
<code>public Object getTarget();</code>	Retorna a instância de <i>bean</i> associada à invocação ou ao evento recebido.
<code>public Method getMethod();</code>	Retorna o método que está sendo invocado.
<code>public Object[] getParameters();</code>	Retorna o conjunto de parâmetros associado ao método do negócio interceptado.
<code>public void setParameters(Object[] params);</code>	Permite que o conjunto de parâmetros seja alterado pelo interceptador, supondo que a exatidão de tipo seja mantida.
<code>public Object proceed() throws Exception;</code>	A execução passa para o próximo interceptador do encadeamento (se houver) ou para o método que foi interceptado.

Figura 8.15 Contextos de invocação no EJB.

A anotação `@AroundInvoke` introduz um interceptador na classe de *bean* *eShop*. O método interceptador deve ter a seguinte sintaxe:

Objeto <nomeMétodo>(javax.ejb.ContextoInvocação)

Então, esse método é chamado quando qualquer um dos métodos do negócio é chamado em *eShop*. O parâmetro associado aumenta significativamente os recursos dos interceptadores, fornecendo metadados associados à invocação que está sendo interceptada (por exemplo, referências ao *bean*, ao método invocado e aos parâmetros reais associados à invocação) e também recursos limitados a interceder – isto é, para alterar os parâmetros antes que o método seja executado. A última linha do método, a chamada para *proceed*, retorna o controle para o método interceptado (ou para o próximo interceptador do encadeamento, caso mais de um interceptador esteja definido).

Os principais métodos associados ao contexto de invocação estão resumidos na Figura 8.15.

Interceptação de eventos de ciclo de vida: um mecanismo semelhante pode ser usado para interceptar e reagir a eventos de ciclo de vida associados a um componente. Em particular, a especificação EJB permite a um desenvolvedor de *bean* associar interceptadores à criação e à exclusão de componentes, usando as seguintes anotações, respectivamente:

```
@PostConstruct  
@PreDestroy
```

As anotações são associadas a determinados métodos na classe de *bean*, com o efeito de que esses métodos serão chamados quando o evento de ciclo de vida associado acontecer. Por exemplo, no trecho de código a seguir da classe *eShop*, *TidyUp* será chamado imediatamente antes que o componente seja destruído:

```
@Stateful  
public class eShop implements Orders {  
    ...  
    public void MakeOrder (...) {...}  
    ...  
    @PreDestroy void TidyUp() {...}  
}
```

Essa anotação é geralmente usada para liberar quaisquer recursos correntemente em uso pela classe *eShop*, por exemplo arquivos abertos ou soquetes associados à implementação de *eShop*. Analogamente, *TidyUp* poderia ser usado para garantir que dados importantes associados a *eShop* sejam escritos no banco de dados de *back-end*.

Se o *bean* tem estado, também é possível capturar eventos de ativação e passivação, usando *@PostActivate* e *@PrePassivate*. Novamente, isso permite que ações importantes sejam executadas em associação com esses eventos de ciclo de vida – por exemplo, garantindo que o estado da conversa associado a uma sessão seja armazenado no banco de dados antes que o *bean* seja passivado.

As anotações são usadas ao longo de todo o EJB 3.0 para fornecer um modelo de programação consistente e simples, por meio do qual os desenvolvedores de componente constróem a lógica da aplicação em POJOs e, então, decoram isso, onde for apropriado, com anotações de meta-nível adicionais que são interpretadas pela estrutura de contêiner.

8.5.2 Fractal

Conforme mencionado anteriormente, Fractal é um modelo de componente leve que traz as vantagens da programação baseada em componentes para o desenvolvimento de sistemas distribuídos [Bruneton *et al.* 2006, fractal.ow2.org I]. O Fractal fornece suporte para *programação com interfaces*, com os benefícios associados em termos da separação entre interface e implementação (benefícios também apresentados pelos objetos distribuídos). Contudo, o Fractal vai além e suporta a *representação explícita* da arquitetura de *software* do sistema, evitando o problema das dependências implícitas discutido na Seção 8.4. A estratégia é deliberadamente mínima, sem nenhum suporte para funcionalidade adicional relacionada a componentes, como a distribuição, o padrão contêiner completo ou o modelo de programação enriquecido oferecido pelos servidores de aplicação. O Fractal é usado para a construção de sistemas de *software* mais complexos (incluindo sistemas de *middleware*, conforme discutido a seguir), usando o modelo de componente como bloco de construção básico, resultando em *software* que tem uma arquitetura baseada em componentes clara e que é *configurável* e também *reconfigurável* em tempo de execução, para satisfazer o ambiente operacional e os requisitos atuais.

O Fractal define um modelo de programação e, como tal, é agnóstico quanto a linguagem de programação. Várias implementações desse modelo estão disponíveis em diversas linguagens diferentes, incluindo:

- Julia e AOKEll (baseadas em Java, com a última oferecendo também suporte para programação orientada a aspectos);
- Cecilia e Think (baseadas em C);
- FracNet (baseada em .Net);
- FracTalk (baseada em Smalltalk);
- Julio (baseada em Python).

Julia e Cecilia são tratadas como as implementações de referência do Fractal.

O Fractal é suportado pelo consórcio OW2 [www.ow2.org], uma comunidade de *software* de código-fonte aberto para *middleware* de sistemas distribuídos que estimula e promove a filosofia baseada em componentes para a construção de tal *software*. Até hoje, o Fractal tem sido usado na construção de uma ampla variedade de plataformas de *middleware*, incluindo Think (um núcleo de sistema operacional configurável), DREAM

(uma plataforma de *middleware* que suporta várias formas de comunicação indireta), Jasmine (uma ferramenta que suporta monitoramento e gerenciamento de plataformas SOA), GOTM (que oferece gerenciamento de transação flexível) e Proactive (uma plataforma de *middleware* para computação de Grade – *Grid*). O Fractal também é a base do GCM (Grid Component Model), que tem sido influente no desenvolvimento de padrões ETSI associados [Baude *et al.* 2009]. Mais detalhes sobre todos esses projetos podem ser encontrados no *site* do OW2 [www.ow2.org].

Note que alguns outros modelos de componente leves foram desenvolvidos especificamente para sistemas distribuídos. Apresentaremos dois – OpenCOM e OSGI – no quadro a seguir.

O modelo de componente básico • No Fractal, um componente oferece uma ou mais interfaces, com dois tipos disponíveis:

- *interfaces de servidor*, que suportam as invocações recebidas (equivalentes às interfaces fornecidas na terminologia da Seção 8.4);
- *interfaces de cliente*, que suportam as invocações enviadas (equivalentes às interfaces exigidas).

Uma interface é uma implementação de um *tipo de interface*, o qual define as operações suportadas por essa interface.

Vínculos no Fractal: para permitir a composição, o Fractal suporta *vínculos* entre interfaces. Dois estilos de vínculo são suportados pelo modelo:

Vínculos primitivos: o estilo de vínculo mais simples é o *primitivo*, que é um mapeamento direto entre uma interface de cliente e uma interface de servidor dentro do mesmo espaço de endereçamento, supondo que os tipos sejam compatíveis. Os vínculos primitivos podem ser implementados eficientemente em determinado ambiente de linguagem, por exemplo, por meio de referências de objeto diretas.

Vínculos compostos: o Fractal também suporta *vínculos compostos*, que são arquiteturas de *software* arbitrariamente complexas (isto é, consistindo em componentes e vínculos) que implementam a comunicação entre várias interfaces, potencialmente em máquinas diferentes. Por exemplo, se você estivesse implementando uma conexão CORBA no Fractal, o vínculo seria formado por componentes representando os elementos arquitetônicos básicos no CORBA, incluindo *proxies*, o núcleo ORB, adaptadores de objeto, esqueletos e serventes (espelhando a arquitetura da Figura 8.5).

Os próprios vínculos compostos são componentes no Fractal, e isso é importante por dois motivos:

- Um sistema desenvolvido usando Fractal é totalmente configurável em termos dos componentes e de suas interconexões. Por exemplo, pode ser estabelecida uma configuração na qual os componentes interagem usando um vínculo composto, implementando qualquer um dos paradigmas de comunicação discutidos nos Capítulos 5 e 6 (invocação remota ou indireta, ponto a ponto ou de vários participantes, etc.). Se determinado paradigma de comunicação ainda não é fornecido, ele pode ser desenvolvido no Fractal e, então, disponibilizado como um componente para futuros desenvolvedores.

OpenCOM • OpenCOM [Coulson *et al.* 2008] é um modelo de componente leve com objetivos muito parecidos com os do Fractal. O OpenCOM é um modelo de componente mínimo e aberto, projetado para ser independente de domínio e de ambiente operacional; ou seja, a tecnologia de componentes é suficientemente flexível para ser aplicada em qualquer contexto, incluindo áreas exigentes, como redes de sensor sem fio com recursos limitados. O OpenCOM também é projetado para oferecer sobrecarga insignificante em termos de requisitos de desempenho e de memória, permitindo seu uso em situações em que o desempenho é fundamental – por exemplo, em implementações de roteador.

A arquitetura global do OpenCOM consiste em um núcleo mínimo, suportando operações de componente básicas que incluem carregamento e descarregamento de um componente e o vínculo de componentes. Isso, então, é melhorado com extensões refletivas e de plataforma opcionais, as quais suportam o carregamento dinâmico de recursos refletivos. Além disso, diferentes modelos servem de base para as operações importantes da plataforma, incluindo a semântica de carregamento e vínculo. Portanto, as extensões são conceitualmente semelhantes aos controladores no Fractal.

O OpenCOM tem sido usado no desenvolvimento de uma variedade de plataformas de *middleware* experimentais, incluindo ReMMoC [Grace *et al.* 2003], que oferece descoberta de serviço em ambientes de computação ubíquos altamente heterogêneos (veja o Capítulo 19), e GridKIT [Grace *et al.* 2008], uma estrutura de *middleware* experimental, altamente configurável e reconfigurável para computação em Grade, que também apresenta uma estrutura para a construção de redes virtualizadas arbitrárias, incluindo redes de sobreposição *peer-to-peer* estruturadas e não estruturadas.

OSGi • OSGi [www.osgi.org] é uma especificação de plataforma de *middleware* baseada em Java, gerenciada pela organização de padrões abertos, a OSGi Alliance. Existem várias implementações dessa especificação, incluindo Equinox, Knopflerfish, Felix e Concierge. A plataforma suporta a distribuição e o subsequente gerenciamento de ciclo de vida e adaptação de sistemas de *software* modulares, organizados como *pacotes* de comunicação (semelhantes aos componentes). Os pacotes se comunicam por meio de uma ou mais interfaces de serviço, com os serviços publicados em um registro, suportando, assim, o vínculo dinâmico. Como a unidade do gerenciamento de ciclo de vida, determinado pacote pode ser instalado, iniciado, ativado, parado e desinstalado. Os pacotes também podem ser distribuídos dinamicamente em tempo de execução e os já existentes podem ser atualizados. O OSGi foi desenvolvido originalmente para a programação de *gateways* de serviço (daí o nome original, *Open Service Gateway*), mas agora é usado em uma grande variedade de domínios de aplicação, incluindo a programação de telefones móveis, como *middleware* para computação em Grade e também como arquitetura de *plug-in* no IDE (Integrated Development Environment – ambiente de desenvolvimento integrado) Eclipse, uma estrutura multilínguagem popular para desenvolvimento de *software*.

O OSGi tem como objetivo a distribuição e o gerenciamento de configurações de *software* centralizadas, residindo, por exemplo, em um único dispositivo ou em um servidor. Também foi desenvolvida uma implementação distribuída de OSGi, R-OSGi [Rellermeier *et al.* 2007]. Isso permite que arquiteturas de *software* sejam distribuídas em limites de serviço entre configurações de rede arbitrárias. O R-OSGi usa o padrão *proxy*, apresentado no Capítulo 2, para obter distribuição transparente nesses limites.

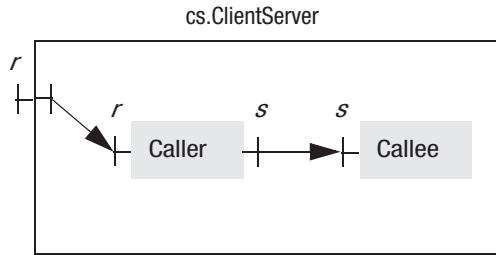


Figura 8.16 Um exemplo de configuração de componente no Fractal.

- Uma vez estabelecido, qualquer aspecto da arquitetura de *software* pode ser reconfigurado em tempo de execução, incluindo os vínculos compostos. A capacidade de adaptar estruturas de comunicação em tempo de execução é muito útil, por exemplo, para introduzir mais níveis de segurança ou alterar a implementação para que seja mais escalável à medida que o tamanho do sistema aumenta.

Veremos mais detalhes sobre o suporte para reconfiguração na seção sobre membranas e controladores, a seguir.

Composição hierárquica: o modelo de componente é hierárquico, pois um componente consiste em uma série de subcomponentes e vínculos associados, em que os próprios subcomponentes podem ser compostos. Por exemplo, o núcleo ORB no exemplo anterior poderia ser decomposto, dada sua inherente complexidade. A composição é suportada pela linguagem Fractal ADL (Architectural Description Language), a qual apresentamos por meio de um exemplo simples, que mostra a criação de um componente contendo dois subcomponentes que interagem como na estratégia cliente-servidor:

```
<definition name="cs.ClientServer">
    <interface name="r" role="server"
        signature="java.lang.Runnable" />
    <component name="caller" definition="hw.CallerImpl" />
    <component name="callee" definition="hw.CalleeImpl" />
    <binding client="this.r" server="caller.r" />
    <binding client="caller.s" server="callee.s" />
</definition>
```

A Fractal ADL é baseada na XML. Esse exemplo mostra um componente *cs.ClientServer* com dois subcomponentes, *caller* e *callee*; vínculos são criados entre a interface de cliente, *this.r* (isto é, a interface *r* definida no componente contêiner *cs.ClientServer*) e a interface associada *caller.r* (a interface *r* definida no componente *caller*), e entre a interface de cliente *caller.s* e a interface de servidor correspondente *callee.s*. A configuração associada está ilustrada na Figura 8.16.

O Fractal também suporta *compartilhamento*, por meio do qual determinado componente pode ser compartilhado entre várias arquiteturas de *software*. Os desenvolvedores do Fractal argumentam que isso é necessário para representar fielmente as arquiteturas de sistema, incluindo o acesso aos recursos subjacentes que são fundamentalmente compartilhados, como uma conexão TCP, por exemplo. Como mais um exemplo, seria possível *callee* (o componente servidor) ser compartilhado entre várias configurações.

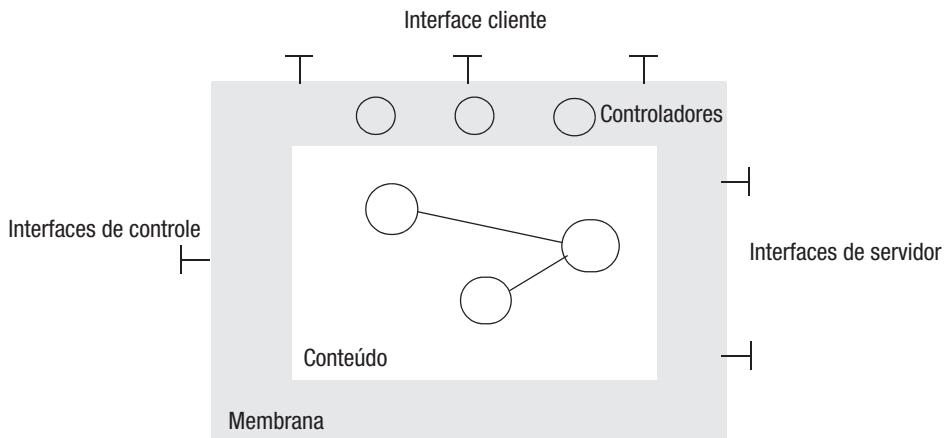


Figura 8.17 A estrutura de um componente do Fractal.

Membranas e controladores • Na implementação, um componente consiste em uma *membrana*, a qual define recursos de controle associados ao componente por meio de um conjunto de *controladores*, e também no *conteúdo* associado – os subcomponentes (e vínculos) que constituem sua arquitetura. As interfaces podem ser *internas* à membrana e, portanto, visíveis apenas para os componentes dentro da parte relativa ao conteúdo, ou *externas* e, portanto, visíveis para quaisquer outros componentes. Essa estrutura está ilustrada na Figura 8.17.

O conceito de membrana é fundamental para a estratégia do Fractal: uma membrana fornece um regime de controle configurável para o conjunto de componentes encapsulados (o conteúdo). Em outras palavras, o conjunto de controladores define os recursos de controle e a semântica associada desses componentes. Mudando o conjunto de controladores, mudamos também os recursos.

Os controladores podem ser usados para vários propósitos:

- Um dos principais usos dos controladores é na implementação de *gerenciamento de ciclo de vida*, incluindo as operações associadas à ativação e à passivação, como *suspend*, *resume* e *checkpoint*. Por exemplo, o Fractal suporta um controlador *LifeCycleController*, que possui três métodos, *startFc*, *stopFc* e *getFcState*, que implementam essas três funções, respectivamente. Isso é fundamental nos casos em que as reconfigurações da arquitetura de *software* subjacente estão sendo feitas em tempo de execução. Considere o exemplo simples da configuração cliente-servidor anterior e suponha que o servidor seja substituído dinamicamente por outro, aprimorado (talvez um que suporte *multithreading* para obter melhor desempenho de saída). Neste caso, para evitar inconsistências, é interessante suspender a configuração, substituir o componente *callee* por um novo e, então, retomar a configuração.
- Os controladores também oferecem recursos de reflexão (veja o Capítulo 2). Em particular, os recursos de *introspecção* são fornecidos por meio de duas interfaces, *Component* e *ContentController*, as quais suportam introspecção (descoberta dinâmica) das interfaces associadas a um componente e percorrem a arquite-

```

public interface Component {
    Object[] getFcInterfaces ();
    Object getFcInterface (String iftName);
    Type getFcType ();
}

public interface ContentController {
    Object[] getFcInternalInterfaces ();
    Object getFcInterfaceInterface (String iftName);
    Component[] getFcSubComponents ();
    void addFcSubComponent (Component c);
    void removeFcSubComponent (Component c);
}

```

Figura 8.18 Interfaces *Component* e *ContentController* no Fractal.

tura de uma estrutura de componente composta, respectivamente. As interfaces completas dos dois controladores estão mostradas na Figura 8.18. Novamente, a introspecção é importante para suportar reconfiguração dinâmica. Voltando a nosso exemplo de cliente-servidor, é possível, por meio das interfaces anteriores, descobrir a arquitetura precisa da configuração de componente subjacente (neste caso, uma configuração simples, consistindo em dois componentes) e também garantir que um componente *callee* substituto suporte precisamente a mesma interface do antigo.

- Também podem ser introduzidos controladores para oferecer recursos de interceptação, espelhando a capacidade oferecida no EJB e relatada na Seção 8.5.1. A interceptação é um mecanismo poderoso, com muitos usos. Na seção sobre EJB, é fornecido um exemplo de uso de interceptação para implementar registro. Por exemplo, a interceptação poderia ser usada no exemplo de cliente-servidor para registrar todas as chamadas feitas pelo componente *caller*, de uma maneira completamente transparente tanto para *caller* como para *callee*. Outro uso de interceptação é na implementação de uma política de controle de acesso que só permita o prosseguimento de uma invocação se determinado principal tiver direitos de acesso a determinado recurso (veja a Seção 11.2.4).

Tendo estudado as funções relativas das membranas e dos controladores, agora é possível relacionar as membranas aos contêineres, apresentados na Seção 8.4, e no estudo de caso do EJB anterior. Assim como os contêineres, as membranas fornecem um lugar para a distribuição de componentes; as duas técnicas também suportam gerenciamento de sistemas distribuídos implícito, os contêineres fazendo chamadas implícitas para serviços dos sistemas distribuídos e as membranas por meio de seus controladores constituintes. Contudo, as membranas são significativamente mais flexíveis:

- Em termos de reflexão, o suporte pode variar de componentes tipo caixa preta, em que a estrutura interna fica escondida, estratégias nas quais são oferecidos recursos de introspecção limitados (descobrindo interfaces dinamicamente), até recursos de reflexão avançados que suportam introspecção total e fornecem suporte inerente para a subsequente adaptação das estruturas internas.
- Em termos de suporte para preocupações não funcionais, em um extremo, as membranas não podem fornecer mais do que um simples encapsulamento de

componentes (se, por exemplo, controladores mínimos forem implementados); no outro extremo, elas podem suportar gerenciamento de sistemas distribuídos completo de componentes, incluindo o suporte para transação e segurança, como em servidores de aplicativo, mas de maneira completamente configurável e reconfigurável. Graças a essa flexibilidade inerente, o Fractal é referido como modelo de componente *aberto*.

Um exemplo funcional de implementação completa de Fractal pode ser encontrado no tutorial *online* [fractal.ow2.org II]. Esse exemplo mostra como o Fractal pode ser usado para implementar um servidor HTTP mínimo e configurável, chamado Comanche.

8.6 Resumo

Este capítulo examinou o projeto de soluções de *middleware* completas, baseadas em objetos distribuídos e componentes. Conforme ficará claro, elas representam uma evolução natural ao se pensar em tais abstrações de programação. Os objetos distribuídos são importantes para trazer as vantagens do encapsulamento e da abstração de dados para os sistemas distribuídos e também as ferramentas e técnicas associadas do campo do projeto orientado a objetos. Portanto, os objetos distribuídos representam um passo significativo em relação às estratégias anteriores, baseadas diretamente no modelo cliente-servidor. No entanto, na aplicação da estratégia de objeto distribuído, surgiram várias limitações significativas e elas foram apresentadas e analisadas neste capítulo. Em resumo, na prática, frequentemente é complexo demais usar soluções de *middleware* como o CORBA para aplicações e serviços distribuídos sofisticados, particularmente ao se lidar com propriedades avançadas de tais sistemas, como, por exemplo, a confiabilidade (tolerância a falhas e segurança).

As tecnologias de componente superaram essas limitações por meio de sua separação de preocupações intrínseca entre lógica da aplicação e gerenciamento de sistemas distribuídos. A identificação explícita de dependências também ajuda a suportar a composição terceirizada de sistemas distribuídos. Este capítulo examinou a especificação EJB 3.0, que fez avanços ao simplificar o desenvolvimento de sistemas distribuídos por meio de uma estratégia que enfatiza o uso de objetos Java puros, com as complexidades gerenciadas declarativamente por meio do uso de anotações Java. Como vimos, tecnologias mais leves, como Fractal e OpenCOM, também foram introduzidas para trazer as vantagens da programação baseada em componentes para o desenvolvimento das próprias plataformas de *middleware*, com pouca sobrecarga adicional em termos de desempenho.

As tecnologias de componente são importantes para o desenvolvimento de aplicações distribuídas, mas assim como qualquer tecnologia, têm suas vantagens e desvantagens. Por exemplo, a estratégia de componentes é muito prescritiva e mais conveniente para aplicações naturalmente semelhantes às arquiteruras de três camadas físicas. Para oferecer uma perspectiva mais ampla sobre a variedade de plataformas de *middleware* disponíveis, os dois próximos capítulos examinarão estratégias alternativas, baseadas na adoção de padrões da Web (serviços Web) e sistemas *peer-to-peer*.

Exercícios

- 8.1 A *Task Bag* (sacola de tarefas) é um objeto que armazena pares (chave e valor). Uma chave é um *string* e um valor é uma sequência de bytes. Sua interface fornece os seguintes métodos remotos:

pairOut: com dois parâmetros, através dos quais o cliente especifica uma *chave* e um *valor* a ser armazenado.

pairIn: cujo primeiro parâmetro permite que o cliente especifique a *chave* de um par a ser removido da *Task Bag*. O valor no par é fornecido para o cliente por intermédio de um segundo parâmetro. Se nenhum par correspondente estiver disponível, uma exceção será lançada.

readPair: é igual a *pairIn*, exceto que o par permanece na *Task Bag*.

Usando a IDL do CORBA, defina a interface da *Task Bag*. Defina uma exceção que possa ser disparada quando qualquer uma das operações não puder ser executada. Sua exceção deve retornar um valor inteiro indicando o número do problema e um *string* descrevendo o problema. A interface da *Task Bag* deve definir um único atributo, fornecendo o número de tarefas na sacola. [página 341](#)

- 8.2 Defina uma assinatura alternativa para os métodos *pairIn* e *readPair*, cujo valor de retorno indica quando nenhum par correspondente está disponível. O valor de retorno deve ser definido como um tipo enumerado cujos valores podem ser *ok* e *wait*. Discuta os méritos relativos das duas estratégias alternativas. Qual estratégia você usaria para indicar um erro como uma chave contendo caracteres inválidos? [página 342](#)
- 8.3 Qual dos métodos na interface da *Task Bag* poderia ter sido definido como uma operação *oneway*? Forneça uma regra geral a respeito dos parâmetros e exceções de métodos *oneway*. De que maneira o significado da palavra-chave *oneway* difere do restante da IDL? [página 342](#)
- 8.4 O tipo *union* da IDL pode ser usado para um parâmetro que precisará passar um entre poucos número de tipos. Utilize-o para definir o tipo de um parâmetro que às vezes é vazio e às vezes tem o tipo *Value*. [página 345](#)
- 8.5 Na Figura 8.2, o tipo *All* foi definido como uma sequência de comprimento fixo. Redefina isso como um vetor de mesmo comprimento. Dê algumas recomendações quanto à escolha entre vetores e sequências em uma interface IDL. [página 345](#)
- 8.6 A *Task Bag* se destina ao uso por clientes em cooperação, alguns dos quais adicionam pares (descrevendo tarefas) e outros os removem (e executam as tarefas descritas). Quando um cliente é informado de que não há um par correspondente disponível, ele não pode continuar seu trabalho até que um par se torne disponível. Defina uma interface de *callback* apropriada para uso nesta situação. [página 357](#)
- 8.7 Descreva as modificações necessárias na interface da *Task Bag* para permitir o uso de *callbacks*. [página 357](#)
- 8.8 Quais dos parâmetros dos métodos na interface da *Task Bag* são passados por valor e quais são passados por referência? [página 343](#)
- 8.9 Use o compilador de IDL Java para processar a interface que você definiu no Exercício 8.1. Ispicie a definição das assinaturas dos métodos *pairIn* e *readPair* no equivalente em Java gerado da interface IDL. Procure também, na definição gerada, o argumento de valor dos métodos *pairIn* e *readPair*. Agora, dê um exemplo mostrando como o cliente invocará o método *pairIn*, explicando como adquirirá o valor retornado por meio do segundo argumento. [página 346](#)
- 8.10 Dê um exemplo para mostrar como um cliente Java acessará o atributo que fornece o número de tarefas no objeto *Task Bag*. Sob que aspectos um atributo difere de uma variável de instância de um objeto? [página 345](#)

- 8.11 Explique por que as interfaces de objetos remotos, em geral, e os objetos CORBA, em particular, não fornecem construtores. Explique como os objetos CORBA podem ser criados na ausência de construtores. *Capítulo 5 e página 355*
- 8.12 Redefina a interface da *Task Bag* do Exercício 8.1 na IDL, de modo que ela faça uso de uma *struct* para representar um *Par*, o qual consiste em uma *Chave* e em um *Valor*. Note que não há necessidade de usar um *typedef* para definir uma *struct*. *página 345*
- 8.13 Discuta as funções do repositório de implementações do ponto de vista da capacidade de mudança de escala e da tolerância a falhas. *página 350, página 351*
- 8.14 Até que ponto os objetos CORBA podem ser migrados de um servidor para outro? *página 350, página 351*
- 8.15 Explique exatamente como o *middleware* baseado em componentes em geral e o EJB, em particular, podem superar as principais limitações do *middleware* de objeto distribuído. Dê exemplos para ilustrar sua resposta. *páginas 358 a 364*
- 8.16 Discuta se a arquitetura do EJB seria conveniente para implementar um jogo *online* para muitos jogadores (um domínio de aplicação apresentado inicialmente no Capítulo 1, Seção 1.2.2). Quais seriam as vantagens e desvantagens de usar EJB nesse domínio? *página 364*
- 8.17 O Fractal seria uma escolha de implementação mais conveniente para o domínio do MMOG? Justifique sua resposta. *página 372*
- 8.18 Explique como a filosofia baseada em contêiner poderia ser adotada para fornecer transparência de migração para componentes distribuídos. *página 362*
- 8.19 Como você obteria o mesmo efeito no Fractal? *página 375*
- 8.20 Considere a implementação de RMI Java como um vínculo composto no Fractal. Discuta até que ponto tal vínculo pode ser configurável e reconfigurável. *Capítulo 5 e página 373*

9

Serviços Web

- 9.1 Introdução
- 9.2 Serviços Web
- 9.3 Descrições de serviço e IDL para serviços Web
- 9.4 Um serviço de diretório para uso com serviços Web
- 9.5 Aspecto de segurança em XML
- 9.6 Coordenação de serviços Web
- 9.7 Aplicações de serviços Web
- 9.8 Resumo

Um serviço Web (*Web service*) fornece uma interface de serviço que permite aos clientes interagirem com servidores de uma maneira mais geral do que acontece com os navegadores Web. Os clientes acessam operações de um serviço Web por meio de requisições e respostas formatadas em XML e, normalmente, transmitidas por HTTP. Os serviços Web podem ser acessados de uma maneira mais *ad hoc* do que os serviços baseados em CORBA, permitindo que eles sejam mais facilmente usados em aplicações de Internet.

Assim como no CORBA e em Java, as interfaces dos serviços Web podem ser descritas em uma IDL. Entretanto, para os serviços Web, informações adicionais precisam ser descritas, incluindo a codificação e os protocolos de comunicação em uso e o local do serviço.

Os usuários exigem uma maneira segura de criar, armazenar e modificar documentos e trocá-los pela Internet. Os canais seguros TLS (Transport Layer Security, descrito no Capítulo 9) não fornecem todos os requisitos necessários. A segurança XML se destina a suprir essa falta.

Os serviços Web são cada vez mais importantes nos sistemas distribuídos: eles suportam atividade conjunta na Internet global, incluindo a área fundamental da integração de empresa para empresa (*business-to-business*, B2B) e também a emergente cultura de “*mashup*”, permitindo que desenvolvedores criem *software* inovador em cima da base de serviços já existente. Os serviços Web também fornecem o *middleware* subjacente para a computação de grade (*grid*) e em nuvem.

9.1 Introdução

O crescimento da Web nas últimas duas décadas (veja a Figura 1.6) prova a eficácia do uso de protocolos simples na Internet, como base para um grande número de serviços e aplicações remotos. Em particular, o protocolo de requisição-resposta HTTP (Seção 5.2) permite que clientes de propósito geral, chamados de navegadores, vejam páginas Web e outros recursos com referência aos seus URLs. Veja uma nota, no quadro a seguir, sobre URIs, URLs e URNs.

Entretanto, o uso de um navegador de propósito geral como cliente, mesmo com as melhorias fornecidas por *applets* específicos de uma aplicação, baixados por *download*, restringe a abrangência potencial dessas aplicações. No modelo cliente-servidor original, tanto o cliente como o servidor eram funcionalmente especializados. Os serviços Web (*Web services*) retornam a esse modelo, no qual um cliente específico da aplicação interage pela Internet com um serviço que possui uma interface funcionalmente especializada.

Assim, os serviços Web fornecem infraestrutura para manter uma forma mais rica e mais estruturada de interoperabilidade entre clientes e servidores. Eles fornecem uma base por meio da qual um programa cliente em uma organização pode interagir com um servidor em outra organização, sem supervisão humana. Em particular, os serviços Web permitem o desenvolvimento de aplicações complexas, fornecendo serviços que integram vários outros serviços. Devido à generalidade de suas interações, os serviços Web não podem ser acessados diretamente pelos navegadores.

O fornecimento de serviços Web como um acréscimo aos servidores Web é baseado na capacidade de usar uma requisição HTTP para provocar a execução de um programa. Lembre-se de que, em uma requisição HTTP, quando um URL se refere a um programa executável – por exemplo, uma busca – o resultado é produzido por esse programa e retornado. De maneira semelhante, os serviços Web são uma extensão da Web e podem ser fornecidos por servidores Web. Entretanto, seus servidores não precisam ser servidores Web. Os termos *servidor Web* e *serviços Web* não devem ser confundidos: um servidor Web fornece um serviço HTTP básico, enquanto um serviço Web fornece um serviço baseado nas operações definidas em sua interface.

A representação de dados externa e o empacotamento das mensagens trocadas entre clientes e serviços Web são feitos em XML, que foi descrita na Seção 4.3.3. Para recapitular, XML é uma representação textual que, embora mais volumosa do que as representações alternativas, foi adotada por sua legibilidade e pela consequente facilidade de depuração.

O protocolo SOAP (Seção 9.2.1) especifica as regras de uso da XML para empacotar mensagens, por exemplo, para suportar um protocolo de requisição-resposta. A Figura 9.1 resume os principais pontos a respeito da arquitetura de comunicação em que os serviços Web operam: um serviço Web é identificado por um URI e pode ser acessado pelos clientes usando mensagens formatadas em XML. O protocolo SOAP é usado para

URI, URL e URN • O URI (Uniform Resource Identifier) é um identificador de recurso geral, cujo valor pode ser um URL ou um URN. O URL, que inclui informações de localização do recurso, como o nome de domínio do servidor de um recurso que está sendo nomeado, é bem conhecido de todos os usuários da Web. Os URNs (Uniform Resource Names) são independentes da localização – eles contam com um serviço de pesquisa para fazer o mapeamento para os URLs dos recursos. Os URNs serão discutidos com mais detalhes na Seção 13.1.

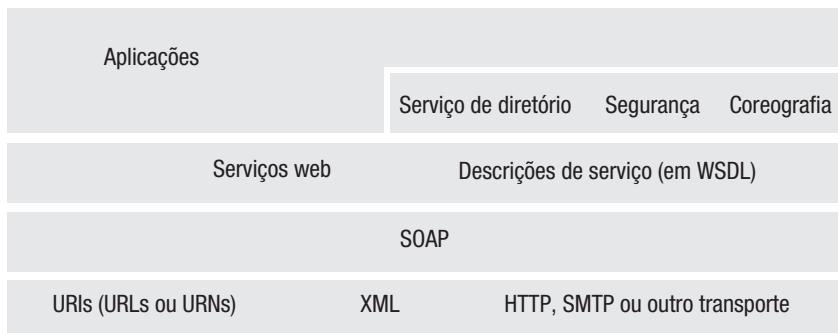


Figura 9.1 Infraestrutura e componentes dos serviços Web.

encapsular essas mensagens e transmiti-las por HTTP ou outro protocolo, por exemplo, TCP ou SMTP. Um serviço Web divulga para potenciais clientes a interface e outros aspectos dos serviços que implementa por meio das descrições de serviço.

A camada superior da Figura 9.1 ilustra o seguinte:

- Os serviços e aplicações Web podem ser construídos em cima de outros serviços Web.
- Alguns serviços Web fornecem a funcionalidade geral exigida para a operação de um grande número de outros serviços Web. Eles incluem os serviços de diretório, segurança e coreografia, todos os quais serão discutidos posteriormente neste capítulo.

Geralmente, um serviço Web fornece uma *descrição do serviço*, a qual inclui uma definição de interface e outras informações, como o URL do servidor. Isso é usado como base para um entendimento comum entre cliente e servidor quanto ao serviço oferecido. A Seção 9.3 apresentará a WSDL (Web Services Description Language).

Outra necessidade comum no *middleware* é um serviço de atribuição de nomes, ou de diretório, para permitir que os clientes descubram serviços. Os clientes de serviços Web têm necessidades semelhantes, mas frequentemente saem-se bem sem os serviços de diretório. Por exemplo, frequentemente, eles descobrem serviços a partir das informações presentes em uma página Web, por exemplo, como resultado de uma busca no Google. Entretanto, algum trabalho precisa ser feito para fornecer um serviço de diretório que seja conveniente para uso dentro das organizações. Isso será discutido na Seção 9.4.

A segurança em XML será apresentada na Seção 9.5. Nessa estratégia de segurança, documentos ou partes de documentos podem ser assinados ou cifrados. Um documento que possui elementos assinados ou cifrados pode, então, ser transmitido ou armazenado; posteriormente, podem ser feitas adições e estas também poderão ser assinadas ou cifradas.

Os serviços Web dão acesso a recursos de clientes remotos, mas não fornecem uma maneira de coordenar suas operações mútuas. A Seção 9.6 discutirá a coreografia dos serviços Web, a qual se destina a permitir que um serviço Web utilize padrões de acesso predefinidos no uso de um conjunto de outros serviços Web.

A última seção deste capítulo considera aplicações de serviços Web, incluindo o suporte para arquitetura orientada a serviços, a computação em grade e a computação em nuvem.

9.2 Serviços Web

Geralmente, uma interface de serviço Web consiste em um conjunto de operações que podem ser usadas por um cliente na Internet. As operações de um serviço Web podem ser fornecidas por uma variedade de recursos diferentes, por exemplo, programas, objetos ou bancos de dados. Um serviço Web pode ser gerenciado por um servidor Web, junto a páginas Web, ou pode ser um serviço totalmente separado.

A principal característica da maioria dos serviços Web é que eles podem processar mensagens SOAP formatadas em XML (veja a Seção 9.2.1). Uma alternativa é a estratégia REST, que está descrita em linhas gerais a seguir. Cada serviço Web usa sua própria descrição para tratar das características específicas das mensagens que recebe. Para uma boa narrativa de aspectos mais detalhados dos serviços Web, consulte Newcomer [2002] ou Alonso *et al.* [2004].

Muitos servidores Web comerciais conhecidos, incluindo Amazon, Yahoo, Google e eBay, oferecem interfaces de serviço que permitem aos clientes manipular seus recursos Web. Como exemplo, o serviço Web oferecido pela Amazon.com fornece operações que permitem aos clientes obter informações sobre produtos, adicionar um item a um carrinho de compras ou verificar o status de uma transação. Os serviços Web da Amazon [associates.amazon.com] podem ser acessado por SOAP ou por REST. Isso permite que aplicações de outros fornecedores construam serviços com valor agregado sobre aqueles fornecidos pela Amazon.com. Por exemplo, uma aplicação de controle de inventário e aquisição poderia pedir o fornecimento de mercadorias da Amazon.com, à medida que elas fossem necessárias, e controlar automaticamente a mudança de status de cada requisição. Mais de 50.000 desenvolvedores se registraram para uso desses serviços Web nos dois primeiros anos após eles terem sido introduzidos [Greenfield e Dornan 2004].

Outro exemplo interessante de aplicação que exige a presença de um serviço Web é a que implementa *sniping* em leilões da eBay. *Sniping* significa fazer um lance durante os últimos segundos antes que um leilão termine. Embora os seres humanos possam realizar as mesmas ações, por meio da interação direta com a página Web, eles não podem fazer isso com tanta rapidez.

Combinação de serviços Web • O fornecimento de uma interface de serviço permite que suas operações sejam combinadas com as de outros serviços para fornecer nova funcionalidade (veja também a Seção 9.7.1). A aplicação de aquisição mencionada anteriormente também poderia estar usando outros fornecedores. Como outro exemplo das vantagens de combinar vários serviços, considere o fato de que atualmente as pessoas utilizam seus navegadores para fazer reservas de passagens aéreas e de hotéis e para alugar carros com uma seleção de sites Web diferentes. Entretanto, se cada um desses sites Web fornecesse uma interface de serviço padrão, um serviço de agente de viagens poderia usar suas operações para fornecer ao viajante uma combinação desses serviços. Esse ponto está ilustrado na Figura 9.2.

Padrões de comunicação • O serviço de agente de viagens ilustra o possível uso dos dois padrões de comunicação alternativos disponíveis nos serviços Web:

- O processamento de uma reserva demora um tempo longo para terminar, e bem poderia ser suportado por uma troca assíncrona de documentos, começando com os detalhes das datas e destinos, seguida do retorno das informações de status de tempos em tempos e, finalmente, dos detalhes da conclusão. O desempenho não é problema neste caso.

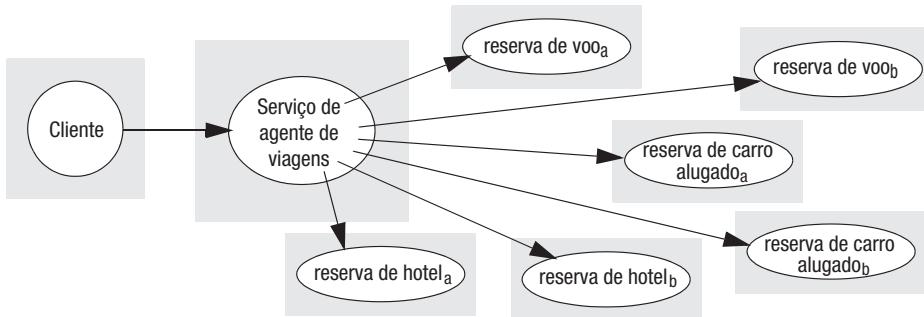


Figura 9.2 O serviço de agente de viagens combina vários serviços Web.

- A verificação dos detalhes do cartão de crédito e as interações com o cliente devem ser fornecidas por um protocolo de requisição-resposta.

Em geral, os serviços Web usam um padrão de comunicação de requisição-resposta síncrona com seus clientes, ou se comunicam por meio de mensagens assíncronas. Este último estilo de comunicação pode ser usado mesmo quando as requisições exigem respostas, no caso em que o cliente envia uma requisição e posteriormente recebe a resposta de forma assíncrona. Um padrão de estilo evento também pode ser usado: por exemplo, os clientes de um serviço de diretório podem se registrar nos eventos de interesse e serão notificados quando certos eventos ocorrerem. Por exemplo, um evento poderia ser a chegada ou a saída de um serviço.

Para possibilitar o uso de uma variedade de padrões de comunicação, o protocolo SOAP (Seção 9.2.1) é baseado no empacotamento de mensagens unidirecionais únicas. Ele suporta interações de requisição-resposta usando pares de mensagens únicas e especificando como irá representar as operações, seus argumentos e resultados.

Mais geralmente, os serviços Web são projetados para suportar computação distribuída na Internet, na qual coexistem diferentes linguagens de programação e paradigmas. Assim, eles são projetados para serem independentes de qualquer paradigma de programação em particular. Isso contrasta, por exemplo, com os objetos distribuídos, que defendem um paradigma de programação bastante específico para o desenvolvedores (uma discussão mais a fundo sobre as distinções entre os serviços Web e os objetos distribuídos pode ser encontrada na Seção 9.2.2).

Baixo acoplamento • Há um interesse considerável no *baixo acoplamento* em sistemas distribuídos, particularmente na comunidade de serviços Web. Contudo, é comum a terminologia ser mal definida e imprecisa. No contexto de serviços Web, baixo acoplamento se refere a minimizar as dependências entre os serviços para se ter uma arquitetura subjacente flexível (reduzindo o risco de uma alteração em um serviço causar uma reação em cadeia em outros serviços). Isso é parcialmente suportado pela independência pretendida pelos serviços Web, com a intenção subsequente de produzir combinações de serviços Web, conforme discutido anteriormente. Entretanto, o baixo acoplamento é melhorado por meio de vários recursos adicionais:

- A programação com interfaces (discutida no Capítulo 5) fornece um nível de baixo acoplamento, separando a interface de sua implementação (suportando também áreas de heterogeneidade importantes – por exemplo, na escolha da linguagem de

programação e da plataforma usadas). A programação com interfaces é adotada pela maioria dos paradigmas de sistemas distribuídos, incluindo objetos distribuídos e componentes (discutidos no Capítulo 8), assim como serviços Web.

- Há uma tendência em usar interfaces simples e genéricas em sistemas distribuídos, e isso é exemplificado pela interface mínima oferecida pela World Wide Web e pela estratégia REST nos serviços Web. Essa estratégia contribui para o baixo acoplamento por reduzir a dependência em relação a nomes de operação específicos (o estudo de caso do Google, no Capítulo 21, fornece um bom exemplo desse estilo de programação distribuída). Uma consequência disso é que os dados se tornam mais importantes do que a operação, com a semântica da operação conjunta frequentemente mantida nos dados (por exemplo, a definição de documento XML associada em serviços Web); essa visão *orientada a dados* é discutida mais a fundo no contexto dos sistemas móveis na Seção 19.3.1.
- Conforme mencionado anteriormente, os serviços Web podem ser usados com uma variedade de paradigmas de comunicação, incluindo comunicação por requisição-resposta, troca de mensagens assíncrona ou mesmo paradigmas de comunicação indireta (conforme apresentado no Capítulo 6). O nível de acoplamento é afetado diretamente por essa escolha. Por exemplo, na comunicação por requisição-resposta, os dois participantes são intrinsecamente acoplados; a troca de mensagens assíncrona oferece certo grau de desacoplamento (referido no Capítulo 6 como desacoplamento em relação ao sincronismo), enquanto a comunicação indireta também oferece desacoplamento em relação ao tempo e ao espaço.

Em conclusão, existem várias dimensões no baixo acoplamento, e é importante ter isso em mente ao se utilizar o termo. Os serviços Web suportam intrinsecamente um nível de baixo acoplamento, devido à filosofia de projeto adotada e à estratégia de programação com interfaces usada. Isso pode ser melhorado por escolhas de projeto adicionais, incluindo a adoção da estratégia REST e o uso de comunicação indireta.

Representação de mensagens • Tanto o protocolo SOAP quanto os dados que ele transporta são representados em XML – um formato textual auto-descritivo apresentado na Seção 4.3.3. As representações textuais ocupam mais espaço do que as binárias e exigem mais tempo para seu processamento. Nas interações de estilo documento, a velocidade não é problema, mas ela é importante nas interações tipo requisição-resposta. Entretanto, pode ser argumentado que há uma vantagem em ser um formato legível para seres humanos, que facilita a construção de mensagens simples e a depuração das mais complexas. Ele também permite que um usuário veja o texto de uma mensagem na sua frente. No entanto, existem situações em que ela é lenta demais.

REST (REpresentational State Transfer) • REST [Fielding 2000] é uma estratégia com um estilo de operação muito restrito, no qual os clientes usam URLs e as operações HTTP *GET*, *PUT*, *DELETE* e *POST* para manipular recursos representados em XML. A ênfase está na manipulação de recursos de dados, em vez de interfaces. Quando um novo recurso é criado, ele recebe um novo URL por meio do qual pode ser acessado ou atualizado. Os clientes recebem o estado inteiro de um recurso, em vez de chamar uma operação para fornecer alguma parte dele. Fielding acredita que, no contexto da Internet, a proliferação de diferentes interfaces de serviço não será tão útil quanto um conjunto mínimo de operações simples e uniformes. É interessante notar que, de acordo com Greenfield e Dornan [2004], 80% dos pedidos para os serviços Web na Amazon.com são feitos por intermédio da interface REST, com os 20% restantes usando SOAP.

Cada item em uma descrição em XML é anotado com seu tipo, e o significado de cada tipo é definido por um esquema referenciado dentro da descrição. Isso torna o formato extensível, permitindo que qualquer tipo de dados seja transportado. Não há limite para a riqueza e complexidade em potencial dos documentos formatados em XML, mas poderia haver um problema na interpretação daqueles que se tornassem excessivamente complexos.

Referências de serviço • Em geral, cada serviço Web tem um URI, o qual os clientes utilizam para se referirem a ele. O URL é a forma mais frequentemente usada de URI. Como um URL contém o nome de domínio de um computador, o serviço ao qual ele se refere sempre será acessado nesse computador. Entretanto, o ponto de acesso de um serviço Web com um URN pode depender do contexto e mudar de tempos em tempos – seu URL atual pode ser obtido a partir de um serviço de pesquisa de URN. Essa referência de serviço é conhecida nos serviços Web como *destino final (endpoint)*.

Ativação de serviços • Um serviço Web será acessado por meio do computador cujo nome de domínio está incluído em seu URL corrente. Esse computador pode, ele próprio, executar o serviço Web ou executá-lo em outro computador servidor. Por exemplo, um provedor de serviço com dezenas de milhares de clientes precisará implantar centenas de computadores para fornecer esse serviço. Um serviço Web pode funcionar continuamente ou ser ativado sob demanda. O URL é uma referência persistente – significando que ele continuará a se referir ao serviço enquanto esse servidor existir.

Transparência • Uma tarefa importante de muitas plataformas de *middleware* é proteger o programador dos detalhes da representação e do empacotamento dos dados e, às vezes, fazer as invocações remotas parecerem ser locais. Nada disso é fornecido como parte de uma infraestrutura ou plataforma de *middleware* para serviços Web. No nível mais simples, clientes e servidores podem ler e gravar suas mensagens diretamente em SOAP, usando XML.

Porém, por conveniência, os detalhes do protocolo SOAP e da XML geralmente são ocultos por uma API local, em uma linguagem de programação como Java, Perl, Python ou C++. Nesse caso, a descrição do serviço pode ser usada como base para a geração automática dos procedimentos de empacotamento e desempacotamento necessários.

Proxies: uma opção para ocultar a diferença entre chamadas locais e remotas é fornecer um *proxy* cliente ou um conjunto de procedimentos *stub*. A Seção 9.2.3 explicará como isso é feito em Java. Os *proxies* clientes, ou *stubs*, fornecem uma forma estática de invocação na qual a estrutura de cada chamada e os procedimentos de empacotamento são gerados antes que quaisquer invocações sejam feitas.

Invocação dinâmica: uma alternativa para os *proxies* é fornecer aos clientes uma operação genérica para ser usada independentemente do procedimento remoto a ser chamado, semelhante ao procedimento *DoOperation* definido na Figura 5.3 (mas sem o primeiro argumento). Nesse caso, o cliente especifica o nome de uma operação e seus argumentos, e eles são convertidos dinamicamente para SOAP e XML. A comunicação assíncrona de mensagens únicas pode ser obtida de maneira semelhante, fornecendo-se aos clientes operações genéricas para enviar e receber mensagens.

9.2.1 SOAP

O protocolo SOAP (Simple Object Access Protocol) é projetado para permitir tanto interação cliente-servidor como assíncrona pela Internet. Ele define um esquema para uso da XML para representar o conteúdo de mensagens de requisição-resposta (veja a Figura 5.4), assim como um esquema para a comunicação de documentos. Originalmente, o

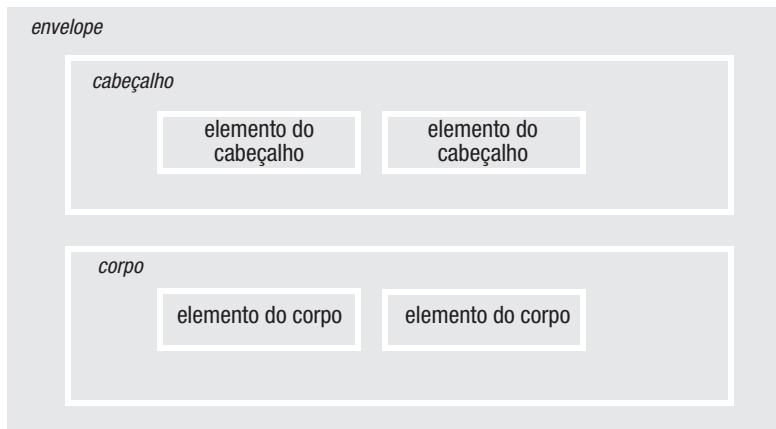


Figura 9.3 Mensagem SOAP em um envelope.

protocolo SOAP era baseado apenas em HTTP, mas a versão atual é projetada para usar uma variedade de protocolos de transporte*, incluindo SMTP, TCP ou UDP. A descrição desta seção é baseada no protocolo SOAP versão 1.2 [www.w3.org/IX], que é uma recomendação do W3C (World Wide Web Consortium). O protocolo SOAP é uma extensão do XML-RPC da Userland [Winer 1999].

A especificação do protocolo SOAP declara:

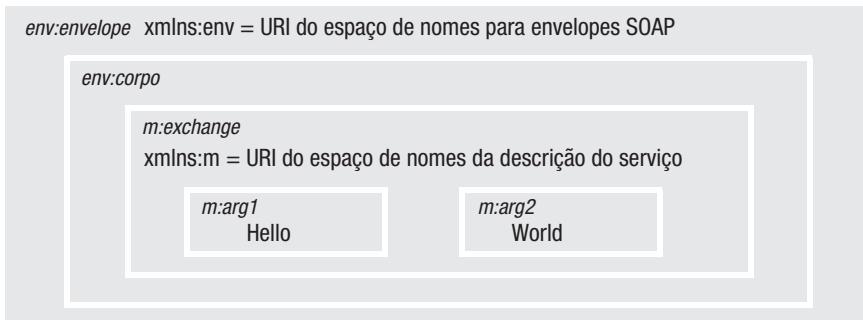
- como a XML deve ser usada para representar o conteúdo de mensagens individuais;
- como duas mensagens podem ser combinadas para produzir um padrão de requisição-resposta;
- as regras sobre como os destinatários das mensagens devem processar os elementos XML que elas contêm;
- como HTTP e SMTP devem ser usados para comunicar mensagens SOAP. É esperado que as versões futuras da especificação definam como usar outros protocolos de transporte, por exemplo, TCP.

Esta seção descreve como o protocolo SOAP usa XML para representar mensagens e HTTP para comunicação. Entretanto, normalmente o programador não precisa se preocupar com esses detalhes, pois APIs SOAP foram implementadas em muitas linguagens de programação, incluindo Java, Javascript, Perl, Python.NET, C, C++, C# e Visual Basic.

Para suportar comunicação cliente-servidor, o protocolo SOAP especifica como se faz para usar o método HTTP *POST* para a mensagem de requisição e para a mensagem de resposta. O uso combinado de XML e HTTP fornece um protocolo padrão para comunicação cliente-servidor pela Internet.

Pretende-se que uma mensagem SOAP possa ser passada por meio de intermediários no caminho para o computador que gerencia o recurso a ser acessado, e que serviços de *middleware* de nível mais alto, como transações ou segurança, possam usar esses intermediários para realizar o processamento.

* N. de R.T.: O leitor habituado com o modelo de referência OSI (Open System Interconnection), comumente usado na área de redes de computadores, pode estranhar a menção aos protocolos HTTP e SMTP como protocolos de transporte. Entretanto, em serviços Web, o termo *protocolo de transporte* é empregado para referenciar qualquer protocolo que sirva como “meio de transporte” para uma mensagem SOAP.



Nesta figura e na próxima, cada elemento XML é representado por uma caixa com seu nome em itálico, seguido dos atributos e de seu conteúdo.

Figura 9.4 Exemplo de uma requisição simples sem cabeçalhos.

Mensagens SOAP • Uma mensagem SOAP é transportada em um *envelope*. Dentro do envelope existe um cabeçalho opcional e um corpo, como mostrado na Figura 9.3. Os cabeçalhos das mensagens podem ser usados para estabelecer o contexto necessário para um serviço ou para manter um *log* ou uma auditoria das operações. Um intermediário pode interpretar e atuar sobre as informações presentes nos cabeçalhos das mensagens, por exemplo, adicionando, alterando ou removendo informações. O corpo da mensagem transporta um documento XML para um serviço Web em particular.

Os elementos XML *envelope*, *cabeçalho* e *corpo*, juntos a outros atributos e elementos de mensagens SOAP, são definidos como um esquema no espaço de nomes XML do protocolo SOAP. A definição desse esquema pode ser encontrada no site Web do W3C [[www.w3.org IX](http://www.w3.org/2003/05/soap/bindings/HTTP/)]. Como utilizam uma codificação textual, os esquemas XML podem ser vistos com a opção “exibir código-fonte” de um navegador. Tanto o cabeçalho como o corpo contêm elementos internos.

A seção anterior explicou que a descrição de serviço contém informações que devem ser compartilhadas por clientes e servidores. Os remetentes de mensagens usam essas descrições para gerar o corpo e para garantir que ele possua o conteúdo correto, e os destinatários das mensagens as utilizam para analisar e verificar a validade do conteúdo.

Uma mensagem SOAP pode ser usada para transmitir um documento ou para suportar comunicação cliente-servidor:

- Um documento a ser comunicado é colocado diretamente dentro do elemento *corpo*, junto a uma referência a um esquema XML contendo a descrição do serviço – a qual define os nomes e tipos usados no documento. Esse tipo de mensagem SOAP pode ser enviado de forma síncrona ou de forma assíncrona.
- Para comunicação cliente-servidor, o elemento *corpo* contém uma requisição (*Request*) ou uma resposta (*Reply*). Esses dois casos estão ilustrados nas Figuras 9.4 e 9.5.

A Figura 9.4 mostra um exemplo de mensagem de requisição simples, sem cabeçalhos. O *corpo* engloba um elemento com o nome do procedimento a ser chamado e o URI do espaço de nomes (o arquivo que contém o esquema XML) para a descrição do serviço relevante, que é denotada por *m*. Os elementos internos de uma mensagem de requisição contêm os argumentos do procedimento. Essa mensagem de requisição fornece dois *strings* a serem retornados na ordem oposta pelo procedimento que está no servidor. O espaço de nomes XML, denotado por *env*, contém as definições SOAP de um

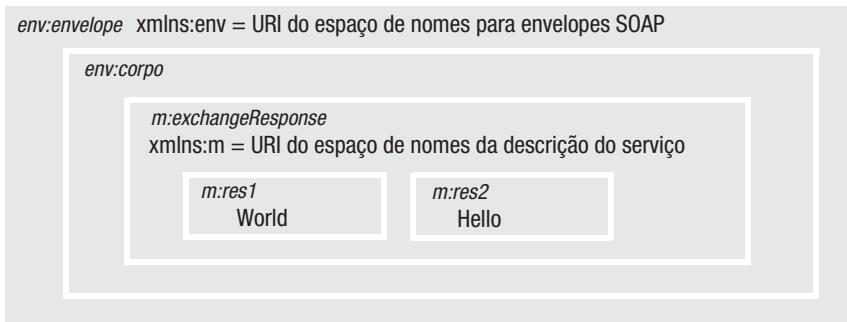


Figura 9.5 Exemplo de resposta correspondente à requisição da Figura 9.4.

envelope. A Figura 9.5 mostra a mensagem de resposta bem-sucedida correspondente, a qual contém os dois argumentos de saída. Note que o nome do procedimento tem *Response* anexado a ele. Se um procedimento tem um valor de retorno, então ele pode ser denotado como um elemento chamado *rpc: result*. Note que a mensagem de resposta usa os mesmos dois esquemas XML da mensagem de requisição, o primeiro definindo o envelope SOAP e o segundo definindo os nomes do procedimento e do argumento específicos da aplicação.

Erros SOAP: Se uma requisição falha de alguma maneira, as descrições do erro são transmitidas no corpo de uma mensagem de resposta em um elemento *fault*. Esse elemento contém informações sobre o erro, incluindo um código e um *string* associado, junto a detalhes específicos da aplicação.

Cabeçalhos SOAP • Os cabeçalhos das mensagens se destinam ao uso pelos intermediários para adicionar um serviço responsável por tratar a mensagem transportada no corpo. Entretanto, dois aspectos dessa utilização não ficam claros na especificação do SOAP:

1. A forma como os cabeçalhos serão usados por qualquer serviço de *middleware* de mais alto nível, em particular. Por exemplo, um cabeçalho poderia conter:
 - um identificador de transação para uso em um serviço de transação;
 - um identificador de mensagem para relacionar uma mensagem com outra, por exemplo, para implementar distribuição confiável;
 - um nome de usuário, uma assinatura digital ou uma chave pública.
2. A maneira como as mensagens serão direcionadas por meio de um conjunto de intermediários para o destinatário final. Por exemplo, uma mensagem transportada por HTTP poderia ser direcionada por meio de um encadeamento de servidores *proxies*, alguns dos quais poderiam assumir o papel do SOAP.

Entretanto, a especificação define as funções e tarefas dos intermediários. Um atributo chamado *papel* (*role*) pode especificar se todo intermediário, nenhum deles ou apenas o destinatário final deve processar o elemento (consulte [[www.w3.org IX](http://www.w3.org/2005/08/xop/doc/IX.html)]). As ações em particular a serem executadas são definidas pelas aplicações; por exemplo, uma ação poderia ser o registro do conteúdo de um elemento.

Transporte de mensagens SOAP • Um protocolo de transporte é exigido para enviar uma mensagem SOAP para seu destino. As mensagens SOAP são independentes do tipo de

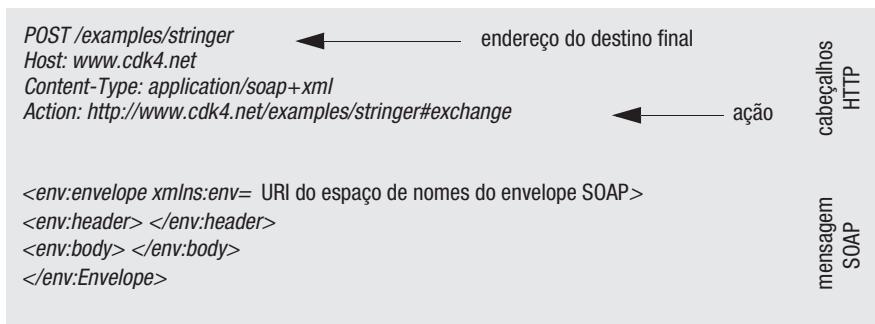


Figura 9.6 Uso de requisição HTTP *POST* na comunicação cliente-servidor do protocolo SOAP.

transporte usado – seus envelopes não contêm nenhuma referência ao endereço de destino. Fica para o protocolo HTTP (ou qualquer protocolo usado para transporte de uma mensagem SOAP) especificar o endereço do destino.

A Figura 9.6 ilustra como o método HTTP *POST* é usado para transmitir uma mensagem SOAP. Os cabeçalhos e o corpo HTTP são usados como segue:

- os cabeçalhos HTTP especificam o endereço do destino final (o URI do receptor final) e a ação a ser executada. O parâmetro *Action* se destina a otimizar o envio – revelando o nome da operação, sem a necessidade de analisar a mensagem SOAP presente no corpo da mensagem HTTP;
- o corpo HTTP transporta a mensagem SOAP.

Como o protocolo HTTP é síncrono, ele é usado para retornar uma resposta contendo a resposta SOAP; por exemplo, aquela mostrada na Figura 9.5. A Seção 5.2 pormenorizou os códigos de *status* e os motivos retornados pelo protocolo HTTP para requisições bem-sucedidas e malsucedidas.

Se uma requisição SOAP é apenas uma solicitação para um retorno de informações, não tendo argumentos e sem alterar dados no servidor, então o método HTTP *GET* pode ser usado para transportá-la.

A questão acima sobre o cabeçalho *Action* e sobre o envio se aplica a qualquer serviço que execute uma variedade de ações diferentes para os clientes, mesmo que ele não ofereça essas operações. Por exemplo, um serviço Web pode ser capaz de lidar com diferentes tipos de documentos, como requisições de compra e de consultas, os quais são tratados por diferentes módulos de *software*. O cabeçalho *Action* permite que o módulo correto seja escolhido sem inspecionar a mensagem SOAP. Esse cabeçalho pode ser usado, caso o tipo de conteúdo HTTP seja especificado como *application/soap+xml*.

A separação da definição do envelope SOAP das informações sobre como e para onde elas devem ser enviadas torna possível usar uma variedade de protocolos subjacentes diferentes. A especificação SOAP diz como o protocolo SMTP pode ser usado como uma maneira alternativa de transmitir documentos codificados como mensagens SOAP.

No entanto, essa vantagem também é uma desvantagem. Ela implica que o desenvolvedor deve estar envolvido nos detalhes do protocolo de transporte específico escolhido. Além disso, torna difícil usar diferentes protocolos para diferentes partes da rota seguida por uma mensagem em particular.

WS-Addressing: avanços no endereçamento e no direcionamento do protocolo SOAP • Dois problemas foram mencionados anteriormente:

- como tornar o protocolo SOAP independente do transporte subjacente usado;
- e como especificar uma rota a ser seguida por uma mensagem SOAP por meio de um conjunto de intermediários.

Um trabalho anterior nessa área, realizado por Nielsen e Thatte [2001], sugere que o endereço do destino final e a informação de envio devem ser especificados nos cabeçalhos SOAP. Isso separa efetivamente o destino da mensagem do protocolo subjacente. Eles sugeriram especificar o caminho a ser seguido fornecendo o endereço do destino final e o próximo passo (ou etapa). Cada um dos intermediários atualizaria a informação do próximo passo.

O trabalho de Box e Curbura [2004] sugere que fazer os intermediários alterarem os cabeçalhos poderia levar a brechas de segurança. Eles propuseram o *WS-Addressing*, que permite ao cabeçalhos SOAP especificar dados de roteamento de mensagem, com uma infraestrutura SOAP subjacente fornecendo a informação do próximo passo.

As recomendações do W3C para o *WS-Addressing* estão definidas no endereço [[www.w3.org XXIII](http://www.w3.org/2004/07/ws-addressing/)]. Essa forma de endereçamento usa uma *referência de destino final (Endpoint Reference)* – uma estrutura XML contendo o endereço de destino, informações de roteamento e, possivelmente, outras informações sobre o serviço. Para suportar interações assíncronas de longa duração, os cabeçalhos SOAP podem fornecer um endereço de retorno e identificadores de mensagem próprios e de mensagens relacionadas.

WS-ReliableMessaging: comunicação confiável • O protocolo normal do SOAP, HTTP, é executado sobre TCP, cujo modelo de falha foi discutido na Seção 4.2.4. Em resumo: o protocolo TCP não garante o envio de mensagens em face de todas as dificuldades – e quando atinge um tempo limite na espera por confirmações, ele declara que a conexão está desfeita, no ponto em que os processos que estão se comunicando ficam sem saber se as mensagens que enviaram recentemente foram recebidas ou não.

O trabalho anterior visando ao fornecimento de comunicação confiável de mensagens SOAP com distribuição garantida, sem duplicação e com a ordem das mensagens assegurada, levou a duas especificações concorrentes de Ferris e Langworthy [2004] e Evans *et al.* [2003].

Mais recentemente, o Oasis (um consórcio global que trabalha no desenvolvimento, acordo e adoção de padrões para comércio eletrônico e serviços Web) elaborou uma recomendação chamada *WS-ReliableMessaging* [[www.oasis.org](http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-reliablemessaging)]. Isso permite que uma mensagem SOAP seja enviada *pelo menos uma vez, no máximo uma vez ou exatamente uma vez*, com a seguinte semântica:

Pelo menos uma vez: a mensagem é entregue pelo menos uma vez, mas se não puder ser entregue, um erro será relatado.

No máximo uma vez: a mensagem é entregue no máximo uma vez, mas se não puder ser entregue, nenhum erro é relatado.

Exatamente uma vez: a mensagem é entregue exatamente uma vez, mas se não puder ser entregue, um erro será relatado.

A ordem das mensagens também é fornecida, em combinação com qualquer um dos modos anteriores:

Em ordem: as mensagens serão entregues para o destino na ordem em que foram enviadas por um remetente em particular.

Note que a recomendação *WS-ReliableMessaging* se preocupa com o envio de mensagens únicas e não deve ser confundida com a semântica de chamada RPC, descrita na Seção 5.3.1, que se refere ao número de vezes que o servidor executa o procedimento remoto. No Exercício 9.16, o leitor verá uma consideração melhor sobre a comparação.

Passando por firewalls • Os serviços Web se destinam a serem usados por clientes de uma organização que acessam servidores de outra pela Internet. A maioria das organizações utiliza um *firewall* para proteger os recursos presentes em suas próprias redes, e os protocolos de transporte, como os usados pela RMI Java ou CORBA, normalmente não seriam capazes de atravessar um *firewall*. Entretanto, os *firewalls* normalmente permitem que mensagens HTTP e SMTP passem por eles. Portanto, é conveniente usar um desses protocolos para transportar mensagens SOAP.

9.2.2 Uma comparação de serviços Web com o modelo de objeto distribuído

Um serviço Web tem uma interface que pode fornecer operações para acessar e atualizar os recursos de dados que gerencia. De forma superficial, a interação entre cliente e servidor é muito parecida com RMI, em que um cliente usa uma referência de objeto remoto para invocar uma operação em um objeto remoto. Para um serviço Web, o cliente usa um URI para invocar uma operação no recurso nomeado por esse URI. Para argumentos sobre as semelhanças e diferenças entre serviços Web e objetos distribuídos, consulte Birman [2004], Vinoski [2002] e Vogels [2003].

Tentaremos mostrar que existem limites para a analogia acima, fazendo uso do exemplo do quadro compartilhado utilizado na Seção 5.5 para Java RMI e na Seção 8.3 para CORBA.

Referências de objeto remoto versus URIs • O URI de um serviço Web pode ser comparado com a referência de objeto remoto de um único objeto. Entretanto, no modelo de objeto distribuído, os objetos podem criar objetos remotos dinamicamente e retornar referências remotas para eles. O destinatário dessas referências remotas pode usá-las para invocar operações nos objetos a que se referem. No exemplo do quadro compartilhado, uma invocação do método de fábrica *newShape* faz uma nova instância de *Shape* ser criada e uma referência remota ser retornada para ela. Nada parecido com isso pode ser feito com serviços Web, que não podem criar instâncias de objetos remotos; com efeito, um serviço Web consiste em um único objeto remoto e, assim, tanto a coleta de lixo como a referência a objeto remoto são irrelevantes.

Modelo de serviços Web • Os usuários dos *toolkits* de serviços Web Java (JAX-RPC) [[java.sun.com VII](#)] devem modelar seus programas que usam serviços Web considerando o fato de que não estão usando invocação remota transparente de Java para Java, mas sim usando o modelo de serviços Web, no qual objetos remotos não podem ser instanciados. Isso é levado em conta pelo JAX-RPC, que não permite que referências de objeto remoto sejam passadas como argumentos, nem retornadas como resultados.

A Figura 9.7 mostra uma versão da interface apresentada na Figura 5.16, que foi modificada, como segue, para se tornar uma interface de serviços Web:

- Na versão original (objeto distribuído) do programa, instâncias de *Shape* são criadas no servidor e referências remotas são retornadas para eles por *newShape*, cuja versão (serviço Web) modificada é mostrada na linha 1. Para evitar a instanciação

```

import java.rmi.*;
public interface ShapeList extends Remote {
    int newShape(GraphicalObject g) throws RemoteException;
    int numberOfShapes() throws RemoteException;
    int getVersion() throws RemoteException;
    int getGOVersion(int i) throws RemoteException;
    GraphicalObject getAllState(int i) throws RemoteException;
}

```

1

Figura 9.7 Interface de serviço Web Java *ShapeList*.

de objetos remotos e o consequente uso de referências de objeto remoto, a interface *Shape* foi removida e suas operações (*getAllState* e *getGOVersion* – originalmente *getVersion*) foram adicionadas na interface *ShapeList*.

- Na versão original (objeto distribuído) do programa, o servidor armazenava um vetor de *Shape*. Isso foi mudado para um vetor de *GraphicalObject*. A nova versão (serviço Web) do método *newShape* retorna um valor inteiro que fornece o deslocamento do objeto *GraphicalObject* nesse vetor.

Essa alteração no método *newShape* significa que ele não é mais um método de fábrica – isto é, ele não cria instâncias de objetos remotos.

Serventes • No modelo de objeto distribuído, o programa servidor geralmente é modelado como um conjunto de serventes (potencialmente, objetos remotos). Por exemplo, a aplicação de quadro compartilhado usava um servente para a lista de figuras e um servente para cada objeto gráfico criado. Esses serventes eram criados como instâncias das classes serventes *ShapeList* e *Shape*, respectivamente. Quando o servidor iniciava, sua função *main* criava a instância de *ShapeList*, e, sempre que o cliente chamava o método *newShape*, o servidor criava uma instância de *Shape*.

Em contraste, os serviços Web não suportam serventes. Portanto, as aplicações de serviços Web não podem criar serventes como e quando eles são necessários para manipular diferentes recursos do servidor. Para impor essa situação, as implementações de interfaces de serviço Web não devem ter construtores nem métodos principais.

9.2.3 O uso de SOAP com Java

A API Java para desenvolvimento de serviços Web e clientes em SOAP é chamada de JAX-RPC. Ela está descrita no exercício dirigido sobre serviços Web Java [[java.sun.com VII](#)]. Essa API oculta todos os detalhes do protocolo SOAP dos programadores tanto de clientes como de serviços.

A JAX-RPC faz o mapeamento de alguns dos tipos da linguagem Java para definições em XML usadas tanto em mensagens SOAP como em descrições de serviço. Os tipos permitidos incluem *Integer*, *String*, *Date* e *Calendar*, assim como *java.net.uri*, que permite que URIs sejam passados como argumentos ou retornados como resultados. Ela suporta alguns dos tipos de coleção (incluindo *Vector*), assim como os tipos primitivos da linguagem e *vetores*.

Além disso, instâncias de algumas classes podem ser passadas como argumentos e resultados de chamadas remotas, desde que:

- Cada uma de suas variáveis de instância seja de um dos tipos permitidos.
- Elas tenham um construtor público padrão.

```

import java.util.Vector;
public class ShapeListImpl implements ShapeList{
    private Vector theList = new Vector();
    private int version = 0;
    private Vector theVersions = new Vector();

    public int newShape(GraphicalObject g) throws RemoteException{
        version++;
        theList.addElement(g);
        theVersions.addElement(new Integer(version));
        return theList.size();
    }
    public int numberShapes(){}
    public int getVersion(){}
    public int getGOVersion(int i){}
    public GraphicalObject getAllState(int i){}
}

```

Figura 9.8 Implementação em Java do servidor *ShapeList*.

- Elas não implementem a interface *Remote*.

Em geral, conforme mencionado na seção anterior, os valores de tipos que são referências remotas (isto é, que implementam a interface *Remote*) não podem ser passados como argumentos, nem retornados como resultados de chamadas remotas.

A interface de serviço • A interface Java de um serviço Web deve obedecer às regras a seguir, algumas das quais estão ilustradas na Figura 9.7:

- Ela deve estender a interface *Remote*.
- Ela não deve ter declarações constantes, como *public final static*.
- Os métodos devem disparar a exceção *java.rmi.RemoteException* ou uma de suas subclasses.
- Os parâmetros e tipos de retorno do método devem ser tipos permitidos na JAX-RPC.

O programa servidor • A classe que implementa a interface *ShapeList* aparece na Figura 9.8. Conforme explicado anteriormente, não há nenhum método *main*, e a implementação da interface *ShapeList* não tem construtor. Na verdade, um serviço Web é um objeto único que oferece um conjunto de procedimentos. O código-fonte dos programas mostrados nas Figuras 9.7, 9.8 e 9.9 está disponível na página deste livro, em www.grupoa.com.br ou em www.cdk5.net/Web.

A interface de serviço e sua implementação são compiladas normalmente. Duas ferramentas, chamadas *wscompile* e *wsdeploy*, podem ser usadas para gerar a classe esqueleto e a descrição do serviço (em WSDL, conforme descrito na Seção 9.3), usando informações relativas ao URL do serviço, seu nome e a descrição de um arquivo de configuração escrito em XML. O nome do serviço (neste caso, *MyShapeListService*) é usado para gerar o nome da classe utilizada no programa cliente para acessá-lo. Isto é, *MyShapeListService_Impl*.

Contêiner de servlet • A implementação do serviço é executada como um *servlet* dentro de um *contêiner de servlet*, cuja função é carregar, inicializar e executar *servlets*. O contêiner de *servlet* inclui um despachante (*dispatcher*) e esqueletos (veja a Seção 5.4.2). Quando chega uma requisição, o despachante faz seu mapeamento para um esqueleto

em particular, o qual a transforma em código Java e passa para o método apropriado no *servlet*, o qual executa a requisição e produz uma resposta, que o esqueleto transforma de volta em uma resposta SOAP. O URL de um serviço consiste em uma concatenação do URL do contêiner de *servlet* e da categoria e do nome do serviço, por exemplo, <http://localhost:8080/ShapeList-jaxrpc/ShapeList>.

O Tomcat [jakarta.apache.org] é um contêiner de *servlet* comumente usado. Quando o Tomcat está em execução, sua interface de gerenciamento está disponível em um URL para visualização com um navegador. Essa interface mostra os nomes dos *servlets* que estão correntemente distribuídos e fornece operações para gerenciá-los e para acessar informações sobre cada um, incluindo sua descrição de serviço. Uma vez que um *servlet* seja distribuído no Tomcat, os clientes podem acessá-lo, e os efeitos combinados de suas operações serão armazenados em suas variáveis de instância. Em nosso exemplo, uma lista de objetos *GraphicalObjects* será construída, à medida que cada um deles for adicionado, como resultado de uma requisição cliente para a operação *newShape*. Se um *servlet* for interrompido pela interface de gerenciamento do Tomcat, os valores das variáveis de instância serão reconfigurados quando ele for reiniciado.

O Tomcat também dá acesso a uma descrição de cada um dos serviços que contém, para permitir que os programadores projetem programas clientes e para facilitar a compilação automática dos *proxies* exigidos pelo código do cliente. A descrição do serviço é legível para seres humanos, pois é expressa em notação XML (mais especificamente, em WSDL, conforme apresentado na Seção 9.3).

Note que é possível desenvolver serviços Web sem usar contêineres de *servlet*; por exemplo, o Apache Axis oculta do programador esse nível de detalhe.

O programa cliente • O programa cliente pode usar *proxies* estáticos, *proxies* dinâmicos ou uma interface de invocação dinâmica. Em todos os casos, a descrição do serviço relevante pode ser usada para obter as informações exigidas pelo código do cliente. Em nosso exemplo, a descrição do serviço pode ser obtida do Tomcat.

Proxies estáticos: a Figura 9.9 mostra o cliente *ShapeList* fazendo uma chamada por meio de um *proxy* – um objeto local que passa mensagens para o serviço remoto. O código do *proxy* é gerado por *wscompile* a partir da descrição do serviço. O nome da classe do *proxy* é formado pela adição de *_Impl* ao nome do serviço – neste caso, a classe do *proxy* é chamada de *MyShapeListService_Impl*. Esse nome é específico da implementação, pois a especificação do SOAP não fornece uma regra para atribuição de nomes de classes de *proxy*.

Na linha 1, o método *createProxy* é chamado. Esse método aparece na linha 5, onde vai para a linha 6 para criar um *proxy*, usando a classe *MyShapeListService_Impl*; em seguida, ele retorna o *proxy* (note que às vezes os *proxies* são chamados de *stubs*, daí o nome da classe *Stub*). Na linha 2, o URL do serviço é fornecido para o *proxy* por intermédio do argumento dado na linha de comando. Na linha 3, o tipo do *proxy* é ajustado para estar de acordo com o tipo da interface – *ShapeList*. A linha 4 faz uma chamada para o procedimento remoto *getAllState*, solicitando ao serviço para que retorne o objeto que está no elemento zero no vetor de *GraphicalObjects*.

Como o *proxy* é criado no momento da compilação, ele é chamado de *proxy* estático. A descrição do serviço a partir da qual ele foi gerado não terá sido necessariamente gerada a partir de uma interface Java, mas pode ter sido feita por qualquer uma das várias

```

package staticstub;
import javax.xml.rpc.Stub;
public class ShapeListClient {
    public static void main(String[] args) /* passa URL de serviço */
        try {
            Stub proxy = createProxy();
            proxy._setProperty(
                javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, args[0]);
            ShapeList aShapeList = (ShapeList)proxy;
            GraphicalObject g = aShapeList.getAllState(0);
        } catch (Exception ex) { ex.printStackTrace(); }
    }

    private static Stub createProxy() {
        return
            (Stub) (new MyShapeListService_Impl().getShapeListPort());
    }
}

```

Figura 9.9 Implementação em Java do cliente *ShapeList*.

ferramentas associadas a uma variedade de linguagem diferentes. Ela pode até ter sido escrita diretamente em XML.

Proxies dinâmicos: em vez de usar um *proxy* estático previamente compilado, o cliente usa um *proxy* dinâmico cuja classe é criada em tempo de execução a partir das informações presentes na descrição do serviço e na interface do serviço. Esse método evita a necessidade de usar, na implementação, um nome específico para a classe *proxy*.

Interface de invocação dinâmica: isso permite que um cliente chame um procedimento remoto mesmo que sua assinatura ou o nome do serviço seja desconhecido até o momento da execução. Em contraste com as alternativas anteriores, o cliente não exige um *proxy*. Em vez disso, ele precisa usar uma série de operações para configurar o nome da operação do servidor, o valor de retorno e cada um dos parâmetros antes de fazer a chamada de procedimento.

Implementação de SOAP Java • A maneira como a API Java é implementada pode ser explicada com referência à Figura 5.15. Os parágrafos a seguir explicam as funções dos vários componentes presentes em um ambiente Java/SOAP – as interações entre os componentes são iguais às de antes. Não existe nenhum módulo de referência remota.

Módulos de comunicação: as tarefas desses módulos são executadas por dois módulos HTTP. O módulo HTTP no servidor seleciona o despachante de acordo com o URL dado no cabeçalho *Action* da requisição POST.

Proxy cliente: um método de *proxy* (ou *stub*) conhece o URL do serviço e empacota seu próprio nome de método e seus argumentos, junto a uma referência para o esquema XML do serviço, em um envelope de requisição SOAP. Desempacotar a resposta consiste em analisar um envelope SOAP para extrair os resultados, o valor de retorno ou o relatório de falha. A chamada de método de requisição do cliente é enviada para o serviço como uma requisição HTTP.

Despachante e esqueleto: conforme mencionado anteriormente, o despachante e os esqueletos ficam no contêiner de *servlet*. O despachante extrai o nome da operação do cabeçalho *Action* na requisição HTTP e invoca o método correspondente no esqueleto apropriado, passando a ele o envelope SOAP. Um método de esqueleto executa as seguintes tarefas: ele analisa o envelope SOAP presente na mensagem de requisição e extraí seus argumentos, chama o método correspondente e monta um envelope de resposta SOAP contendo os resultados.

Erros, falhas e correção no SOAP/XML: falhas podem ser relatadas pelo módulo HTTP, pelo despachante, pelo esqueleto ou pelo próprio serviço. O serviço pode relatar seus erros por meio de um valor de retorno ou de parâmetros de falha especificados na descrição do serviço. O esqueleto é responsável por verificar se o envelope SOAP contém uma requisição e se o código XML no qual ele está escrito é bem formado. Tendo estabelecido que o código XML é bem formado, o esqueleto usará o espaço de nomes XML presente no envelope para verificar se a requisição corresponde ao serviço oferecido e se a operação e seus argumentos são apropriados. Se a validação da requisição falhar em um desses níveis, um erro será retornado para o cliente. Verificações semelhantes são feitas pelo *proxy*, quando ele recebe o envelope SOAP contendo o resultado.

9.2.4 Comparação entre serviços Web e CORBA

A principal diferença entre serviços Web e o CORBA, ou outro *middleware* semelhante, é o contexto no qual eles se destinam a serem usados. O CORBA foi projetado para uso dentro de uma única organização, ou entre um pequeno número de organizações colaboradoras. Isso resultou em certos aspectos do projeto sendo centralizados demais para uso cooperativo por parte de organizações independentes ou para uso *ad hoc* sem arranjos anteriores, conforme será explicado a partir de agora.

Problemas de atribuição de nomes • No CORBA, cada objeto remoto é referenciado por meio de um nome, que é gerenciado por uma instância do serviço de atribuição de nomes CORBA (Seção 8.3.5). Esse serviço, assim como o DNS, fornece um mapeamento de um nome para um valor, a ser usado como endereço (um IOR, no CORBA). No entanto, ao contrário do DNS, o serviço de atribuição de nomes CORBA é projetado para uso dentro de uma organização, em vez de ser usado por toda a Internet.

No serviço de atribuição de nomes CORBA, cada servidor gerencia um grafo de nomes com um contexto de atribuição de nomes inicial e é inicialmente independente de quaisquer outros servidores. Embora organizações separadas possam confederar (unir) seus serviços de atribuição de nomes, isso não é automático. Antes que um servidor possa se unir com outro, ele precisa conhecer o contexto de atribuição de nomes inicial deste. Assim, o projeto do serviço de atribuição de nomes CORBA restringe efetivamente o compartilhamento de objetos CORBA dentro de um pequeno conjunto de organizações que tenham confederado seus serviços de atribuição de nomes.

Problemas de referência • Agora, consideraremos se uma referência de objeto remoto CORBA, que é chamada de IOR (Seção 8.3.3), poderia ser usada como uma referência de objeto em nível de Internet, da mesma maneira que um URL. Cada IOR contém um campo que especifica o identificador de tipo da interface do objeto que referencia. Entretanto, esse identificador de tipo é entendido apenas pelo repositório de interfaces que armazena a definição do tipo correspondente. Isso tem a implicação de que o cliente e o servidor precisam usar o mesmo repositório de interfaces, o que na realidade não é prático em escala global.

Em contraste, no modelo de serviços Web, um serviço é identificado por meio de um URL, permitindo que um cliente em qualquer parte da Internet faça uma requisição para um serviço que pode pertencer a qualquer organização de qualquer lugar. Isto é, um serviço Web pode ser compartilhado por clientes de toda a Internet. O DNS é o único serviço exigido para acesso de URL – e é projetado para funcionar eficientemente no âmbito da Internet.

Separação de ativação e localização • As tarefas de localizar e ativar serviços Web são bem separadas. Em contraste, uma referência persistente do CORBA se refere a um componente da plataforma (o repositório de implementação) que ativa o objeto correspondente sob demanda em um computador determinado e também é responsável por localizar o objeto quando ele tiver sido ativado.

Facilidade de uso • A infraestrutura HTTP e XML de serviços Web é bem entendida e conveniente para uso e já está instalada em todos os sistemas operacionais comumente usados, embora o usuário exija uma API de linguagem de programação com suporte para SOAP. Em contraste, a plataforma CORBA é um *software* grande e complexo, exigindo instalação e suporte.

Eficiência • O CORBA foi projetado para ser eficiente: o CDR CORBA (Seção 4.3.1) é binário, enquanto a XML é textual. Um estudo realizado por Olson e Ogbuji [2002] compara o desempenho do CORBA com o do SOAP e da XML-RPC. Eles descobriram que as mensagens de requisição SOAP são 14 vezes maiores do que as equivalentes em CORBA, e que uma requisição SOAP demora 882 vezes mais que uma invocação CORBA equivalente. Embora o desempenho relativo dependa da linguagem utilizada e da implementação do *middleware* específica empregada, esse exemplo fornece uma indicação da sobrecarga em potencial das estratégias baseadas em XML. Para muitas aplicações, contudo, a sobrecarga de mensagem e o pior desempenho do SOAP não são notados e seus efeitos se tornam menos óbvios pela disponibilidade de largura de banda, processadores, memória e espaço em disco mais baratos.

O W3C e outros investigaram a possibilidade de permitir a inclusão de dados binários em elementos da XML para aumentar a eficiência. Discussões sobre esse assunto podem ser encontradas no endereço [www.w3.org XXI](http://www.w3.org/XXI) e [www.w3.org XXII](http://www.w3.org/XXII). Note que a XML já fornece representações em hexadecimal e base64 de dados binários. A representação em base64 é usada em conjunto com a criptografia na XML (veja a Seção 9.5). Existe uma sobrecarga de tempo e espaço considerável quando dados binários são convertidos para base64 ou hexadecimal. Portanto, o que é realmente necessário é poder incluir uma representação em binário de uma sequência de itens de dados previamente analisada, como, por exemplo, a produzida pelo CDR CORBA ou por *gzip*. Outra estratégia, que também está sendo investigada, é pegar uma mensagem SOAP, junto a anexos, alguns dos quais podem ser binários, e usar um documento MIME de várias partes para transportá-la.

As vantagens do CORBA • A disponibilidade de serviços CORBA para transações, controle de concorrência, segurança e controle de acesso, eventos e objetos persistentes o torna uma escolha desejável em muitas aplicações destinadas para uso interno de uma organização ou em um grupo de organizações relacionadas. Em geral, essa é uma boa escolha para as aplicações que exigem interações muito complexas. Além disso, o modelo de objeto distribuído é atraente para o projeto de aplicações complexas e vale o esforço de aprendizado extra, necessário para entender os detalhes do relacionamento entre o objeto modelo CORBA (Seção 8.3) e a linguagem de programação em particular que esteja em uso.

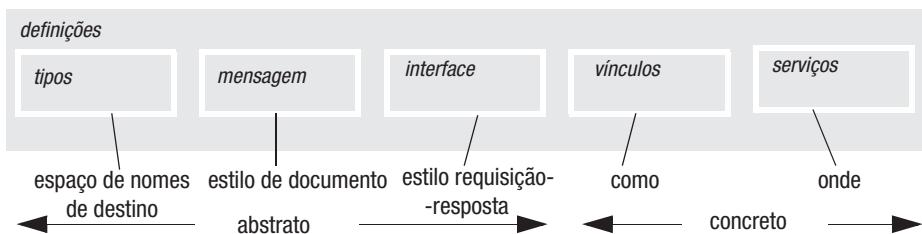


Figura 9.10 Os principais elementos em uma descrição WSDL.

9.3 Descrições de serviço e IDL para serviços Web

As definições de interface são necessárias para permitir que os clientes se comuniquem com os serviços. Para serviços Web, as definições de interface são fornecidas como parte de uma descrição de serviço mais geral, que especifica duas outras características adicionais – como as mensagens devem ser comunicadas (por exemplo, por SOAP com HTTP) e o URI do serviço. Para fornecer serviço em um ambiente com múltiplas linguagens, as descrições de serviço são escritas em XML.

Uma descrição de serviço forma a base de um acordo entre um cliente e um servidor quanto ao serviço oferecido. Ela reúne todos os pontos pertinentes ao serviço que são relevantes para seus clientes. As descrições de serviço geralmente são usadas para gerar *stubs* de cliente que implementam automaticamente o comportamento correto para o cliente.

O componente do tipo IDL de uma descrição de serviço é mais flexível do que as outras IDLs, pois um serviço pode ser especificado em termos dos tipos de mensagens que enviará e receberá, ou em termos das operações que suporta, para permitir a troca de documentos e interações estilo requisição-resposta.

Uma variedade de métodos de comunicação diferentes pode ser usada pelos serviços Web e seus clientes. Portanto, o método de comunicação fica para ser decidido pelo provedor do serviço e é especificado na descrição do serviço, em vez de ser incorporado ao sistema, como no CORBA, por exemplo.

A capacidade de especificar o URI de um serviço como parte de sua descrição evita a necessidade do serviço vinculador ou de separado atribuição de nomes, usado pela maioria dos outros *middlewares*. Isso implica que o URI não pode ser alterado, uma vez que a descrição do serviço tenha se tornado disponível para clientes em potencial. No entanto, o esquema URN aceita a troca de local, permitindo um procedimento indireto no nível da referência.

Em contraste, na estratégia do vinculador, o cliente usa um nome para pesquisar a referência do serviço em tempo de execução, possibilitando que as referências de serviço mudem com o passar do tempo. Porém, essa estratégia exige um procedimento indireto de um nome para uma referência de serviço para todos os serviços, mesmo que muitos deles possam sempre permanecer no mesmo local.

No contexto dos serviços Web, a WSDL (ou Web Services Description Language) é comumente usada para descrições de serviço. A versão atual, a WSDL 2.0 [[www.w3.org XI](http://www.w3.org/XI)], tornou-se a recomendação do W3C em 2007. Ela define um esquema XML para representar os componentes de uma descrição de serviço, os quais incluem, por exemplo, os nomes de elemento *definições*, *tipos*, *mensagens*, *interface*, *vínculos* e *serviços*.

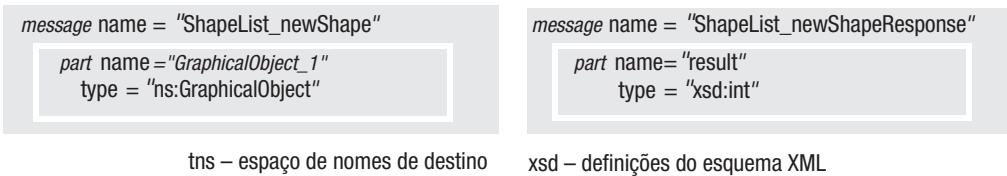


Figura 9.11 Mensagens de requisição-resposta WSDL para a operação *newShape*.

A WSDL separa a parte abstrata de uma descrição de serviço da parte concreta, como se vê na Figura 9.10.

A parte abstrata da descrição inclui um conjunto de definições (*definitions*) dos tipos usados pelo serviço, em particular os tipos dos valores trocados nas mensagens. O exemplo em Java da Seção 9.2.3, cuja interface Java aparece na Figura 9.7, usa os tipos Java *int* e *GraphicalObject*. O primeiro (assim como qualquer tipo básico) pode ser transformado diretamente no equivalente da XML. No entanto, *GraphicalObject* é definido na linguagem Java em termos dos tipos *int*, *String* e *boolean*. *GraphicalObject* é representado na XML, para uso comum por clientes heterogêneos, como *complexType*, consistindo em uma sequência de tipos XML nomeados, incluindo, por exemplo:

```
<element name="isFilled" type="boolean"/>
<element name="originx" type="int"/>
```

O conjunto de nomes definido dentro da seção tipos (*types*) de uma definição WSDL é chamado de *espaço de nomes de destino*. A seção mensagem (*message*) da parte abstrata contém uma descrição do conjunto de mensagens trocadas. Para o estilo documento de interação, essas mensagens serão usadas diretamente. Para o estilo de interação requisição-resposta, existem duas mensagens para cada operação, as quais são usadas para descrever as operações na seção *interface*. A parte concreta especifica como e onde o serviço pode ser contatado.

A modularidade inherente de uma definição WSDL permite que seus componentes sejam combinados de diferentes maneiras; por exemplo, a mesma interface pode ser usada com diferentes vínculos ou localizações. Os tipos podem ser definidos dentro do elemento *types*, ou em um documento separado referenciado por um URI do elemento *types*. Neste último caso, as definições de tipo podem ser referenciadas a partir de vários documentos WSDL diferentes.

Mensagens ou operações • Nos serviços Web, tudo que o cliente e o servidor precisam é ter uma ideia comum sobre a mensagem a ser trocada. Para um serviço baseado na troca de um pequeno número de tipos diferentes de documento, a WSDL descreve apenas os tipos das diferentes mensagens a serem trocadas. Quando um cliente envia uma dessas mensagens para um serviço Web, este decide qual operação vai efetuar e qual tipo de mensagem vai enviar de volta para o cliente, de acordo com o tipo da mensagem que recebeu. Em nosso exemplo em Java, duas mensagens serão definidas para cada uma das operações na interface – uma para a requisição e uma para a resposta. Por exemplo, a Figura 9.11 mostra as mensagens de requisição-resposta da operação de *newShape*, que tem um único argumento de entrada de tipo *GraphicalObject* e um único argumento de saída de tipo *int*.

Contudo, para serviços que suportam várias operações diferentes, é mais eficiente especificar as mensagens trocadas como requisições de operações com argumentos e suas respostas correspondentes, permitindo que o serviço envie cada requisição para a operação apropriada. Entretanto, na WSDL uma operação é uma construção de mensa-

Nome	Mensagens enviadas por			
	Cliente	Servidor	Distribuição	Mensagem de falha
In-Out	Requisição	Resposta		Pode substituir resposta
In-Only	Requisição			Nenhuma
Robust In-Only	Requisição		Garantida	Pode ser enviada
Out-In	Resposta	Requisição		Pode substituir resposta
Out-Only		Requisição		Nenhuma
Robust Out-Only		Requisição	Garantida	Pode ser enviada

Figura 9.12 Padrões de troca de mensagem para operações WSDL.

gens de requisição-resposta relacionadas, em vez da definição de uma operação em uma interface de serviço.

Interface • O conjunto de operações pertencentes a um serviço Web é agrupado em um elemento da XML chamado *interface* (também chamado de *portType*). Cada operação deve especificar o padrão de troca de mensagens entre cliente e servidor. As opções disponíveis incluem as que aparecem na Figura 9.12. A primeira delas, *In-Out*, é a forma requisição-resposta de comunicação cliente-servidor normalmente usada. Nesse padrão, a mensagem de resposta pode ser substituída por uma mensagem de falha. *In-Only* serve para mensagens unilaterais com semântica *maybe* e *Out-Only* serve para mensagens unilaterais do servidor para o cliente; mensagens de falha não podem ser enviadas com nenhuma das duas. *Robust In-Only* e *Robust Out-Only* são as mensagens correspondentes com distribuição garantida; e mensagens de falha podem ser trocadas. *Out-In* é uma interação requisição-resposta iniciada pelo servidor. A WSDL 2.0 também é extensível, pois as organizações podem introduzir seus próprios padrões de troca de mensagem, caso os predefinidos se mostrem inadequados.

Voltando ao nosso exemplo em Java, cada uma das operações é definida de modo a ter um padrão *In-Out*. A operação *newShape* aparece na Figura 9.13, usando as mensagens definidas na Figura 9.11. Essa definição, junto às definições das quatro outras operações, é incluída em um elemento *interface* da XML. Uma operação também pode especificar as mensagens de falha que podem ser enviadas.

Contudo, se, por exemplo, uma operação tem dois argumentos, digamos, um inteiro e um *string*, então não há necessidade de definir um novo tipo de dados, pois esses tipos são definidos para esquemas XML. Entretanto, será necessário definir uma mensagem

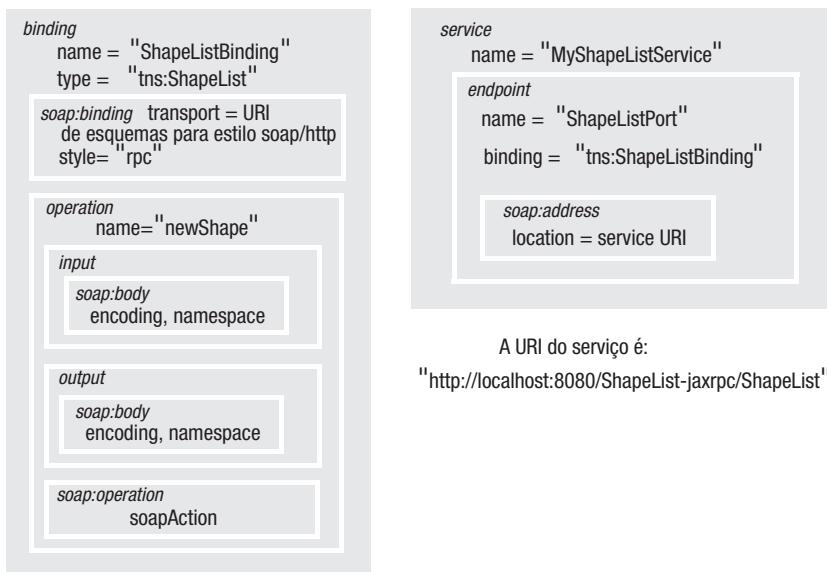
```

operation name = "newShape"
  pattern = In-Out
    input message = "tns:ShapeList_newShape"
    output message = "tns:Shape List_newShapeResponse"
  
```

tns – espaço de nomes de destino xsd – definições do esquema XML

Os nomes *operation*, *pattern*, *input* e *output* são definidos no esquema XML da WSDL

Figura 9.13 Operação *newShape* da WSDL.



A URI do serviço é:
`"http://localhost:8080/ShapeList-jaxrpc/ShapeList"`

Figura 9.14 Definições de vínculo e serviço SOAP.

que tenha essas duas partes. Essa mensagem pode, então, ser usada como entrada ou saída na definição da operação.

Herança: toda interface WSDL pode estender uma ou mais outras interfaces WSDL. Essa é uma forma simples de herança, na qual uma interface suporta as operações de todas as interfaces que ela estende, além daquelas que ela mesma define. A definição recursiva de interfaces não é permitida; isto é, se a interface B estende a interface A, então a interface A não pode estender a interface B.

Parte concreta • A parte restante (concreta) de um documento WSDL consiste na escolha de protocolos, definido pelo elemento vínculo (*binding*) e na escolha do ponto final ou do servidor por meio do elemento serviço (*service*). As duas estão relacionadas, pois a forma de endereço depende do tipo de protocolo que está sendo usado. Por exemplo, um ponto final SOAP usará um URI, enquanto um ponto final CORBA usaria um identificador de objeto específico do CORBA.

Vínculo: a seção vínculo (*binding*) em um documento WSDL informa quais formatos de mensagem e forma de representação de dados externa devem ser usados. Por exemplo, os serviços Web frequentemente usam SOAP, HTTP e MIME. Os vínculos podem ser associados a operações, ou interfaces, em particular, ou podem ficar livres para uso por uma variedade de diferentes serviços Web.

A Figura 9.14 mostra um exemplo de vínculo, incluindo um *soap:binding*, que especifica o URL de um protocolo para transmitir envelopes SOAP: o vínculo HTTP para SOAP. Os atributos opcionais desse elemento também podem especificar o seguinte:

- o padrão de troca de mensagem, que pode ser *rpc* (requisição-resposta) ou troca de *document* – o valor padrão é *document*;

- o esquema XML dos formatos de mensagem – o padrão é o *envelope* SOAP;
- o esquema XML da representação de dados externa – o padrão é a codificação SOAP da XML.

A Figura 9.14 também mostra os detalhes dos vínculos de uma das operações (*newShape*), especificando que a mensagem de *entrada* e a de *saída* devem ser passadas na seção *soap:body*, usando um estilo de codificação em particular e, além disso, que a operação deve ser transmitida como uma *Action* SOAP.

Serviço: cada elemento serviço (*service*) em um documento WSDL especifica o nome do serviço e um ou mais *pontos finais* (ou portas) onde uma instância do serviço pode ser contatada. Cada um dos elementos ponto-finais (*endpoint*) se refere ao nome do vínculo em uso e, no caso de um vínculo SOAP, utiliza um elemento *soap:address* para especificar o URI da localização do serviço.

Documentação • Informações legíveis para seres humanos e pela máquina podem ser inseridas em um elemento documento (*documentation*) na maioria dos pontos dentro da WSDL. Essas informações podem ser removidas antes que a WSDL seja usada para processamento automático, por exemplo, por compiladores de *stub*.

Uso de WSDL • Documentos WSDL completos podem ser acessados por meio de seus URIs por clientes e servidores, direta ou indiretamente, por intermédio de um serviço de diretório como o UDDI. Estão disponíveis ferramentas para gerar definições WSDL a partir das informações fornecidas por meio de uma interface gráfica com o usuário, eliminando a necessidade de envolvimento dos usuários com os detalhes complexos e com a estrutura da WSDL. Por exemplo, a *Web Services Description Language* do toolkit Java permite a criação, representação e manipulação de documentos WSDL descrevendo serviços [wsdl4j.sourceforge.org]. As definições WSDL também podem ser geradas a partir de definições de interface escritas em outras linguagens, como JAX-RPC Java, discutida na Seção 9.2.2.

9.4 Um serviço de diretório para uso com serviços Web

Existem muitas maneiras pelas quais os clientes podem obter descrições de serviço; por exemplo, qualquer um que forneça um serviço Web de mais alto nível, como o serviço de agente de viagens, discutido na Seção 9.1, quase certamente faria uma página Web anunciando que o serviço e os clientes em potencial se deparariam com a página ao procurar serviços desse tipo.

Entretanto, qualquer organização que pretenda basear suas aplicações em serviços Web achará mais conveniente usar um serviço de diretório para tornar esses serviços disponíveis para os clientes. Esse é o objetivo do serviço UDDI (Universal Description, Discovery and Integration) [Bellwood *et al.* 2003], que fornece um serviço de nome e um serviço de diretório (veja a Seção 13.3). Isto é, as descrições de serviço WSDL podem ser pesquisadas pelo nome (um serviço de catálogo telefônico) ou pelo atributo (um serviço de páginas amarelas). Elas também podem ser acessadas diretamente por meio de seus URLs, o que é conveniente para desenvolvedores que estejam projetando programas clientes que utilizam o serviço.

Os clientes podem usar a estratégia das páginas amarelas para pesquisar uma categoria de serviço em particular, como um agente de viagens ou uma livraria, ou podem

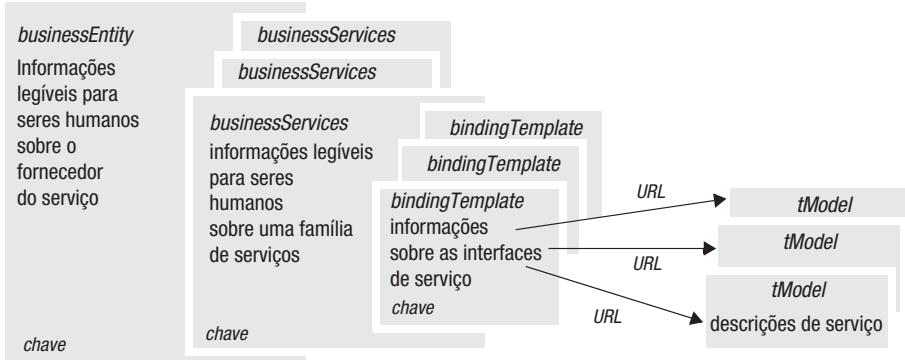


Figura 9.15 As principais estruturas de dados UDDI.

usar a estratégia do catálogo telefônico para pesquisar um serviço com referência à organização que o fornece.

Estruturas de dados • As estruturas de dados que suportam UDDI são projetadas de forma a permitir todos os estilos de acesso anteriores e podem incorporar qualquer volume de informações legíveis para seres humanos. Os dados são organizados em termos das quatro estruturas mostradas na Figura 9.15, cada uma das quais podendo ser acessada individualmente, por meio de um identificador chamado de *chave* (além de *tModel*, que pode ser acessado por um URL):

businessEntity: descreve a organização que fornece esses serviços Web, dando seu nome, endereço, atividades etc.;

businessServices: armazena informações sobre um conjunto de instâncias de um serviço Web, como seu nome e uma descrição de seu propósito; por exemplo, agente de viagens ou livraria;

bindingTemplate: contém o endereço de uma instância de serviço Web e referências para descrições do serviço;

tModel: contém descrições de serviço, normalmente documentos WSDL, armazenadas fora do banco de dados e acessadas por meio de URLs.

Pesquisa (lookup) • O UDDI fornece uma API para pesquisar (*lookup*) serviços com base em dois conjuntos de operações de consulta:

- O conjunto de operações *get_xxx* inclui *get_BusinessDetail*, *get_ServiceDetail*, *get_bindingDetail* e *get_tModelDetail*; elas recuperam uma entidade correspondente a uma dada chave;
- O conjunto de operações *find_xxx* inclui *find_business*, *find_service*, *find_binding* e *find_tModel*; elas recuperam o conjunto de entidades que correspondem a um conjunto de critérios de busca em particular, fornecendo um resumo dos nomes, descrições, chaves e URLs.

Assim, os clientes que estiverem de posse de uma chave em particular podem usar uma operação *get_xxx* para recuperar diretamente a entidade correspondente. Outros clientes podem usar navegação para ajudar nas buscas – começando com um conjunto de resulta-

dos abrangente e reduzindo-o gradualmente. Por exemplo, eles podem começar usando a operação *find_business* para obter uma lista contendo um resumo das informações sobre os provedores correspondentes. A partir desse resumo, o usuário pode usar a operação *find_service* para refinar a busca, fazendo-a corresponder ao tipo de serviço exigido. Nos dois casos, ele encontrará a chave de um *bindingTemplate* conveniente e, com isso, encontrará o URL para recuperar o documento WSDL de um serviço apropriado.

Além disso, o UDDI fornece uma interface de publicação/assinatura, por meio da qual os clientes registram o interesse em um conjunto de entidades em particular, em um registro UDDI, e obtêm notificações sobre alterações de forma síncrona ou assíncrona.

Publicação • O UDDI fornece uma interface para publicar (anunciar) e atualizar informações sobre serviços Web. Na primeira vez que uma estrutura de dados (veja a Figura 9.15) é publicada em um servidor UDDI, ele recebe uma chave na forma de um URI (por exemplo, *uddi:cdk5.net:213*) e esse servidor se torna seu proprietário.

Registros • O serviço UDDI é baseado em dados replicados armazenados em registros. Um registro UDDI consiste em um ou mais servidores UDDI, cada um dos quais tendo uma cópia do mesmo conjunto de dados. Os dados são replicados entre os membros de um registro. Cada um deles pode responder às consultas e publicar informações. As alterações em uma estrutura de dados devem ser enviadas ao seu proprietário – isto é, o servidor em que ela foi publicada pela primeira vez. É possível um proprietário transmitir sua posse para outro servidor UDDI no mesmo registro.

Esquema de replicação: os membros de um registro propagam cópias das estruturas de dados uns para os outros como segue: um servidor que fez alterações notifica os outros servidores no registro, os quais, então, solicitam as alterações. Uma forma de carimbo de tempo vetorial é usada para determinar quais das alterações devem ser propagadas e aplicadas. O esquema é simples, em comparação com outros esquemas de replicação que usam carimbo de tempo vetoriais, como Gossip (Seção 18.4.1) ou Coda (Seção 18.4.3), pois:

1. Todas as alterações em uma estrutura de dados em particular são feitas no mesmo servidor.
2. As atualizações de um servidor em particular são recebidas em ordem sequencial pelos outros membros, mas nenhuma ordenação em particular é imposta entre as operações de atualizações executadas por diferentes servidores.

Interação entre servidores: conforme descrito anteriormente, os servidores interagem uns com os outros para transmitir o esquema de replicação. Eles também podem interagir para transferir a posse de estruturas de dados. Entretanto, a resposta a uma operação de pesquisa é dada por um único servidor, sem nenhuma interação com outros servidores no registro, ao contrário do serviço de diretório X.500 (Seção 13.5), no qual os dados são particionados entre os servidores, que cooperam uns com os outros na busca do servidor relevante para uma requisição em particular.

9.5 Aspectos de segurança em XML

A segurança em XML consiste em um conjunto de projetos relacionados do W3C para assinatura, gerenciamento de chaves e criptografia. Ela se destina a ser usada no trabalho cooperativo pela Internet, envolvendo documentos cujo conteúdo talvez precise ser autenticado ou cifrado. Normalmente, os documentos são criados, trocados, armazenados

e, depois, novamente trocados, possivelmente após serem modificados por uma série de usuários diferentes.

A WS-Security [Kaler 2002] é outra estratégia de segurança relativa à aplicação de integridade, confidencialidade e autenticação de mensagens no SOAP.

Como exemplo de um contexto no qual a segurança em XML seria útil, considere um documento contendo os registros médicos de um paciente. Diferentes partes desse documento são usadas no consultório médico local e em várias clínicas e hospitais especializados visitados pelo paciente. Ele será atualizado por médicos, especialistas e enfermeiras que estejam tomando notas sobre condições e tratamento, por administradores que estejam marcando compromissos e por farmacêuticos para fornecer remédios. Diferentes partes do documento serão visíveis pelas diferentes funções mencionadas anteriormente e, possivelmente, também pelo paciente. É fundamental que certas partes do documento, por exemplo, as recomendações quanto ao tratamento, possam ser atribuídas à pessoa que as fez e possa haver garantias de que não tenham sido alteradas.

Essas necessidades não podem ser atendidas pelo TLS (anteriormente conhecido como SSL, Seção 11.6.3), usado para criar um canal seguro para a comunicação de informações. Ele permite que os processos nos dois extremos do canal negoциem a necessidade de autenticação, de criptografia e as chaves e os algoritmos a serem usados, tanto quando um canal for estabelecido, quanto durante seu tempo de vida. Por exemplo, os dados sobre uma transação financeira poderiam ser assinados e enviados à vontade, até que informações sigilosas, como detalhes do cartão de crédito, fossem fornecidas, no ponto em que a criptografia seria aplicada.

Para possibilitar o novo tipo de utilização mencionado anteriormente, a segurança deve ser especificada e aplicada no próprio documento, em vez de ser uma propriedade do canal que o transmitirá de um usuário para outro.

Isso é possível em XML, ou em outros formatos de documento estruturados, nos quais podem ser usados metadados. Marcas (*tags*) XML podem ser usadas para definir as propriedades dos dados no documento. Em particular, a segurança em XML depende de novas *tags* que possam ser usadas para indicar o início e o fim de seções de dados cifrados, ou assinados, e de assinaturas. Uma vez que a segurança necessária tenha sido aplicada dentro de um documento, ele pode ser enviado para uma variedade de usuários diferentes, até por meio de *multicast*.

Requisitos básicos • A segurança em XML deve fornecer pelo menos o mesmo nível de proteção do TLS. Isto é:

Ser capaz de cifrar um documento inteiro ou apenas algumas partes selecionadas dele: por exemplo, considere as informações sobre uma transação financeira, as quais incluem o nome de uma pessoa, o tipo de transação e os detalhes sobre o cartão de crédito ou débito que está sendo usado. Em um caso, apenas os detalhes do cartão poderiam ser ocultos, tornando possível identificar a transação antes de decifrar o registro. No outro caso, o tipo de transação também poderia ser oculto, para que intrusos não pudessem identificar, por exemplo, se é um pedido ou um pagamento.

Ser capaz de assinar um documento inteiro ou apenas algumas partes selecionadas dele: quando um documento se destina a ser usado para trabalho cooperativo por um grupo de pessoas, podem existir algumas partes críticas do documento que devem ser assinadas para garantir que foram feitas por uma pessoa em particular

ou que não foram alteradas. Contudo, também é útil poder ter outras partes que possam ser alteradas durante o uso do documento – essas não devem ser assinadas.

Requisitos básicos adicionais • Mais requisitos surgem da necessidade de armazenar documentos, possivelmente para modificá-los e enviá-los para uma variedade de destinatários diferentes:

Fazer adições a um documento assinado e assinar o resultado final: por exemplo, Alice pode assinar um documento e passá-lo para Bob, que “atesta a assinatura dela” adicionando uma observação nesse sentido e depois assina o documento inteiro. (A Seção 11.1 apresenta os nomes, incluindo Alice e Bob, usados como protagonistas nos protocolos de segurança.)

Autorizar vários usuários diferentes a ver partes distintas de um documento: no caso de um registro médico, um pesquisador pode ver uma seção em particular dos dados médicos, um administrador pode ver detalhes pessoais e um médico pode ver ambos.

Fazer adições em um documento que já contém seções cifradas e cifrar parte da nova versão, possivelmente incluindo algumas das seções já cifradas.

A flexibilidade e os recursos de estruturação da notação XML tornam possível fazer tudo o que foi descrito anteriormente, sem quaisquer adições no esquema derivado dos requisitos básicos.

Requisitos relativos aos algoritmos • Os documentos XML seguros são assinados e/ou cifrados bem antes de qualquer consideração com relação a quem os acessará. Se o criador não estiver mais envolvido, não será possível negociar os protocolos e o uso de autenticação ou criptografia. Portanto:

O padrão deve especificar um conjunto de algoritmos a serem fornecidos em qualquer implementação de segurança em XML: pelo menos um algoritmo de criptografia e um de assinatura devem ser obrigatórios para permitir a maior interoperabilidade possível. Outros algoritmos opcionais devem ser fornecidos para uso dentro de grupos menores.

Os algoritmos usados para criptografia e autenticação de um documento em particular devem ser selecionados a partir desse conjunto, e os nomes dos algoritmos em uso devem ser referenciados dentro do próprio documento XML: se os lugares onde o documento será usado não podem ser previstos, então um dos protocolos exigidos deve ser usado.

A segurança em XML define os nomes dos elementos que podem ser usados para especificar o URI do algoritmo em uso para assinatura ou criptografia. Portanto, para ser capaz de selecionar uma variedade de algoritmos dentro do mesmo documento XML, geralmente um elemento especificando um algoritmo é aninhado dentro de um elemento que contém informações assinadas ou cifradas.

Requisitos para localização de chaves • Quando um documento é criado, e sempre que ele for atualizado, as chaves apropriadas devem ser escolhidas, sem qualquer negociação com as partes que poderão acessar o documento no futuro. Isso leva aos seguintes requisitos:

Ajudar os usuários de documentos seguros na localização das chaves necessárias: por exemplo, um documento que inclui dados assinados deve conter informações quanto a chave pública a ser usada para validar a assinatura, como um nome

<i>Tipo de algoritmo</i>	<i>Nome do algoritmo</i>	<i>Exigido</i>	<i>referência</i>
Resumo de mensagem	SHA-1	Exigido	Seção 11.4.3
Codificação	base64	Exigido	[Freed e Borenstein 1996]
Assinatura (assimétrica)	DSA com SHA-1	Exigido	[NIST 1994]
Assinatura MAC (simétrica)	RSA com SHA-1	Recomendado	Seção 11.3.2
Canonização	HMAC-SHA-1	Exigido	Seção 11.4.2 e Krawczyk <i>et al.</i> [1997]
	XML canônica	Exigido	Página 409

Figura 9.16 Algoritmos exigidos para assinatura XML.

que possa ser usado para obter a chave, ou um certificado digital. Um elemento *KeyInfo* pode ser usado para esse propósito.

Tornar possível a usuários em cooperação ajudarem uns aos outros com chaves: desde que o próprio elemento *KeyInfo* não seja vinculado à assinatura por meio de criptografia, as informações podem ser adicionadas sem violar a assinatura digital. Por exemplo, Alice assina um documento e o envia para Bob com um elemento *KeyInfo* especificando apenas o nome da chave. Quando Bob recebe o documento, ele recupera as informações necessárias para validar a assinatura e adiciona isso no elemento *KeyInfo* ao passar o documento para Carol.

O elemento KeyInfo • A segurança em XML especifica um elemento *KeyInfo* para indicar a chave a ser usada para validar uma assinatura ou para decifrar alguns dados. Ele pode conter, por exemplo, certificados digitais, os nomes de chaves ou algoritmos de acordo de chave. Seu uso é opcional: o assinante talvez não queira revelar nenhuma informação sobre chave para todas as pessoas que acessam o documento e, em alguns casos, a aplicação que está usando a segurança em XML já pode ter acesso às chaves em uso.

XML canônica • Algumas aplicações podem fazer alterações que não têm efeito sobre o conteúdo das informações de um documento XML. Isso acontece porque há uma variedade de maneiras de representar o que é logicamente o mesmo documento XML. Por exemplo, os atributos podem estar em ordens diferentes e diferentes codificações de caractere podem ser usadas, embora o conteúdo da informação seja equivalente. A XML canônica [www.w3.org X] foi projetada para uso com assinaturas digitais, as quais são usadas para garantir que o conteúdo das informações de um documento não tenha sido alterado. Os elementos da XML se tornam canônicos antes de serem assinados e o nome do algoritmo de canonização é armazenado junto com a assinatura. Isso permite que o mesmo algoritmo seja usado quando a assinatura é validada.

A forma canônica é um padrão da XML através de uma disposição em série, como um fluxo de bytes. Ela adiciona atributos padrão e remove esquemas supérfluos, colocando os atributos e as declarações de esquema na ordem lexicográfica em cada elemento. Ela usa uma forma padrão para quebras de linha e a codificação UTF-8 para caracteres. Quaisquer dois documentos XML equivalentes têm a mesma forma canônica.

Quando um subconjunto de um documento XML, digamos, um elemento, torna-se canônico, a forma canônica inclui o contexto ancestral, isto é, os espaços de nomes declarados e os valores dos atributos. Assim, quando a XML canônica for usada em conjunto com assinaturas digitais, a assinatura de um elemento não passará em sua validação, caso esse elemento seja colocado em um contexto diferente.

<i>Tipo de algoritmo</i>	<i>Nome do algoritmo</i>	<i>Exigido</i>	<i>referência</i>
Cifra de bloco	TRIPLEDES, AES-128,	exigido	Seção 11.3.1
	AES-256		
	AES-192	opcional	
Codificação	base64	exigido	[Freed e Borenstein 1996]
Transporte de chave	RSA-v1.5,	exigido	Seção 11.3.2
	RSA-OAEP		[Kaliski e Staddon 1998]
Envoltório de chave simétrica (assinatura por chave compartilhada)	TRIPLEDES	exigido	[Housley 2002]
	KeyWrap, AES-128		
	KeyWrap, AES- -256KeyWrap		
Acordo de chave	AES-192 KeyWrap	opcional	
	Diffie-Hellman	opcional	[Rescorla, 1999]

Figura 9.17 Algoritmos exigidos para criptografia (os algoritmos da Figura 9.16 também são exigidos).

Uma variação desse algoritmo, chamado Exclusive Canonical XML, omite o contexto da disposição em série. Isso poderia ser usado se a aplicação se destinasse a um elemento assinado em particular para ser usado em diferentes contextos.

Uso de assinaturas digitais em XML • A especificação para assinaturas digitais em XML [[www.w3.org XII](http://www.w3.org/XII)] é uma recomendação do W3C que define novos tipos de elemento XML para conter as assinaturas, os nomes dos algoritmos, as chaves e as referências para informações assinadas. Os nomes fornecidos nessa especificação são definidos no esquema de assinatura da XML, o qual inclui os elementos *Signature*, *SignatureValue*, *SignedInfo* e *KeyInfo*. A Figura 9.16 mostra os algoritmos que devem estar disponíveis em uma implementação de assinatura XML.

Serviço de gerenciamento de chaves • A especificação do serviço de gerenciamento de chaves em XML [[www.w3.org XIII](http://www.w3.org/XIII)] contém protocolos para distribuir e registrar chaves públicas para uso em assinaturas XML. Embora não exija nenhuma infraestrutura de chave pública em particular, o serviço é projetado para ser compatível com as já existentes, por exemplo, certificados X.509 (Seção 11.4.4), SPKI (infraestrutura de chave pública simples, Seção 11.4.4) ou identificadores de chave PGP (Pretty Good Privacy, Seção 11.5.2).

Os clientes podem usar esse serviço para encontrar a chave pública de uma pessoa. Por exemplo, se Alice quiser enviar um *e-mail* cifrado para Bob, ela pode usar esse serviço para obter a chave pública dele. Em outro exemplo, Bob recebe um documento assinado de Alice, contendo o certificado X.509 dela e, então, pede ao serviço de informação de chave para que extraia a chave pública.

Criptografia XML • O padrão para criptografia em XML está definido na recomendação do W3C, que especifica a maneira de representar dados cifrados em XML e o processo para cífrá-los e decífrá-los [[www.w3.org XIV](http://www.w3.org/XIV)]. Ele apresenta um elemento *EncryptedData* para englobar partes de dados cifrados.

A Figura 9.17 especifica os algoritmos de criptografia que devem ser incluídos em uma implementação de criptografia em XML. Algoritmos de cifra de bloco são usados para cifrar os dados, e codificação base64 é usada em XML para representar

1. O cliente solicita ao serviço de agente de viagens informações sobre um conjunto de serviços; por exemplo, voos, aluguel de carro e reservas em hotel.
2. O serviço de agente de viagens reúne as informações sobre preço e disponibilidade e as envia para o cliente, o qual escolhe uma das opções a seguir em nome do usuário:
 - (a) refinar a consulta, possivelmente envolvendo mais provedores para obter mais informações; e, em seguida, repetir o passo 2;
 - (b) fazer reservas;
 - (c) sair.
3. O cliente solicita uma reserva e o serviço de agente de viagens verifica a disponibilidade.
4. *Ou* todos estão disponíveis;
 - Ou* para os serviços que não estão disponíveis;
 - Ou* são oferecidas alternativas para o cliente, que volta para o passo 3;
 - Ou* o cliente volta para o passo 1.
5. Faz o depósito.
6. Fornece ao cliente um número de reserva como confirmação.
7. Durante o período decorrido até o pagamento final, o cliente pode modificar ou cancelar as reservas.

Figura 9.18 Cenário do agente de viagens.

assinaturas digitais e dados cifrados. Algoritmos de transporte de chave são algoritmos de criptografia de chave pública projetados para uso na criptografia e decifração das chaves em si.

Os algoritmos de envoltório de chave simétrica são algoritmos de criptografia de chave secreta compartilhada, projetados para cifrar e decifrar chaves simétricas por meio de outra chave. Isso poderia ser usado se uma chave precisasse ser incluída em um elemento *KeyInfo*.

Um algoritmo de acordo de chave permite que uma chave secreta compartilhada seja extraída a partir de um cálculo em duas chaves públicas. Esse algoritmo se tornou disponível para uso por aplicações que precisam concordar com uma chave compartilhada sem nenhuma troca. Ele não é aplicado pelo sistema de segurança da XML em si.

9.6 Coordenação de serviços Web

A infraestrutura SOAP suporta interações requisição-resposta entre clientes e serviços Web. Entretanto, muitas aplicações úteis envolvem várias requisições que precisam ser executadas em uma ordem em particular. Por exemplo, ao se fazer reservas para um voo, são reunidas as informações sobre preço e disponibilidade, antes que as reservas sejam feitas. Quando um usuário interage com páginas Web por intermédio de um navegador, por exemplo, para fazer reserva em um voo, ou para dar um lance em um leilão, a interface fornecida pelo navegador, que é baseada nas informações fornecidas pelo servidor, controla a sequência em que as operações são executadas.

Entretanto, se for um serviço Web que estiver fazendo reservas, como o serviço de agente de viagens mostrado na Figura 9.2, esse serviço precisará trabalhar a partir de

uma descrição da maneira apropriada para prosseguir ao interagir com outros serviços que realizam, por exemplo, aluguel de carros e reservas em hotel, assim como reservas em voos. A Figura 9.18 mostra um exemplo de tal descrição.

Esse exemplo ilustra a necessidade de serviços Web, como clientes, de receberem uma descrição de um protocolo em particular a ser seguido ao interagir com outros serviços Web. Contudo, também existe o problema da manutenção da consistência nos dados do servidor quando ele está recebendo e respondendo a requisições de vários clientes. Os Capítulos 16 e 17 discutirão as transações, ilustrando os problemas por meio de uma série de transações bancárias. Como um exemplo simples, em uma transferência de dinheiro entre duas contas bancárias, a consistência exige que o depósito em uma conta e o saque da outra devam ser realizados. O Capítulo 17 apresentará o protocolo de *commit* (confirmação) em duas fases, que é usado por servidores em cooperação para garantir a consistência de transações.

Em alguns casos, transações atômicas são adequadas aos requisitos de aplicações que usam serviços Web. Entretanto, atividades como aquelas do agente de viagens demoram um longo tempo para terminar e seria impraticável usar um protocolo de *commit* em duas fases para executá-las, pois envolve manter recursos bloqueados por longos períodos de tempo. Uma alternativa é usar um protocolo mais relaxado, no qual cada participante faz alterações no estado persistente, quando elas ocorrem. No caso de falha, um protocolo em nível de aplicação é usado para desfazer essas ações.

No *middleware* convencional, a infraestrutura fornece um protocolo de requisição-resposta simples, deixando outros serviços, como transações, persistência e segurança, serem implementados como serviços separados de mais alto nível, que podem ser usados quando forem necessários. O mesmo vale para serviços Web, em que o W3C e outros vêm trabalhando na definição de serviços de mais alto nível.

Tem-se trabalhado em um modelo geral para a coordenação de serviços Web, o qual é semelhante ao modelo de transação distribuída descrito na Seção 17.2, pois tem funções de coordenador e participante que são capazes de atuar em protocolos específicos, por exemplo, para executar uma transação distribuída. Esse trabalho, que é chamado WS-Coordination, é descrito por Langworthy [2004]. O mesmo grupo também mostrou como transações podem ser executadas dentro desse modelo. Para um estudo amplo sobre protocolos de coordenação de serviços Web, consulte Alonso *et al.* [2004].

No restante desta seção, descreveremos, em linhas gerais, as ideias ligadas à coreografia de serviço Web. Considere o fato de que seria possível descrever todos os caminhos alternativos válidos possíveis pelo conjunto de interações entre pares de serviços Web, trabalhando em uma tarefa conjunta, como o cenário do agente de viagens. Se tal descrição estivesse disponível, ela poderia ser usada como auxiliar na coordenação de tarefas conjuntas. Ela também poderia ser usada como uma especificação a ser seguida por novas instâncias de um serviço, como um novo serviço de reservas de voo que quisesse se juntar a uma colaboração.

O W3C usa o termo *coreografia* para se referir a uma linguagem baseada na WSDL para definir a coordenação. Por exemplo, a linguagem poderia especificar restrições sobre a ordem e as condições em que as mensagens são trocadas pelos participantes. Uma coreografia se destina a fornecer uma descrição global de um conjunto de interações, mostrando o comportamento de cada membro de um conjunto de participantes, visando melhorar a interoperabilidade.

Requisitos da coreografia • A coreografia se destina a suportar interações entre serviços Web que geralmente são gerenciados por diferentes empresas e organizações. Uma colaboração envolvendo vários serviços Web e clientes deve ser descrita em termos dos conjuntos de interações observáveis entre pares delas. Tal descrição poderia ser vista como um contrato entre os participantes. Ela poderia ser usada da seguinte forma:

- para gerar esboços de código para um novo serviço que quisesse participar;
- como base para gerar mensagens de teste para um novo serviço;
- para promover um entendimento comum da colaboração;
- para analisar a colaboração, por exemplo, para identificar possíveis situações de impasse.

O uso de uma descrição de coreografia comum por um conjunto de serviços Web em colaboração deve resultar em serviços mais robustos, com melhor interoperabilidade. Além disso, deve ser mais fácil desenvolver e introduzir novos serviços, tornando o serviço global mais útil.

O documento da minuta do trabalho do W3C, no endereço [www.w3.org XV], sugere que uma linguagem de coreografia deve incluir as seguintes características:

- composição hierárquica e recursiva das coreografias;
- a capacidade de adicionar novas instâncias de um serviço existente e novos serviços;
- caminhos concorrentes, caminhos alternativos e a capacidade de repetir uma seção de uma coreografia;
- tempos limites variáveis – por exemplo, diferentes períodos para manter reservas;
- exceções, por exemplo, para tratar das mensagens que chegam fora de sequência e ações do usuário, como cancelamentos;
- interações assíncronas (*callbacks*);
- passagem de referência, por exemplo, para permitir que uma operadora de aluguel de carros consulte um banco para uma verificação de crédito em nome de um usuário;
- marcação dos limites das transações separadas que ocorrem, por exemplo, para permitir recuperação;
- a capacidade de incluir documentação legível para seres humanos.

Um modelo baseado nesses requisitos está descrito em outro documento de minuta de trabalho do W3C [www.w3.org XVI].

Linguagens de coreografia • A intenção é produzir uma linguagem declarativa, baseada em XML, para definir coreografias que possam usar definições WSDL. O W3C elaborou uma recomendação para *Web Services Choreography Definition Language Version 1* [www.w3.org XVII]. Antes disso, um grupo de empresas enviou para o W3C uma especificação para a interface de coreografia de serviços Web [www.w3.org XVIII].

9.7 Aplicações de serviços Web

Atualmente, os serviços Web representam o paradigma dominante para programação de sistemas distribuídos. Nesta seção, discutiremos diversas áreas importantes em que os serviços Web têm sido extensivamente empregados: no suporte para arquitetura orientada a serviços, em grade (*grid*) e, recentemente, na computação em nuvem.

9.7.1 Arquitetura orientada a serviços

A *arquitetura orientada a serviços* (*SOA, Service-Oriented Architecture*) é um conjunto de princípios de projeto por meio do qual os sistemas distribuídos são desenvolvidos usando-se conjuntos de serviços pouco acoplados que podem ser descobertos dinamicamente e, então, comunicar-se uns com os outros, ou que são coordenados por meio de coreografia para fornecer serviços aprimorados. O modelo de arquitetura orientada a serviços é um conceito abstrato que pode ser implementado usando-se uma variedade de tecnologias, incluindo as estratégias de objeto distribuído ou componente, discutidas no Capítulo 8. Contudo, a principal maneira de concretizar a arquitetura orientada a serviços é por meio do uso de serviços Web, em grande medida devido ao baixo acoplamento inerente a essa estratégia (conforme discutido na Seção 9.2).

Esse estilo de arquitetura pode ser usado dentro de uma empresa ou organização para oferecer uma arquitetura de *software* flexível e obter interoperabilidade entre os vários serviços. No entanto, seu uso mais importante é na Internet, oferecendo uma visão comum dos serviços, tornando-os globalmente acessíveis e receptivos à composição subsequente. Isso torna possível transcender os níveis de heterogeneidade inerentes à Internet e também lidar com o problema de diferentes organizações adotarem internamente diferentes produtos de *middleware* – é possível que uma organização use CORBA e outra use .NET, mas que então ambas exponham interfaces usando serviços Web, estimulando, assim, a interoperabilidade global. A propriedade resultante é conhecida como *integração de empresa para empresa (B2B, business-to-business)*. Já vimos um exemplo de necessidade da integração B2B, na Figura 9.18 (o cenário do agente de viagens), em que o agente pode lidar com uma ampla variedade de empresas que oferecem voos, carros e acomodação em hotel.

A arquitetura orientada a serviços também possibilita e estimula uma estratégia de *mashup* para desenvolvimento de *software*. *Mashup* é um novo serviço criado por um desenvolvedor externo, que combina dois ou mais serviços disponíveis no ambiente distribuído. A cultura do *mashup* conta com a pronta disponibilidade de serviços úteis, com interfaces bem definidas, complementados com uma comunidade de inovação aberta na qual indivíduos ou grupos se envolvem no desenvolvimento de serviços combinados experimentais e os tornam disponíveis para outros, para mais desenvolvimento. Agora as duas condições são satisfeitas pela Internet, particularmente com o surgimento da computação em nuvem como um serviço (conforme apresentado na Seção 7.7.1), na qual importantes desenvolvedores de *software*, como Amazon, Flickr e eBay, tornam serviços disponíveis para outros desenvolvedores por meio de interfaces publicadas. Como exemplo, consulte JBidwatcher [www.jbidwatcher.org], um *mashup* baseado em Java que faz interface com o eBay para gerenciar ofertas de maneira mais proativa em nome de um cliente, por exemplo, monitorando leilões e dando lances no último minuto para maximizar as chances de sucesso.

9.7.2 A grade

O termo *grade* (*grid*) é usado para se referir ao *middleware* projetado para permitir o compartilhamento de recursos como arquivos, computadores, *software*, dados e sensores em uma escala muito grande. Normalmente, os recursos são compartilhados por grupos de usuários em diferentes organizações, os quais estão colaborando na solução de problemas que exigem grandes números de computadores para resolvê-los, pelo compartilhamento de dados ou pelo compartilhamento de poder de computação. Esses recursos são necessariamente suportados por *hardware*, sistemas operacionais, lingua-

O World-Wide Telescope: uma aplicação de grade

Esse projeto está relacionado à distribuição de recursos de dados compartilhados pela comunidade de astronomia. Ele está descrito no trabalho de Szalay e Gray [2004], Szalay e Gray [2001] e Gray e Szalay [2002]. Os dados astronômicos consistem em repositórios de arquivo de observações, cada um dos quais abrangendo um período de tempo em particular, uma parte do espectro eletromagnético (ótico, raios X, rádio) e uma área em particular do céu. Essas observações são feitas por diferentes instrumentos instalados em vários lugares do mundo.

Um estudo sobre como os astrônomos compartilham seus dados é útil para extrair as características de uma aplicação típica de grade, pois eles compartilham livremente seus resultados uns com os outros e os problemas de segurança podem ser omitidos, tornando esta discussão mais simples.

Os astrônomos fazem estudos que precisam combinar dados sobre os mesmos objetos celestes, mas que envolvem períodos de tempo diferentes e diversas partes do espectro. A capacidade de usar observações de dados independentes é importante para a pesquisa. A visualização permite aos astrônomos verem os dados como mapas de dispersão bidimensionais ou tridimensionais.

As equipes que reúnem os dados os armazenam em imensos repositórios de arquivo (atualmente, na ordem de terabytes), os quais são gerenciados de forma local por cada equipe. Os instrumentos usados na coleta de dados estão sujeitos à lei de Moore. Por isso, o volume dos dados reunidos cresce exponencialmente. À medida que são reunidos, os dados brutos são analisados e processados em uma sequência de passos e armazenados como dados derivados para uso pelos astrônomos de todo o mundo. Porém, antes que os dados possam ser usados por outros pesquisadores, os cientistas que trabalham em um campo em particular precisam concordar com uma maneira comum de rotular seus dados.

Szalay e Gray [2004] mencionam que, no passado, os dados de pesquisa científica eram incluídos pelos autores em artigos e publicados em periódicos que ficavam em bibliotecas. Contudo, atualmente, o volume de dados é grande demais para ser incluído em uma publicação. Isso se aplica não apenas à astronomia, mas também aos campos da física de partícula, genoma e pesquisa biológica. A função do autor agora se relaciona às colaborações, o qual leva de 5 a 10 anos para construir sua experiência, antes de produzir os dados que são publicados para o mundo em repositórios de arquivo baseados na Web. Assim, os cientistas que trabalham em projetos se tornam geradores (anunciantes) de dados e bibliotecários, assim como autores.

Essa função adicional exige que todo projeto que gerencia um repositório de arquivo de dados o torne acessível para outros pesquisadores. Isso implica uma sobrecarga considerável, além da tarefa original de análise dos dados. Para tornar tal compartilhamento possível, os dados brutos exigem metadados para descrever, por exemplo, a hora em que foram coletados, a parte do céu observada e o instrumento utilizado. Além disso, os dados derivados precisam ser acompanhados de metadados que descrevam os parâmetros das operações com os quais foram processados.

O cálculo dos dados derivados exige suporte computacional pesado. Frequentemente, eles precisam ser recalculados, à medida que as técnicas melhoram. Tudo isso é uma despesa considerável para o projeto que possui os dados.

O objetivo do World-Wide Telescope é unificar os repositórios de arquivo de astronomia do mundo em um banco de dados gigante, contendo literatura, imagens, dados brutos, conjuntos de dados derivados e dados de simulação de astronomia.

gens de programação e aplicações heterogêneas. É necessário gerenciamento para ordenar o uso de recursos para garantir que os clientes obtenham o que precisam e que os serviços possam fornecê-los. Em alguns casos, são exigidas técnicas de segurança sofisticadas para garantir o uso correto dos recursos nesse tipo de ambiente. Para um

exemplo de aplicação de grade, veja o quadro, que apresenta o aplicativo World-Wide Telescope, desenvolvido na Microsoft Research.

Requisitos das aplicações de grade • O World-Wide Telescope é um caso típico de uma variedade de *aplicações de grade com uso intensivo de dados*, em que:

- os dados são reunidos por meio de instrumentos científicos;
- os dados são armazenados em repositórios de arquivo em um conjunto de *sites* separados, cujas localizações podem ser em diferentes lugares, em qualquer parte do mundo;
- os dados são gerenciados por equipes de cientistas pertencentes a organizações separadas;
- um volume imenso e cada vez maior (terabytes ou petabytes) de dados brutos é gerado a partir dos instrumentos;
- programas de computador serão usados para analisar e fazer resumos dos dados brutos, por exemplo, para classificar, calibrar e catalogar os dados brutos que representam objetos celestes.

A Internet torna todos esses repositórios de arquivo de dados potencialmente disponíveis para cientistas de todo o mundo. Eles poderão obter dados de diferentes instrumentos, extraídos em diferentes momentos e em diferentes locais. Entretanto, um cientista em particular, usando esses dados para sua própria pesquisa, estará interessado apenas em um subconjunto dos objetos presentes nos repositórios de arquivo.

O imenso volume de dados em um repositório de arquivo torna impraticável transferi-los para o local do usuário, antes de processá-los para extrair os objetos de interesse, devido às considerações sobre o tempo de transmissão e o espaço no disco local exigidos. Portanto, não é apropriado usar FTP ou acesso Web nesse contexto. O processamento dos dados brutos deve ocorrer no local onde eles são reunidos e armazenados em um banco de dados. Então, quando um cientista fizer uma consulta sobre objetos em particular, as informações presentes em cada banco de dados devem ser analisadas e, se necessário, visualizações devem ser produzidas, antes de retornar os resultados para a consulta remota.

O fato de os dados serem processados em muitos *sites* diferentes proporciona um paralelismo que efetivamente divide a imensa tarefa que está sendo executada.

A partir das características anteriores, são inferidos os seguintes requisitos:

- R1: Acesso remoto aos recursos – isto é, às informações exigidas presentes nos repositórios de arquivo.
- R2: Processamento de dados no local onde eles são armazenados e gerenciados, ou quando são reunidos ou em resposta a uma requisição. Uma consulta típica poderia resultar em uma visualização baseada nos dados reunidos para uma região do céu, gravados por diferentes instrumentos, em diferentes momentos. Isso envolverá selecionar um pequeno volume de dados de cada repositório de arquivo de dados maciço.
- R3: O gerenciador de recursos de um repositório de arquivo de dados deve ser capaz de criar instâncias de serviço dinamicamente para lidar com a seção de dados em particular exigida, exatamente como no modelo de objeto distribuído, em que serventes são criados, quando necessário, para manipular diferentes recursos gerenciados por um serviço.
- R4: Metadados para descrever:

- características dos dados em um repositório de arquivo, por exemplo, para astronomia: a área do céu, a data e a hora da coleta e os instrumentos utilizados;
- as características de um serviço que esteja gerenciando esses dados, por exemplo, seu custo, sua localização geográfica, seu gerador (anunciante) ou sua carga ou espaço disponível.

- R5: Serviços de diretório baseados nos metadados acima.
- R6: *Software* para gerenciar consultas, transferências de dados e reserva antecipada de recursos, levando em conta que os recursos geralmente são gerenciados pelos projetos que geram os dados e que o acesso a eles talvez precise ser racionado.

Os serviços Web podem tratar dos dois primeiros requisitos, fornecendo uma maneira conveniente de permitir que os cientistas acessem operações sobre dados em repositórios de arquivo remotos. Isso exigirá que cada aplicação em particular forneça uma descrição do serviço que inclua um conjunto de métodos para acessar seus dados. O *middleware* de grade deve tratar dos requisitos restantes.

As grades também são usadas para *aplicações de uso intensivo de poder computacional*, como no processamento do enorme volume de dados produzido pelo acelerador de partículas de alta energia CMS no CERN [www.uscms.org], no teste dos efeitos de moléculas de possíveis remédios [Taufer *et al.* 2003, Chien 2004] ou no suporte de jogos *online* para muitos jogadores usando a capacidade ociosa em grupos de computadores [www.butterfly.net]. Quando aplicações de uso intensivo de poder computacional forem distribuídas em uma grade, o gerenciamento de recursos se preocupará com a alocação de recursos de computação e com a harmonização das cargas.

Finalmente, muitas aplicações em grade como, por exemplo, as médicas e as comerciais, precisarão considerar aspectos de segurança. Mesmo quando a privacidade dos dados não é problema, será importante estabelecer a identidade das pessoas que criaram os dados.

Middleware de grade • A arquitetura aberta de serviços de grade (OGSA, Open Grid Services Architecture) é um padrão para aplicações baseadas em grade [Foster *et al.* 2001, 2002]. Ela fornece uma estrutura em que os requisitos anteriores podem ser satisfeitos, baseada em serviços Web. Os recursos são gerenciados por serviços de grade específicos da aplicação. O *toolkit* Globus implementa a arquitetura.

O Projeto Globus começou em 1994 com o objetivo de fornecer *software* que integrasse e padronizasse as funções exigidas por uma família de aplicações científicas. Essas funções incluem serviços de diretório, segurança e gerenciamento de recursos. O primeiro *toolkit* Globus apareceu em 1997. O OGSA evoluiu a partir da segunda versão do *toolkit*, chamado de GT2, que está descrito em Foster e Kesselman [2004]. A terceira versão (GT3), que apareceu em 2002, foi baseada no OGSA e, portanto, construída em serviços Web. Ela foi desenvolvida pela Globus Alliance (www.globus.org) e está descrita em Sandholm e Gawor [2003]. Desde então, foram lançadas mais duas versões – a mais recente é denominada GT5 e está disponível como *software* de código-fonte aberto [www.globus.org].

Um estudo de caso da OGSA e do *toolkit* Globus (até o GT3) pode ser encontrado no site que acompanha o livro [www.cdk5.net/Web] (em inglês).

9.7.3 Computação em nuvem

A computação em nuvem foi apresentada no Capítulo 1 como um conjunto de serviços de aplicativo, armazenamento e computação baseados na Internet, suficientes para suportar a maioria das necessidades dos usuários, permitindo com isso que, em grande medida ou totalmente, eles prescindam de armazenamento de dados e de *software* aplicativo locais. A computação em nuvem também promove a visão de tudo como serviço, desde infraestrutura física ou virtual até *software*, frequentemente pago de acordo com a utilização, em vez de ser adquirido. Portanto, o conceito está intrinsecamente ligado a um novo modelo comercial de computação, no qual os fornecedores da nuvem oferecem diversos serviços computacionais, de dados e outros, para os clientes, conforme o exigido para seu uso diário – oferecendo, por exemplo, pela Internet, capacidade de armazenamento suficiente para atuar como serviço de repositório de arquivos ou *backup*.

O Capítulo 1 também comentou a sobreposição entre computação em nuvem e a grade. O desenvolvimento da grade precedeu o surgimento da computação em nuvem e foi um fator significativo para sua aparição. Elas têm o mesmo objetivo de fornecer recursos (serviços) na Internet. Enquanto a grade tende a se concentrar em aplicações de uso intensivo de dados de alta capacidade ou computacionalmente dispendiosas, a computação em nuvem é mais geral, oferecendo diversos serviços para computadores individuais e usuários finais. O modelo comercial associado à computação em nuvem também é uma característica distintiva. Portanto, é justo dizer que a grade é um exemplo primitivo de computação em nuvem; no entanto, desde então, a computação em nuvem se desenvolveu significativamente.

Com essa visão de tudo ser serviço, os serviços Web oferecem um caminho de implementação natural para computação em nuvem e, realmente, muitos fornecedores entram por ele. O produto mais notável nesse setor é o AWS (Amazon Web Services) [aws.amazon.com] – examinaremos brevemente essa tecnologia, a seguir. Veremos uma estratégia alternativa à computação em nuvem, no Capítulo 21, quando examinarmos a infraestrutura do Google e o mecanismo Google App Engine associado, os quais apresentam uma estratégia mais leve e de maior desempenho do que os serviços Web.

Amazon Web Services é um conjunto de serviços em nuvem implementados na extensa infraestrutura física pertencente à Amazon.com. Originalmente desenvolvida para propósitos internos no suporte para suas vendas a varejo eletrônicas, agora a Amazon oferece muitos dos recursos para usuários externos, permitindo a eles executar serviços independentes na sua infraestrutura. A implementação do AWS trata dos principais problemas dos sistemas distribuídos, como o gerenciamento da disponibilidade de serviço, escalabilidade e desempenho, permitindo que os desenvolvedores se concentrem no uso de seus serviços. Os serviços são disponibilizados usando-se os padrões de serviço Web descritos anteriormente neste capítulo. Isso tem a vantagem de que os programadores que conhecem os serviços Web podem usar o AWS prontamente e desenvolver *mashups* que incorporam Amazon Web Services em sua construção. Mais geralmente, a estratégia permite interoperabilidade na Internet. A Amazon também adota a estratégia REST, conforme defendida por Fielding [2000] e discutida na Seção 9.2.

A Amazon oferece um conjunto de serviços amplo e extensível, dos quais os mais importantes estão listados na Figura 9.19. Apresentamos o EC2 com mais detalhes. O EC2 é um serviço de computação elástico, no qual o termo *elástico* se refere à proprie-

Serviço web	Descrição
Amazon Elastic Compute Cloud (EC2)	Serviço baseado na Web que oferece acesso a máquinas virtuais de determinado desempenho e capacidade de armazenamento
Amazon Simple Storage Service (S3)	Serviço de armazenamento baseado na Web para dados não estruturados
Amazon Simple DB	Serviço de armazenamento baseado na Web para consultar dados estruturados
Amazon Simple Queue Service (SQS)	Serviço hospedado que suporta filas de mensagens (conforme discutido no Capítulo 6)
Amazon Elastic MapReduce	Serviço baseado na web para computação distribuída usando o modelo MapReduce (apresentado no Capítulo 21)
Amazon Flexible Payments Service (FPS)	Serviço baseado na Web que suporta pagamentos eletrônicos

Figura 9.19 Uma seleção de Amazon Web Services.

dade de oferecer capacidade de computação redimensionada de acordo com as necessidades do cliente. Em vez de uma máquina real, o EC2 oferece ao usuário uma máquina virtual, denominada *instância*, de acordo com a especificação desejada. Por exemplo, um usuário pode solicitar uma instância dos seguintes tipos:

- uma *instância padrão*, projetada para ser adequada para a maioria das aplicações;
- uma *instância de grande capacidade de memória*, que oferece capacidade de memória adicional, por exemplo, para aplicações que envolvem o uso de cache;
- uma instância de *grande capacidade de CPU*, projetada para suportar tarefas de computação intensa;
- uma *instância de computação em grupo*, que oferece um grupo de processadores virtuais com interconexão de alta largura de banda para tarefas de computação de alto desempenho.

Vários deles podem ser ainda mais refinados – por exemplo, para uma instância padrão é possível solicitar uma instância pequena, média ou grande, representando diferentes especificações em termos de poder de processamento, memória, armazenamento em disco, etc.

O ECS é construído sobre o hipervisor Xen, descrito na Seção 7.7.2. As instâncias podem ser configuradas para executar diversos sistemas operacionais, incluindo Windows Server 2008, Linux ou OpenSolaris. Também podem ser configuradas com uma variedade de *software*. Por exemplo, é possível solicitar uma instalação de Apache HTTP para suportar hospedagem na Web.

O EC2 suporta o interessante conceito de endereço IP elástico, que parece um endereço IP tradicional, mas é associado à conta do usuário e não a uma instância em particular. Isso significa que, se uma máquina (virtual) falha, o endereço IP pode ser reatribuído a uma máquina diferente, sem exigir a intervenção de um administrador de rede.

9.8 Resumo

Neste capítulo, mostramos que os serviços Web surgiram da necessidade de fornecer uma infraestrutura para suportar interligação em rede entre diferentes organizações. Essa infraestrutura geralmente usa o conhecido protocolo HTTP para transportar mensagens entre clientes e servidores pela Internet e é baseada no uso de URIs para se referir aos recursos. A XML, um formato textual, é usada para representação e empacotamento de dados.

Duas influências distintas levaram à aparição de serviços Web. Uma delas é a adição de interfaces de serviço em servidores Web com o objetivo de permitir que os recursos de um *site* fossem acessados por programas clientes que não os navegadores, além de usar uma forma de interação mais rica. A outra é o desejo de fornecer algo como RPC na Internet, com base nos protocolos existentes. Os serviços Web resultantes fornecem interfaces com conjuntos de operações que podem ser chamadas de forma remota. Assim como qualquer outra forma de serviço, um serviço Web pode ser cliente de outro serviço Web, permitindo que um serviço Web integre um conjunto de outros serviços Web ou combine com ele.

SOAP é o protocolo de comunicação geralmente usado pelos serviços Web e seus clientes. Ele pode ser usado para transmitir mensagens de requisição e suas respostas entre cliente e servidor, ou pela troca assíncrona de documentos ou por uma forma de protocolo de requisição-resposta baseado em um par de trocas de mensagens assíncronas. Nos dois casos, a mensagem de requisição ou resposta é incluída em um documento formatado em XML, chamado de envelope. Geralmente, o envelope SOAP é transmitido por meio do protocolo HTTP síncrono, embora outros transportes possam ser usados.

Processadores de XML e SOAP estão disponíveis para todas as linguagens de programação e sistemas operacionais amplamente usados. Isso permite que os serviços Web e seus clientes sejam implantados em quase qualquer lugar. Essa forma de interligação em rede é possível pelo fato de que os serviços Web não estão ligados a nenhuma linguagem de programação em particular nem suportam o modelo de objeto distribuído.

Nos *middlewares* convencionais as definições de interface fornecem aos clientes os detalhes dos serviços. Entretanto, no caso de serviços Web, são usadas descrições de serviço. Uma descrição de serviço especifica o protocolo de comunicação a ser usado (por exemplo, SOAP) e o URI do serviço, assim como descreve sua interface. A interface pode ser descrita como um conjunto de operações, ou como um conjunto de mensagens, a serem trocadas entre cliente e servidor.

A segurança em XML foi projetada para fornecer a proteção necessária para o conteúdo de um documento trocado por membros de um grupo de pessoas, as quais têm tarefas diferentes para realizar nesse documento. Diferentes partes do documento estarão disponíveis para diferentes pessoas, algumas com a capacidade de adicionar ou alterar o conteúdo e outras de lê-lo. Para permitir uma flexibilidade completa em seu uso futuro, as propriedades de segurança são definidas dentro do próprio documento. Isso é obtido por meio da XML, que tem um formato auto-descritivo. Elementos da XML são usados para especificar as partes do documento que são cifradas ou assinadas, assim como os detalhes dos algoritmos usados e informações para ajudar a encontrar chaves.

Os serviços Web têm sido usados para diversos propósitos nos sistemas distribuídos. Por exemplo, eles fornecem uma implementação natural do conceito de arquitetura orientada a serviços, na qual seu baixo acoplamento permite interoperabilidade em apli-

cações na Internet – incluindo as de empresa para empresa (B2B). Seu baixo acoplamento inerente também suporta o surgimento de uma estratégia de *mashup* para a construção de serviços Web. Os serviços Web também servem de base para a grade, suportando colaborações entre cientistas ou engenheiros em organizações de diferentes partes do mundo. Muito frequentemente, seu trabalho é baseado no uso de dados brutos coletados por instrumentos em diferentes lugares e, depois, processados de forma local. O *toolkit* Globus é uma implementação da arquitetura que tem sido usada em uma variedade de aplicações de uso intenso de dados e poder computacional. Por fim, os serviços Web são expressivamente usados na computação em nuvem. Por exemplo, o AWS da Amazon é totalmente baseado em padrões de serviço Web, acoplados à filosofia REST de construção de serviço.

Exercícios

- 9.1 Compare o protocolo de requisição-resposta descrito na Seção 5.2 com a implementação de comunicação cliente-servidor no SOAP. Cite dois motivos pelos quais o uso de mensagens assíncronas pelo SOAP é mais apropriado para uso na Internet. Até que ponto o uso de HTTP pelo SOAP reduz a diferença entre as duas estratégias? *páginas 388*
- 9.2 Compare a estrutura dos URLs, conforme usados pelos serviços Web, com as referências de objeto remoto, conforme especificadas na Seção 4.3.4. Cite, em cada caso, como elas são usadas para executar um pedido do cliente. *páginas 393*
- 9.3 Ilustre o conteúdo de uma mensagem SOAP de requisição e de sua mensagem de resposta correspondente para o serviço de *Election* dado no Exercício 5.11. Use, na sua resposta, uma versão pictórica da XML, como mostrado nas Figuras 9.4 e 9.5. *página 389*
- 9.4 Descreva em linhas gerais os cinco principais elementos de uma descrição do serviço WSDL. No caso do serviço *Election* definido no Exercício 5.11, cite o tipo de informação a ser usada pelas mensagens de requisição e de resposta – algum deles precisa ser incluído no espaço de nomes de destino? Para a operação *vote*, desenhe diagramas semelhantes às Figuras 9.11 e 9.13. *página 402*
- 9.5 Continuando com o exemplo do serviço *Election*, explique por que a parte da WSDL definida no Exercício 9.4 é referida como “abstrata”. O que precisaria ser adicionado na descrição do serviço para torná-lo completamente concreto? *página 400*
- 9.6 Defina uma interface Java para o serviço *Election*, conveniente para uso como um serviço Web. Diga por que você acha que a interface que definiu é conveniente. Explique como um documento WSDL para o serviço é gerado e como se torna disponível para os clientes. *página 396*
- 9.7 Descreva o conteúdo de um proxy de cliente Java para o serviço *Election*. Explique como os métodos de empacotamento e desempacotamento podem ser obtidos para um proxy estático. *página 396*
- 9.8 Explique a função de um contêiner de *servlet* na distribuição de um serviço Web e na execução de uma requisição de cliente. *página 396*
- 9.9 No exemplo em Java ilustrado nas Figuras 9.8 e 9.9, o cliente e o servidor estão lidando com objetos, embora os serviços Web não suportem objetos distribuídos. Como isso pode acontecer? Quais são as limitações impostas sobre as interfaces de serviços Web em Java? *página 395*

- 9.10 Descreva, em linhas gerais, o esquema de replicação usado no UDDI. Supondo que carimbos de tempo vetoriais são usados para suportar esse esquema, defina duas operações para uso por registros que precisem trocar dados. *página 406*
- 9.11 Explique por que o UDDI pode ser descrito como serviço de nome e como serviço de diretório, mencionando os tipos de perguntas que podem ser feitas. O segundo “D” no nome UDDI se refere a descoberta – o UDDI é realmente um serviço de descoberta? *Capítulo 13 e página 404*
- 9.12 Descreva, em linhas gerais, a principal diferença entre TLS e segurança em XML. Explique por que a XML é particularmente conveniente para a função que desempenha, em termos dessas diferenças. *Capítulo 11 e página 406*
- 9.13 Os documentos protegidos pela segurança em XML podem ser assinados ou cifrados muito tempo antes que alguém possa prever quem serão os destinatários finais. Quais medidas são adotadas para garantir que estes últimos tenham acesso aos algoritmos usados pelo primeiro? *página 406*
- 9.14 Explique a relevância da XML canônica nas assinaturas digitais. Quais informações contextuais podem ser incluídas na forma canônica? Dê um exemplo de brecha de segurança em que o contexto é omitido da forma canônica. *página 409*
- 9.15 Um protocolo de coordenação poderia ser executado para coordenar as ações dos serviços Web. Descreva em linhas gerais uma arquitetura para (i) um protocolo centralizado e (ii) um protocolo de coordenação distribuída. Em cada caso, descreva as interações necessárias para estabelecer a coordenação entre dois serviços Web. *página 411*
- 9.16 Compare a semântica de chamada RPC com a semântica do *WS-ReliableMessaging*:
- Cite as entidades às quais cada uma se refere.
 - Compare os diferentes significados da semântica disponível (por exemplo, *pelo menos uma vez, no máximo uma vez, exatamente uma vez*). *Capítulo 5 e página 392*

10

Sistemas Peer-to-peer

- 10.1 Introdução
- 10.2 Napster e seu legado
- 10.3 Middleware para peer-to-peer
- 10.4 Sobreposição de roteamento
- 10.5 Estudos de caso: Pastry, Tapestry
- 10.6 Estudo de caso: Squirrel, OceanStore, Ivy
- 10.7 Resumo

Os sistemas *peer-to-peer* representam um paradigma para a construção de sistemas e de aplicativos distribuídos em que dados e recursos computacionais são provenientes da colaboração de muitas máquinas na Internet de maneira uniforme. Seu aparecimento é uma consequência do crescimento rápido da Internet, abrangendo milhões de computadores e número semelhante de usuários exigindo acesso a recursos compartilhados.

Um problema importante dos sistemas *peer-to-peer* é a distribuição de objetos de dados em muitos computadores e o subsequente acesso a eles de uma maneira que equilibre a carga de trabalho e garanta a disponibilidade sem adicionar sobrecargas indevidas. Descreveremos vários sistemas e aplicativos desenvolvidos recentemente projetados para isso.

Vários *middlewares* para sistema *peer-to-peer* têm surgido oferecendo capacidade para compartilhar recursos computacionais, armazenamento e dados em computadores “nos limites da Internet”, em uma escala global. Eles exploram de novas maneiras as técnicas de atribuição de nomes, roteamento, replicação de dados e segurança existentes para construir uma camada de compartilhamento de recursos confiável em um conjunto de computadores e redes inseguros e não confiáveis.

Os aplicativos *peer-to-peer* têm sido usados para fornecer compartilhamento de arquivos, uso de cache Web, distribuição de informações e outros serviços que exploram os recursos de dezenas de milhares de máquinas pela Internet. Eles demonstram sua maior eficácia quando usados para armazenar conjuntos muito grandes de dados imutáveis. Seu projeto diminui sua eficácia para aplicações que armazenam e atualizam objetos de dados mutáveis.

10.1 Introdução

Espera-se que a demanda por serviços na Internet cresça em uma escala limitada apenas pelo tamanho da população mundial. O objetivo dos sistemas *peer-to-peer* é permitir o compartilhamento de dados e recursos em uma escala muito grande, eliminando qualquer exigência de servidores gerenciados separadamente e sua infraestrutura associada.

A abrangência da expansão de serviços populares pelo acréscimo do número de computadores que os contêm é limitada quando todos eles devem pertencer e ser gerenciados pelo provedor de serviço. Os custos de administração e recuperação de falhas tendem a influenciar. A largura de banda de rede fornecida por um único *site* servidor sobre os enlaces físicos disponíveis também é uma restrição importante. Os serviços em nível de sistema, como o Sun NFS (Seção 12.3), o Andrew File System (Seção 12.4) ou os servidores de vídeo (Seção 20.6.1), e os serviços em nível de aplicação, como Google, Amazon ou eBay, apresentam esse problema em diversos graus.

Os sistemas *peer-to-peer* têm como objetivo suportar serviços e aplicativos distribuídos, usando dados e recursos computacionais disponíveis nos computadores pessoais e estações de trabalho que estão presentes em números cada vez maiores na Internet e em outras redes. Isso é cada vez mais atraente, à medida que diminui a diferença de desempenho entre as máquinas *desktop* e servidores e que as conexões a redes de banda larga proliferam.

No entanto, existe outro objetivo mais amplo: um autor [Shirky 2000] definiu os aplicativos *peer-to-peer* como “aplicativos que exploram os recursos disponíveis nos limites da Internet – armazenamento, ciclos de processamento, conteúdo, presença humana”. Cada tipo de compartilhamento de recurso mencionado nessa definição já está representado pelos aplicativos distribuídos disponíveis para a maioria dos tipos de computador pessoal. O objetivo deste capítulo é descrever algumas técnicas gerais que simplificam a construção de aplicativos *peer-to-peer* e que melhoram sua escalabilidade, sua confiabilidade e sua segurança.

Os sistemas cliente-servidor tradicionais gerenciam e fornecem acesso a recursos como arquivos, páginas Web ou outros objetos de informação localizados em um único computador servidor ou em um pequeno agrupamento de servidores fortemente acoplados. Em tais projetos centralizados, são exigidas poucas decisões sobre a distribuição dos recursos ou sobre o gerenciamento dos recursos de *hardware*, mas a escala do serviço é limitada pela capacidade do *hardware* do servidor e pela conectividade da rede. Os sistemas *peer-to-peer* fornecem acesso a recursos de informação localizados em computadores de toda uma rede (seja ela a Internet ou uma rede corporativa). Os algoritmos para a distribuição e subsequente recuperação de objetos de informação são um aspecto importante do projeto do sistema. Seu projeto tem como objetivo distribuir um serviço totalmente descentralizado e organizado, equilibrando, automaticamente, as cargas de armazenamento e processamento de forma dinâmica entre todos os computadores participantes à medida que as máquinas entram e saem do serviço.

Os sistemas *peer-to-peer* compartilham as seguintes características:

- Seu projeto garante que cada usuário contribua com recursos para o sistema.
- Embora eles possam diferir nos recursos com que contribuem, todos os nós em um sistema *peer-to-peer* têm as mesmas capacidades e responsabilidades funcionais.
- Seu correto funcionamento não depende da existência de quaisquer sistemas administrados de forma centralizada.

- Eles podem ser projetados de modo a oferecer um grau limitado de anonimato para os provedores e usuários dos recursos.
- Um problema importante para seu funcionamento eficiente é a escolha de um algoritmo para a distribuição dos dados em muitas máquinas e o subsequente acesso a eles, de uma maneira que equilibre a carga de trabalho e garanta a disponibilidade sem adicionar sobrecargas indevidas.

Os computadores e conexões de rede pertencentes e gerenciadas por muitos usuários e organizações diferentes são necessariamente recursos voláteis; seus proprietários não garantem que irão mantê-los ligados, conectados e isentos de falhas. Portanto, a disponibilidade dos processos e dos computadores participantes dos sistemas *peer-to-peer* é imprevisível. Assim, os serviços *peer-to-peer* não podem contar com acesso garantido a recursos individuais, embora possam ser projetados de forma a tornar a probabilidade de falha no acesso a uma cópia de um objeto replicado arbitrariamente pequena. É interessante notar que essa deficiência dos sistemas *peer-to-peer* pode se tornar uma vantagem, caso a replicação de recursos que eles solicitam seja explorada de forma a se obter um grau de resistência à falsificação feita por computadores (nós) mal-intencionados (isto é, por meio de técnicas de tolerância a falhas bizantina, veja o Capítulo 18).

Vários serviços primitivos baseados na Internet, incluindo o DNS (Seção 13.2.3) e *Netnews/Usenet* [Kantor e Lapsley 1986], adotaram uma arquitetura de vários servidores, escalável e tolerante a falhas. O serviço de registro de nomes e distribuição de *e-mail* Grapevine da Xerox [Birrell *et al.* 1982, Schroeder *et al.* 1984] fornece um exemplo primitivo interessante de serviço distribuído com escalabilidade e tolerância a falhas. O algoritmo de *parlamento de meio expediente* para consenso distribuído de Lamport [Lamport 1989], o sistema de armazenamento replicado Bayou (veja a Seção 18.4.2) e o algoritmo de roteamento de IP entre domínios *classless* (veja a Seção 3.4.3) são todos exemplos de algoritmos distribuídos para a distribuição, ou localização, de informações e podem ser considerados como antecedentes dos sistemas *peer-to-peer*.

No entanto, o potencial para a implantação de serviços *peer-to-peer*, usando recursos nos limites da Internet, surgiu apenas quando um número significativo de usuários adquiriu conexões de banda larga sempre ativas para a rede, tornando seus computadores *desktop* plataformas convenientes para o compartilhamento de recursos. Isso ocorreu primeiro nos Estados Unidos, por volta de 1999. Em meados de 2004, o número mundial de conexões de banda larga na Internet tinha ultrapassado tranquilamente os 100 milhões [Internet World Stats 2004].

Podem ser identificadas três gerações de desenvolvimento de sistemas e aplicativos *peer-to-peer*. A primeira geração foi lançada pelo serviço de troca de músicas Napster [OpenNap 2001], que descreveremos na próxima seção. Uma segunda geração de aplicativos de compartilhamento de arquivos, oferecendo maior escalabilidade, anonimato e tolerância a falhas surgiu logo depois, incluindo Freenet [Clarke *et al.* 2000, freenetproject.org], Gnutella, Kazaa [Leibowitz *et al.* 2003] e BitTorrent [Cohen 2003].

Middleware peer-to-peer • A terceira geração é caracterizada pelo aparecimento de camadas de *middleware* para o gerenciamento de recursos distribuídos em uma escala global independente de aplicativos. Agora, várias equipes de pesquisa concluíram o desenvolvimento, a avaliação e o refinamento de plataformas de *middleware peer-to-peer* e as demonstraram, ou implantaram, em diversos serviços de aplicativo. Os exemplos mais conhecidos e totalmente desenvolvidos incluem o Pastry [Rowstron e Druschel 2001], o Tapestry [Zhao *et al.* 2004], o CAN [Ratnasamy *et al.* 2001], o Chord [Stoica *et al.* 2001] e o Kademia [Maymounkov e Mazières 2002].

	IP	Roteamento em nível de aplicativo
<i>Escala</i>	O IPv4 é limitado a 2^{32} nós endereçáveis. O espaço de nomes IPv6 é muito mais generoso (2^{128}), mas nas duas versões os endereços são estruturados hierarquicamente e grande parte do espaço é previamente alocado de acordo com os requisitos administrativos.	Os sistemas <i>peer-to-peer</i> podem endereçar mais objetos. O espaço de nomes do GUID é muito grande e plano ($>2^{128}$), podendo ser ocupado de forma muito mais completa.
<i>Equilíbrio da carga</i>	As cargas sobre os roteadores são determinadas pela topologia da rede e pelos padrões de tráfego vigentes.	Os objetos podem se posicionar aleatoriamente e, com isso, os padrões de tráfego não têm a ver com a topologia da rede.
<i>Dinâmica da rede (adição/exclusão de objetos/nós)</i>	As tabelas de roteamento de IP são atualizadas de forma assíncrona, com base no melhor esforço, com constantes de tempo na ordem de uma hora.	As tabelas de roteamento podem ser atualizadas de forma síncrona ou assíncrona, com atrasos de frações de segundo.
<i>Tolerância a falhas</i>	A redundância é projetada na rede IP por seus gerentes, garantindo tolerância a falhas de conectividade de um único roteador ou da rede. A replicação com fator de multiplicação n é dispendiosa.	Rotas e referências de objeto podem ser replicadas por um fator n , garantindo a tolerância de n falhas de nós ou conexões.
<i>Identificação do destino</i>	Cada endereço IP é mapeado em exatamente um nó de destino.	As mensagens podem ser direcionadas para a réplica mais próxima de um objeto de destino.
<i>Segurança e anonimato</i>	O endereçamento só é seguro quando todos os nós são confiáveis. O anonimato dos proprietários de endereços não pode ser obtido.	A segurança pode ser obtida, mesmo em ambientes de confiança limitada. Um grau limitado de anonimato pode ser fornecido.

Figura 10.1 Distinções entre IP e roteamento em redes de sobreposição para aplicações *peer-to-peer*.

Essas plataformas são projetadas para colocar recursos (objetos de dados, arquivos) em um conjunto de computadores amplamente distribuídos em toda a Internet e para direcionar mensagens a eles em nome de clientes, retirando destes as decisões sobre o posicionamento de recursos e a necessidade de conter informações sobre o paradeiro dos recursos que exigem. Ao contrário dos sistemas de segunda geração, eles dão garantias do envio de pedidos em um número limitado de passos intermediário de rede. Eles colocam réplicas dos recursos de maneira estruturada nos computadores disponíveis, levando em conta sua disponibilidade volátil, sua confiabilidade e seus requisitos de equilíbrio de carga variáveis, localização do armazenamento e uso das informações.

Os recursos são identificados por identificadores globalmente exclusivos (GUIDs), e esses identificadores normalmente são obtidos por meio da aplicação de funções de resumo (*hashing*) seguras (descrito na Seção 11.4.3), a partir de algum, ou de todos, os estados do recurso. O uso de uma função de resumo seguro torna um recurso “automa-

ticamente certificado” – os clientes que recebem um recurso podem verificar a validade do resumo. Isso o protege contra falsificação de nós não confiáveis nos quais pode ser armazenado. Contudo, essa técnica exige que os estados dos recursos sejam imutáveis, pois uma mudança no estado resultaria em um valor de resumo diferente. Por isso, os sistemas de armazenamento *peer-to-peer* são inherentemente mais convenientes para o armazenamento de objetos imutáveis (como arquivos de música ou vídeo). Seu uso em objetos com valores mutáveis é mais desafiador, mas pode ser resolvido com a adição de servidores confiáveis para gerenciar uma sequência de versões e identificar a versão corrente (como é feito, por exemplo, no OceanStore e no Ivy, descritos nas Seções 10.6.2 e 10.6.3).

O uso de sistemas *peer-to-peer* para aplicações que exigem um alto nível de disponibilidade para os objetos armazenados requer um projeto cuidadoso de aplicativos para evitar situações em que todas as réplicas de um objeto estão indisponíveis simultaneamente. Existe um risco que isso ocorra para objetos que são armazenados em computadores de uma mesma organização proprietária, localização geográfica, administração, conectividade de rede, país ou jurisdição. O uso de GUIDs distribuídas aleatoriamente ajuda no armazenamento de réplicas dos objetos também de forma aleatória entre os diversos nós da rede subjacente. Se a rede subjacente abrange muitas organizações no mundo, então o risco de indisponibilidade simultânea é muito reduzido.

Sobreposição de roteamento versus roteamento de IP • À primeira vista, o roteamento em redes de sobreposição (*routing overlay*) compartilha muitas características com a infraestrutura de roteamento de pacotes IP, que constitui o principal mecanismo de comunicação da Internet (veja a Seção 3.4.3). Portanto, é legítimo perguntar por que um mecanismo adicional de roteamento em nível de aplicativo é exigido nos sistemas *peer-to-peer*. A resposta está nas várias distinções identificadas na tabela da Figura 10.1. Pode-se argumentar que algumas dessas distinções surgem da natureza “legada” do IP como o principal protocolo da Internet, mas o impacto do legado provavelmente é forte demais para ser superado e para reprojetar o IP para suportar aplicativos *peer-to-peer* mais diretamente.

Computação distribuída • A exploração do poder de computação excedente nos computadores do usuário final tem sido assunto de interesse e experiências há um longo tempo. Um trabalho com os primeiros computadores pessoais no PARC da Xerox [Shoch e Hupp 1982] mostrou a exequibilidade da execução de tarefas que exigem alto poder computacional fracamente acopladas, executando processos *background* em aproximadamente 100 computadores pessoais interligados em uma rede local. Mais recentemente, números muito maiores de computadores foram usados para efetuar vários cálculos científicos que exigiam quantidades quase ilimitadas de poder de computação.

O trabalho desse tipo mais conhecido é o projeto *SETI@home* [Anderson *et al.* 2002], que faz parte de um projeto mais amplo da *Search for Extra-Terrestrial Intelligence*. O *SETI@home* particiona um fluxo de dados de radiotelescópio digitalizados, em unidades de trabalho de 107 segundos, cada um com cerca de 350 KB, e os distribui para computadores clientes cujo poder de computação recebe a contribuição de voluntários. Cada unidade de trabalho é distribuída de forma redundante para 3 a 4 computadores pessoais, para proteção contra nós errôneos ou mal-intencionados, e verificadas por padrões significativos de sinal. A distribuição das unidades de trabalho e a coordenação dos resultados são manipuladas por um único servidor, que é responsável pela comunicação com todos os clientes. Anderson *et al.* [2002] relataram que 3,91 milhões de computadores pessoais tinham participado do projeto *SETI@home* em agosto de 2002, resultando no processamento de 221 milhões de unidades de trabalho, representando uma média de

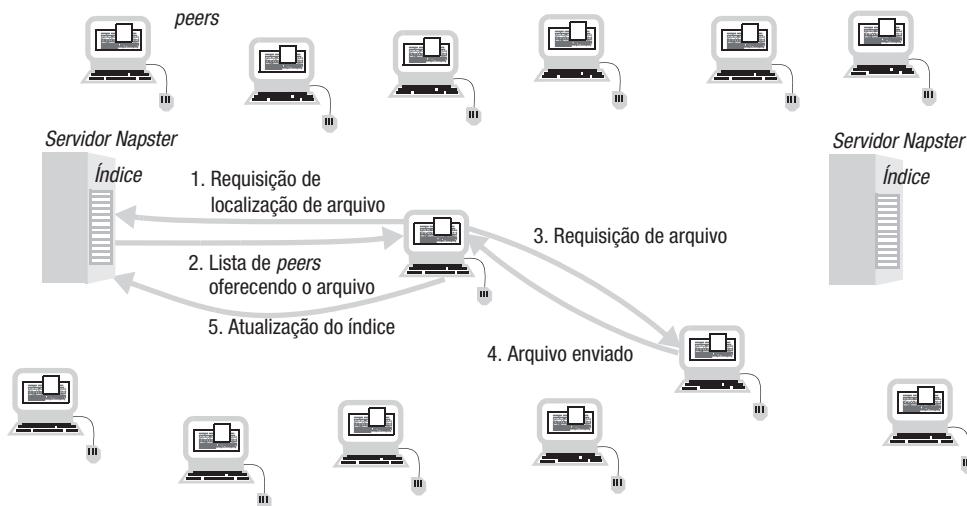


Figura 10.2 Napster: compartilhamento de arquivos peer-to-peer com um índice replicado centralizado.

27,36 teraflops de poder computacional durante 12 meses, até julho de 2003. O trabalho concluído até aquela data representou a maior computação jamais registrada.

A computação do projeto SETI@home é incomum, pois não envolve nenhuma comunicação ou coordenação entre computadores enquanto eles estão processando as unidades de trabalho, e os resultados são informados para um servidor central em uma única mensagem curta, que pode ser enviada quando o cliente e o servidor estão disponíveis. Algumas outras tarefas científicas dessa natureza foram identificadas, incluindo a procura de números primos grandes e tentativas de decifração pela força bruta. Porém, o desencadeamento do poder computacional na Internet para uma variedade mais ampla de tarefas dependerá do desenvolvimento de uma plataforma distribuída que suporte compartilhamento de dados e a coordenação da computação entre os computadores participantes em larga escala. Esse é o objetivo do projeto Grid, discutido no Capítulo 19.

Neste capítulo, focalizaremos os algoritmos e sistemas desenvolvidos até o momento para o compartilhamento de dados em redes *peer-to-peer*. Na Seção 10.2, resumiremos o projeto do Napster e examinaremos as lições aprendidas com ele. Na Seção 10.3, descreveremos os requisitos gerais das camadas de *middleware peer-to-peer*. As seções seguintes abordarão o projeto e a aplicação de plataformas de *middleware peer-to-peer*, começando com uma especificação abstrata, na Seção 10.4, seguida de descrições detalhadas de dois exemplos totalmente desenvolvidos (Seção 10.5) e algumas aplicações deles (Seção 10.6).

10.2 Napster e seu legado

A primeira aplicação na qual surgiu a exigência de um serviço de armazenamento e recuperação de informações com capacidade de escala global, foi o *download* de arquivos de música digital. Tanto a necessidade quanto a exequibilidade de uma solução *peer-to-peer* foram demonstradas pela primeira vez pelo sistema Napster [OpenNap 2001], que fornecia

um meio para os usuários compartilharem arquivos. O Napster se tornou muito popular para troca de música, logo após seu lançamento, em 1999. Em seu auge, vários milhões de usuários estavam registrados e milhares trocavam arquivos de música simultaneamente.

A arquitetura do Napster incluía índices centralizados, mas eram os usuários que forneciam os arquivos, os quais eram armazenados e acessados em seus computadores pessoais. O método de operação do Napster está ilustrado pela sequência de etapas mostrada na Figura 10.2. Note que, na etapa 5, espera-se que os clientes adicionem seus próprios arquivos de música no conjunto de recursos compartilhados, transmitindo para o serviço de indexação do Napster um *link* para cada arquivo disponível. Assim, a motivação do Napster, e o segredo de seu sucesso, foi tornar um grande conjunto de arquivos distribuídos disponível para os usuários em toda a Internet, cumprindo a máxima de Shirky por meio do fornecimento de acesso a “recursos compartilhados nos limites da Internet”.

O Napster foi desativado como resultado de ações jurídicas contra os operadores do serviço Napster, impetradas pelos proprietários dos direitos autorais de parte do material (isto é, música codificada de forma digital) que estava disponível nele. (Veja o quadro: *Os sistemas peer-to-peer e problemas de propriedade de direitos autorais*.)

O anonimato dos receptores e dos provedores de dados compartilhados e de outros recursos é uma preocupação dos projetistas de sistemas *peer-to-peer*. Em sistemas com muitos nós, o roteamento das requisições e dos resultados pode se tornar suficientemente tortuoso para ocultar sua fonte e o conteúdo dos arquivos pode ser distribuído em vários nós, dispersando a responsabilidade por torná-los disponíveis. Estão disponíveis mecanismos de comunicação anônima resistentes à maioria das formas de análise de tráfego [Goldschlag *et al.* 1999]. Se os arquivos também forem cifrados antes de serem colocados nos servidores, os proprietários dos servidores com certeza poderão negar qualquer conhecimento do conteúdo. No entanto, essas técnicas de anonimato aumentam o custo

Os sistemas peer-to-peer e problemas de propriedade de direitos autorais

Os desenvolvedores do Napster alegaram que não eram responsáveis pela violação dos direitos autorais dos proprietários porque não participavam do processo de cópia, o qual era realizado inteiramente entre as máquinas dos usuários. A alegação fracassou, porque os servidores de índice foram considerados uma parte essencial do processo. Como os servidores de índice estavam localizados em endereços conhecidos, seus operadores eram incapazes de se manter anônimos e, portanto, poderiam ser alvo de ações judiciais.

Um serviço de compartilhamento de arquivos mais completamente distribuído poderia ter obtido uma separação melhor das responsabilidades jurídicas, dispersando a responsabilidade por todos os usuários do Napster e, assim, tornando a busca de soluções jurídicas muito difícil, se não impossível. Qualquer que seja a visão que se tenha sobre a legitimidade da cópia de arquivos para o propósito de compartilhar material protegido por direitos autorais, existem justificativas sociais e políticas legítimas para o anonimato de clientes e servidores em alguns contextos de aplicação. A justificativa mais persuasiva surge quando o anonimato é usado para vencer a censura e manter a liberdade de expressão dos indivíduos em sociedades ou organizações opressivas.

Sabe-se que o *e-mail* e os *sites Web* têm desempenhado um papel significativo na obtenção do conhecimento público em tempos de crises políticas em tais sociedades; seu papel poderia ser mais atuante, se os autores pudessem ser protegidos pelo anonimato. O “delato” é um caso relacionado: um “delator” é um funcionário que publica, ou relata, as transgressões de seu empregador para as autoridades, sem revelar sua própria identidade por medo de sanções ou demissão. Em algumas circunstâncias, é razoável que tal ação seja protegida pelo anonimato.

do compartilhamento de recursos, e um trabalho recente mostrou que o anonimato disponível é insuficiente contra alguns ataques [Wright *et al.* 2002].

Os projetos do Freenet [Clarke *et al.* 2000] e do FreeHaven [Dingledine *et al.* 2000] enfocam o fornecimento de serviços de arquivo na Internet oferecendo anonimato para os provedores e usuários dos arquivos. Ross Anderson propôs o Eternity Service [Anderson 1996], um serviço de armazenamento que proporciona garantias de longo prazo da disponibilidade dos dados, resistência a todos os tipos de perda acidental de dados e ataques de negação de serviço. Ele baseia a necessidade de tal serviço na observação de que, enquanto a publicação é um estado permanente para informações impressas – é praticamente impossível excluí-la, uma vez que tenha sido publicada e distribuída para milhares de bibliotecas em diversas organizações e jurisdições pelo mundo – as publicações eletrônicas não podem obter facilmente o mesmo nível de resistência à censura ou supressão. Anderson aborda os requisitos técnicos e econômicos para garantir a integridade do armazenamento, e também aponta que o anonimato é frequentemente um requisito fundamental para a persistência da informação, pois proporciona a melhor defesa contra contestações legais ou ações ilegais, como subornos ou ataques contra os criadores, proprietários ou mantenedores dos dados.

Lições aprendidas com o Napster • O Napster demonstrou a viabilidade da construção de um serviço de larga escala útil, que depende quase totalmente de dados e computadores pertencentes a usuários normais da Internet. Para evitar o esgotamento dos recursos computacionais de usuários individuais (por exemplo, o primeiro usuário a oferecer uma música muito procurada) e suas conexões de rede, o Napster considerava uma distância, a localidade da rede (quantidade de nós intermediários entre o cliente e o servidor) para alocar um servidor a um cliente que estivesse solicitando a música. Esse mecanismo simples de distribuição de carga permitia que o serviço mudasse de escala para satisfazer as necessidades de grandes números de usuários.

Limitações: o Napster usava um índice unificado (replicado) de todos os arquivos de música disponíveis. Para a aplicação em questão, o requisito da consistência entre as réplicas não era fundamental; portanto, isso não atrapalhava o desempenho, mas, para muitas aplicações, constituiria uma limitação. A não ser que o caminho de acesso para os objetos de dados seja distribuído, a descoberta e o endereçamento de objetos provavelmente se tornam um gargalo.

Dependências de aplicação: O Napster tirava proveito, de outras maneiras, das características especiais da aplicação para a qual foi projetado:

- Os arquivos de música nunca são atualizados, evitando qualquer necessidade de tornar todas as réplicas dos arquivos consistentes após as atualizações.
- Nenhuma garantia é exigida com relação à disponibilidade de arquivos individuais – se um arquivo de música estiver temporariamente indisponível, ele poderá ser baixado posteriormente. Isso reduz o requisito da confiança dos computadores individuais e de suas conexões com a Internet.

10.3 Middleware para peer-to-peer

Um problema importante no projeto de aplicativos *peer-to-peer* é o fornecimento de um mecanismo para permitir aos clientes acessarem recursos de dados rapidamente e de maneira segura, quando eles estão localizados por toda a rede. O Napster mantinha, para

esse propósito, um índice unificado dos arquivos disponíveis, fornecendo os endereços de rede de seus computadores. Os sistemas *peer-to-peer* de armazenamento de arquivos de segunda geração, como o Gnutella e o Freenet, empregam índices particionados e distribuídos, mas os algoritmos usados são específicos para cada sistema.

Esse problema de localização existia em vários serviços anteriores ao paradigma *peer-to-peer*. Por exemplo, o Sun NFS trata dessa necessidade com a ajuda de uma camada de abstração de sistema de arquivos virtual, em cada cliente, o qual aceita pedidos para acessar arquivos armazenados em vários servidores, em termos de referências de arquivo virtuais (isto é, *v-nodes*, veja a Seção 12.3). Essa solução conta com um volume de configuração prévia em cada cliente substancial e com a necessidade de intervenção manual quando os padrões de distribuição de arquivo ou o servidor muda. Claramente, ela não tem capacidade de escala além de um serviço gerenciado por uma única organização. O AFS (Seção 12.4) tem propriedades semelhantes.

Os sistemas de *middleware peer-to-peer* são projetados especificamente para atender à necessidade da distribuição automática, e da subsequente localização, dos objetos distribuídos gerenciados por sistemas e aplicativos *peer-to-peer*.

Requisitos funcionais • A função de um *middleware peer-to-peer* é simplificar a construção de serviços implementados em muitas máquinas, em uma rede amplamente distribuída. Para obter isso, ele deve permitir aos clientes localizarem e se comunicarem com qualquer recurso individual disponibilizado para um serviço, mesmo que os recursos estejam amplamente distribuídos entre as máquinas. Outros requisitos importantes incluem a capacidade de adicionar novos recursos e removê-los à vontade, e de adicionar máquinas no serviço e removê-las. Sendo um tipo de *middleware*, o *middleware peer-to-peer* deve oferecer uma interface de programação simples para programadores de aplicações, que seja independente dos tipos de recurso distribuído manipulados pelo aplicativo.

Requisitos não funcionais • Para funcionar eficientemente, o *middleware peer-to-peer* também deve tratar dos seguintes requisitos não funcionais [cf. Kubiatowicz 2003]:

Escalabilidade global: um dos objetivos dos aplicativos *peer-to-peer* é explorar os recursos de *hardware* de grandes quantidades de máquinas conectadas na Internet. Portanto, o *middleware peer-to-peer* deve ser projetado de modo a suportar aplicativos que acessam milhões de objetos em dezenas ou centenas de milhares de computadores.

Balanceamento da carga: o desempenho de qualquer sistema projetado para explorar um grande número de computadores depende da distribuição balanceada da carga de trabalho entre eles. Para os sistemas que estamos considerando, isso será obtido por um posicionamento aleatório dos recursos, junto ao uso de réplicas dos recursos muito utilizados.

Otimização das interações locais entre peers vizinhos: a “distância da rede” entre nós que interagem tem um impacto significativo sobre a latência das interações individuais, como as requisições de cliente para acesso aos recursos. As cargas do tráfego da rede também sofrem esse impacto. O *middleware* deve ter como objetivo colocar os recursos próximos dos nós que mais os acessam.

Acomodar a disponibilidade altamente dinâmica dos computadores: a maioria dos sistemas *peer-to-peer* é construída a partir de computadores que estão livres para entrar ou sair do sistema a qualquer momento. Os computadores e os segmentos de rede usados nos sistemas *peer-to-peer* não pertencem, nem são gerenciados, por nenhuma autoridade única; e nem sua confiabilidade, nem sua participação

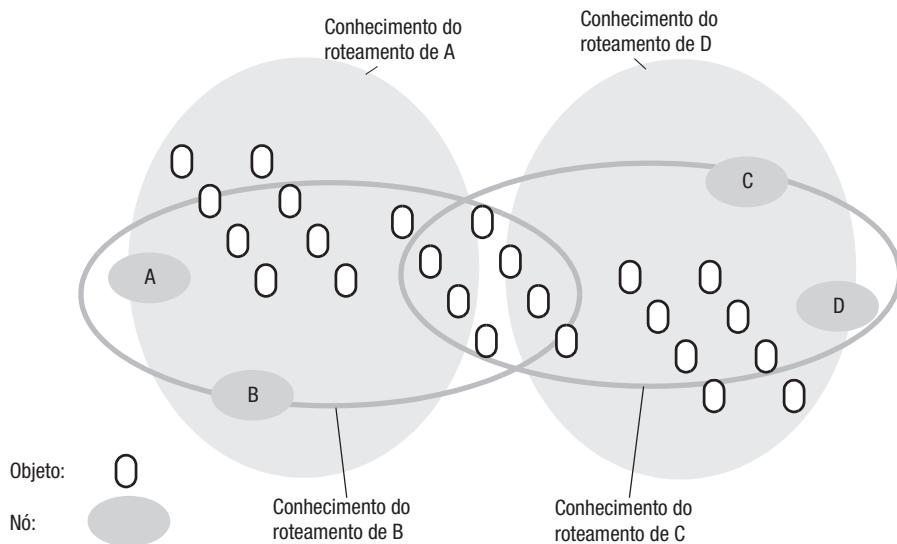


Figura 10.3 Distribuição das informações com sobreposição de roteamento.

contínua no abastecimento de um serviço são garantidas. Um desafio importante para os sistemas *peer-to-peer* é fornecer um serviço confiável, a despeito desses fatos. Quando computadores entram no sistema, eles devem ser integrados nele, e a carga deve ser redistribuída para explorar seus novos recursos. Quando eles saem do sistema, voluntária ou involuntariamente, o sistema deve detectar sua saída e redistribuir sua carga e os recursos.

Estudos de aplicativos e sistemas *peer-to-peer*, como o Gnutella e o Overnet, têm mostrado uma reviravolta considerável dos computadores participantes [Saroiu *et al.* 2002, Bhagwan *et al.* 2003]. Para o sistema de compartilhamento de arquivos *peer-to-peer* Overnet, com 85.000 computadores ativos em toda a Internet, Bhagwan *et al.* mediram uma duração média de sessão de 135 minutos (e uma mediana de 79 minutos) para uma amostra aleatória de 1.468 máquinas em um período de 7 dias, com 260 a 650 dos 1.468 computadores disponíveis para o serviço a qualquer momento. (Uma sessão representa um período durante o qual uma máquina está disponível, antes de ser voluntária, ou inevitavelmente, desconectada.)

Por outro lado, pesquisadores da Microsoft mediram uma duração de sessão de 37,7 horas para uma amostra aleatória de 20.000 máquinas conectadas na rede corporativa daquela empresa, com 14.700 a 15.600 máquinas disponíveis para o serviço a qualquer dado momento [Castro *et al.* 2003]. Essas medidas são baseadas em um estudo de viabilidade para o sistema de arquivo *peer-to-peer* Farsite [Bolosky *et al.* 2000]. A enorme disparidade entre os valores obtidos nesses estudos pode ser atribuída principalmente às diferenças no comportamento e no ambiente de rede entre os usuários de Internet individuais e os usuários de uma rede corporativa, como a da Microsoft.

Segurança dos dados em um ambiente com confiança heterogênea: em sistemas de escala global, com computadores participantes de diversos proprietários, a confiança deve ser estabelecida pelo uso de mecanismos de autenticação e criptografia, para garantir a integridade e a privacidade da informação.

Anonimato, capacidade de negação e resistência à censura: observamos (no quadro anterior) que o anonimato dos proprietários e destinatários dos dados é uma preocupação legítima em muitas situações que exigem resistência à censura. Um requisito relacionado é o de que os computadores que contêm dados devem ser capazes de negar a responsabilidade por contê-los ou fornecê-los, de forma plausível. O uso de grandes números de máquinas nos sistemas *peer-to-peer* pode ser útil na obtenção dessas propriedades.

Portanto, o projeto de uma camada de *middleware* para suportar sistemas *peer-to-peer* em escala global é um problema difícil. Os requisitos de escalabilidade e disponibilidade tornam impraticável manter um banco de dados em todos os nós clientes fornecendo as localizações de todos os recursos (objetos) de interesse.

O conhecimento das localizações dos objetos deve ser particionado e distribuído por toda a rede. Cada nó se torna responsável por manter o conhecimento detalhado das localizações dos nós e objetos em uma parte do espaço de nomes, assim como um conhecimento geral da topologia do espaço de nomes inteiro (Figura 10.3). Um alto grau de replicação desse conhecimento é necessário para garantir a segurança em face da disponibilidade volátil dos computadores e da conectividade intermitente da rede. Nos sistemas que vamos descrever a seguir, fatores de replicação de até 16 são normalmente usados.

10.4 Sobreposição de roteamento

O desenvolvimento de um *middleware* que atenda aos requisitos anteriores é um tópico de pesquisa ativa, e vários sistemas importantes de *middlewares* já surgiram. Neste capítulo, descreveremos dois deles em detalhes.

Um algoritmo distribuído conhecido como *sobreposição de roteamento (routing overlay)* assume a responsabilidade por localizar nós e objetos. O nome denota o fato de que o *middleware* assume a forma de uma camada que é responsável por direcionar requisições de qualquer cliente para um *host* que contenha o objeto para o qual a requisição é endereçada. Os objetos de interesse podem ser alocados e, subsequentemente, movidos para qualquer nó da rede, sem envolvimento do cliente. Ele é chamado de sobreposição porque implementa um mecanismo de roteamento na camada de aplicação que é completamente separado de todos os outros mecanismos de roteamento implantados em nível da rede, como o roteamento IP. Essa estratégia de gerenciamento e localização de objetos replicados foi analisada pela primeira vez e mostrou-se eficaz para redes envolvendo muitos nós, em um artigo inédito de Plaxton *et al.* [1997].

A sobreposição de roteamento garante que qualquer nó possa acessar qualquer objeto por meio do roteamento de cada requisição por uma sequência de nós, explorando o conhecimento existente em cada um deles para localizar o objeto de destino. Os sistemas *peer-to-peer* normalmente armazenam várias réplicas dos objetos para garantir disponibilidade. Neste caso, a sobreposição de roteamento mantém o conhecimento da localização de todas as réplicas disponíveis e distribui as requisições para o nó ativo mais próximo (isto é, um que não tenha falhado) que tenha uma cópia do objeto relevante.

Os GUIDs usados para identificar nós e objetos são um exemplo de nomes “puros”, referidos na Seção 13.1.1, também conhecidos como *identificadores opacos*, pois não revelam nada sobre a localização dos objetos a que se referem.

A principal tarefa de uma sobreposição de roteamento é a seguinte:

```
put(GUID, dados)
Os dados são armazenados em réplicas em todos os nós responsáveis pelo objeto identificado pelo
GUID.

remove(GUID)
Exclui todas as referências para o GUID e para os dados associados.

value = get(GUID)
Os dados associados ao GUID são recuperados de um dos nós responsáveis por eles.
```

Figura 10.4 Interface de programação básica para uma tabela de resumo distribuída (DHT), conforme implementada pela API PAST sobre o Pastry.

Roteamento de requisição para objetos: um cliente que queira invocar uma operação sobre um objeto envia uma requisição incluindo o GUID do objeto para a sobreposição de roteamento, a qual direciona a requisição para um nó em que resida uma réplica do objeto.

A sobreposição de roteamento também deve executar outras tarefas:

Inserção de objetos: um nó que queira tornar um novo objeto disponível para um serviço *peer-to-peer* calcula um GUID para o objeto e o anuncia para a sobreposição de roteamento, a qual, então, garante que o objeto possa ser acessado por todos os outros clientes.

Remoção de objetos: quando os clientes solicitam a remoção de objetos do serviço, a sobreposição de roteamento deve torná-los indisponíveis.

Adição e remoção de nós: os nós (isto é, os computadores) podem entrar e sair do serviço. Quando um nó entra no serviço, a sobreposição de roteamento faz preparativos para que ele assuma parte das responsabilidades dos outros nós. Quando um nó sai (voluntariamente, ou como resultado de uma falha de sistema ou da rede), suas responsabilidades são distribuídas entre os outros nós.

O GUID de um objeto é calculado a partir de todo o estado do objeto, ou de parte dele, usando-se uma função que produz um valor que tem probabilidade muito alta de ser exclusivo. A exclusividade é verificada procurando-se outro objeto com o mesmo GUID. Uma função de resumo (*hashing*), como SHA-1 (veja a Seção 11.4.3), é usada para gerar o GUID a partir do valor do objeto. Como esses identificadores distribuídos aleatoriamente são usados para determinar a colocação de objetos e para recuperá-los, às vezes os sistemas de sobreposição de roteamento são descritos como *tabelas de resumo distribuídas* (*DHT, Distributed Hashing Table*) e isso se reflete pela forma mais simples de API usada para acessá-los, como se vê na Figura 10.4. Com essa API, a operação *put()* é usada para enviar um item de dados para ser armazenado junto a seu GUID. A camada DHT assume a responsabilidade por escolher um local para ele, armazená-lo (com réplicas para garantir a disponibilidade) e fornecer acesso a ele por meio da operação *get()*.

Uma forma ligeiramente mais flexível de API é fornecida por uma camada de *localização e roteamento de objeto distribuído* (*DOLR, Distributed Object Location and Routing*), como se vê na Figura 10.5. Com essa interface, os objetos podem ser armazenados em qualquer lugar, e a camada DOLR é responsável por manter um mapeamento entre identificadores de objeto (GUIDs) e os endereços dos nós nos quais estão localizadas as réplicas dos objetos. Os objetos podem ser replicados e armazenados com o mes-

publish(GUID)

O *GUID* pode ser calculado a partir do objeto (ou de alguma parte dele, por exemplo, seu nome).

Esta função faz o nó efetuar uma operação *publish* no nó para o objeto correspondente ao *GUID*.

unpublish(GUID)

Torna o objeto correspondente ao *GUID* inacessível.

sendToObj(msg, GUID, [n])

Seguindo o paradigma orientado a objetos, uma mensagem de invocação é enviada a um objeto para acessá-lo. Poderia ser uma requisição para abrir uma conexão TCP para transferência de dados, ou para retornar uma mensagem contendo todo o estado do objeto, ou parte dele. O parâmetro opcional final *[n]*, se estiver presente, solicita a distribuição da mesma mensagem para *n* réplicas do objeto.

Figura 10.5 Interface de programação básica para localização e roteamento de objeto distribuído (DOLR), conforme implementada pelo Tapestry.

mo *GUID* em diferentes nós, e a sobreposição de roteamento assume a responsabilidade de rotear as requisições para a réplica mais próxima disponível.

Com o modelo DHT, um item de dados com *GUID* *X* é armazenado no nó cujo *GUID* é numericamente mais próximo a *X*, e nos *r* nós com *GUIDs* numericamente mais próximos a ele, onde *r* é um fator de replicação escolhido para garantir uma probabilidade de disponibilidade muito alta. Com o modelo DOLR, as localizações das réplicas de objetos de dados são decididas fora da camada de roteamento, e o endereço de nó de cada réplica é notificado para o DOLR com a operação *publish()*.

As interfaces das Figuras 10.4 e 10.5. são baseadas em um conjunto de representações abstratas propostas por Dabek *et al.* [2003] para mostrar que a maioria das implementações de sobreposição de roteamento *peer-to-peer* desenvolvidas até agora fornece funcionalidade muito parecida.

A pesquisa no projeto de sistemas de sobreposição de roteamento começou em 2000 e rendeu frutos em 2005, com o desenvolvimento e a avaliação de vários protótipos bem-sucedidos. As avaliações demonstraram que seu desempenho e confiança são adequados para uso em muitos ambientes de produção. Na próxima seção, descreveremos dois deles em detalhes: o Pastry, que implementa uma API de tabela de resumo distribuída (DHT) semelhante àquela apresentada na Figura 10.4, e o Tapestry, que implementa uma API semelhante àquela mostrada na Figura 10.5. Tanto o Pastry como o Tapestry empregam um mecanismo de roteamento conhecido como *roteamento baseado em prefixo*, para determinar as rotas para a distribuição de mensagens com base nos valores dos *GUIDs* para os quais elas são endereçadas. O roteamento baseado em prefixo restringe a busca do próximo nó ao longo da rota, aplicando uma máscara binária que seleciona um número cada vez maior de dígitos hexadecimais do *GUID* de destino, após cada etapa (*hop*). (Essa técnica também é empregada no CIDR, para IP, descrito em linhas gerais na Seção 3.4.3.)

Foram desenvolvidos outros esquemas de roteamento que exploram diferentes medidas de distância para restringir a busca em etapas do próximo destino. O Chord [Stoica *et al.* 2001] baseia a escolha na diferença numérica entre os *GUIDs* do nó selecionado e do nó de destino. O CAN [Ratnasamy *et al.* 2001] usa a distância em um hiperespaço *d*-dimensional, no qual os nós são colocados. O Kademia [Maymounkov e Mazieres 2002] usa a operação XOR dos pares de *GUIDs* como métrica para a distância entre os nós. Como a operação XOR é simétrica, o Kademia pode manter as tabelas de roteamen-

to dos participantes de forma muito simples, pois eles sempre recebem requisições dos mesmos nós contidos em suas tabelas de roteamento.

Os GUIDs não são legíveis para seres humanos; portanto, os aplicativos clientes devem obter os GUIDs dos recursos de interesse por meio de alguma forma de serviço de indexação, usando nomes legíveis para seres humanos ou pedidos de busca. De preferência, esses índices também são armazenados de uma maneira *peer-to-peer* para superar a deficiência dos índices centralizados, evidenciada pelo Napster. Contudo, nos casos simples, como música ou publicações, disponíveis para download *peer-to-peer*, eles podem ser simplesmente indexados nas páginas Web (cf. BitTorrent [Cohen 2003]). No BitTorrent, uma pesquisa de índice Web leva a um arquivo *stub* contendo detalhes sobre o recurso desejado, incluindo seu GUID e o URL de um *rastreador* (*tracker*) – um computador que contém uma lista atualizada dos endereços de rede dos provedores que querem fornecer o arquivo (consulte o Capítulo 20 para mais detalhes sobre o protocolo BitTorrent).

A descrição precedente em relação a sobreposição de roteamento provavelmente terá levantado dúvidas na cabeça do leitor, sobre seu desempenho e confiabilidade. As respostas para essas perguntas surgirão das descrições dos sistemas práticos de sobreposição de roteamento, na próxima seção.

10.5 Estudos de caso: Pastry, Tapestry

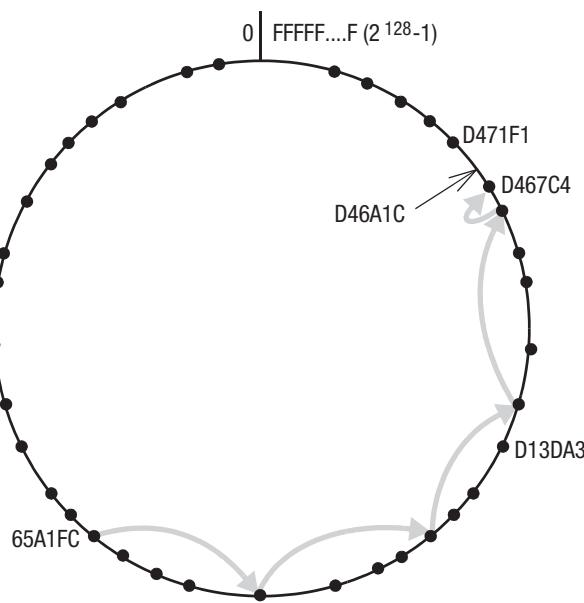
A estratégia do roteamento baseado em prefixo foi adotada pelo Pastry e pelo Tapestry. O Pastry tem um projeto simples, porém eficiente, que o torna um bom primeiro exemplo para estudarmos em detalhes. O Pastry é a infraestrutura de roteamento de mensagens implantada em várias aplicações, incluindo o PAST [Druschel e Rowstron 2001], um sistema de armazenamento de arquivos em repositório (imutável), implementado como uma tabela de resumo distribuída (DHT) com a API da Figura 10.4, e o Squirrel, um serviço *peer-to-peer* para uso de cache Web, descrito na Seção 10.6.1.

O Tapestry é a base do sistema de armazenamento OceanStore, que descreveremos na Seção 10.6.2. Ele tem uma arquitetura mais complexa do que o Pastry, pois seu objetivo é suportar uma variedade mais ampla de estratégias de localidade. Descreveremos isso na Seção 10.5.2, comparando com o Pastry.

Veremos também estratégias não estruturadas alternativas, na Seção 10.5.3, examinando em detalhes o estilo de sobreposição adotado pelo Gnutella.

10.5.1 Pastry

O Pastry [Rowstron e Druschel 2001, Castro *et al.* 2002a, [FreePastry project 2004](#)] é uma sobreposição de roteamento com as características que descrevemos na Seção 10.4. Todos os nós e objetos que podem ser acessados por meio do Pastry recebem GUIDs de 128 bits. Os GUIDs dos nós são calculados por meio da aplicação de uma função de resumo segura (como SHA-1, veja a Seção 11.4.3) tendo como argumento a chave pública fornecida para cada nó. Para objetos como arquivos, o GUID é calculado por meio da aplicação de uma função de resumo segura no nome do objeto, ou em alguma parte do estado armazenado do objeto. O GUID resultante tem as propriedades normais dos valores de resumo seguros – isto é, eles são distribuídos aleatoriamente no intervalo de 0 a $2^{128}-1$. Eles não fornecem nenhum indício sobre o valor a partir do qual foram calculados, e conflitos entre GUIDs de diferentes nós, ou objetos, são extremamente improváveis. (Nesse evento improvável, o Pastry o detecta e adota uma ação corretiva.)



Os pontos representam nós ativos. O espaço é considerado circular: o nó 0 é adjacente ao nó ($2^{128}-1$). O diagrama ilustra o roteamento de uma mensagem do nó 65A1FC para D46A1C, usando apenas informações dos nós folhas, pressupondo conjuntos de folhas de tamanho 8 ($l = 4$). Esse é um tipo de roteamento muito limitado, que tem uma capacidade de escalabilidade muito baixa. Ele não é usado na prática.

Figura 10.6 O roteamento circular, puro, é correto, mas ineficiente.

Em uma rede com N nós participantes, o algoritmo de roteamento do Pastry direcionará corretamente uma mensagem endereçada para qualquer GUID em $O(\log N)$ etapas. Se o GUID identifica um nó correntemente ativo, a mensagem é enviada para esse nó; caso contrário, ela é enviada para o nó ativo cujo GUID é numericamente mais próximo a ele. Os nós ativos assumem a responsabilidade pelo processamento de requisições endereçadas para todos os objetos em sua vizinhança numérica.

As etapas de roteamento envolvem o uso de um protocolo de transporte subjacente (normalmente UDP), para transferir a mensagem para um nó do Pastry que esteja mais próximo de seu destino. No entanto, note que a proximidade referida aqui é um espaço totalmente artificial – o espaço dos GUIDs. O transporte real de uma mensagem pela Internet entre dois nós do Pastry pode exigir um número substancial de etapas de roteamento IP. Para minimizar o risco de caminhos de transporte desnecessariamente estendidos, o Pastry usa uma métrica de localidade baseada na distância da rede subjacente (como contagens de etapas – *hop* – ou medidas da latência de viagem de ida e volta) para selecionar os nós vizinhos apropriados, ao configurar as tabelas de roteamento usadas em cada nó.

Milhares de nós localizados em *sites* amplamente dispersos podem participar de uma sobreposição Pastry; ela é organizada de forma totalmente automática – quando novos nós entram na sobreposição, eles obtêm os dados necessários para construir uma tabela de roteamento e outros estados exigidos, a partir dos membros existentes nas $O(\log N)$ mensagens, onde N é o número de nós participantes na sobreposição. No caso de falha, ou saída de um nó, os nós restantes podem detectar sua ausência e reconfigurar

$p =$	Prefixos de GUID e tratadores de nós n correspondentes															
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	n	n	n	n	n	n		n								
1	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6F	6E	6F
	n	n	n	n	n		n									
2	650	651	652	653	654	655	656	657	658	659	65A	65B	65C	65D	65E	65F
	n	n	n	n	n	n	n	n	n	n		n	n	n	n	n
3	65A0	65A1	65A2	65A3	65A4	65A5	65A6	65A7	65A8	65A9	65AA	65AB	65AC	65AD	65AE	65AF
	n		n													

A tabela de roteamento está localizada em um nó cujo GUID começa com 65A1. Os dígitos estão em hexadecimal. As letras n representam pares [GUID, endereço IP] especificando o próximo salto (*hop*) a ser seguido pelas mensagens endereçadas aos GUIDs que correspondem a cada prefixo dado. As entradas sombreadas indicam que o prefixo corresponde ao GUID correto até o valor dado de p : a próxima linha abaixo, ou o conjunto de folhas, deve ser examinada para encontrar uma rota. Embora existam no máximo 32 linhas na tabela, apenas $\log_{16} N$ linhas serão preenchidas, em média, em uma rede com N nós ativos.

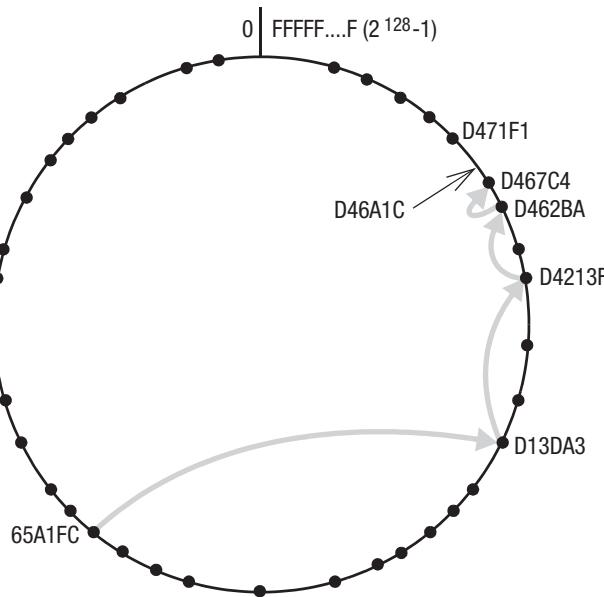
Figura 10.7 Primeiras quatro linhas de uma tabela de roteamento Pastry.

cooperativamente, para refletir as mudanças exigidas na estrutura de roteamento, em um número de mensagens semelhante.

Algoritmo de roteamento • O algoritmo de roteamento completo envolve o uso de uma tabela de roteamento em cada nó para direcionar eficientemente as mensagens, mas para os propósitos da explicação, descreveremos o algoritmo em dois estágios. O primeiro descreve uma forma simplificada do algoritmo, que direciona as mensagens correta, mas inefficientemente, sem uma tabela de roteamento; e o segundo estágio descreve o algoritmo de roteamento completo, que direciona uma requisição para qualquer nó em $O(\log N)$ mensagens.

Estágio I. Cada nó ativo armazena um *conjunto de folhas* – um vetor L (de tamanho $2l$) contendo os GUIDs e endereços IP dos nós cujos GUIDs são numericamente mais próximos nos dois lados de si próprio (l acima e l abaixo). Os conjuntos de folhas são mantidos pelo Pastry à medida que os nós se associam e saem. Mesmo após a falha de um nó, eles serão corrigidos dentro de um curto espaço de tempo. (A recuperação de falha será discutida a seguir.) Portanto, é uma constante do sistema Pastry que os conjuntos de folhas refletem um estado recente do sistema e que convirjam para o estado correto, em face de falhas de até alguma taxa máxima.

O espaço de GUID é tratado de forma circular: o vizinho inferior do GUID 0 é $2^{128}-1$. A Figura 10.6 apresenta uma visão dos nós ativos distribuídos nesse espaço de endereço circular. Como todo conjunto de folhas inclui os GUIDs e os endereços IP dos vizinhos imediatos do nó correto, um sistema Pastry com conjuntos de folhas corretos de tamanho de pelo menos 2 pode direcionar mensagens para qualquer



Roteamento de uma mensagem do nó 65A1FC para D46A1C. Com a ajuda de uma tabela de roteamento adequadamente preenchida, a mensagem pode ser enviada em $\sim \log_{16}(N)$ saltos.

Figura 10.8 Exemplo de roteamento Pastry.

GUID de maneira trivial, como segue. Qualquer nó A que receba uma mensagem M , com endereço de destino D , direciona a mensagem comparando D com seu próprio GUID A e com cada um dos GUIDs em seu conjunto de folhas; a seguir, encaminha M para o nó dentre eles que esteja numericamente mais próximo a D .

A Figura 10.6 ilustra isso para um sistema Pastry com $l = 4$. (Em instalações típicas reais do Pastry, $l = 8$.) Com base na definição de conjuntos de folhas, podemos concluir que, a cada etapa, M é encaminhada para um nó que está mais próximo a D do que o nó corrente, e que esse processo finalmente enviará M para o nó ativo mais próximo a D . Porém, tal esquema de roteamento é claramente muito inefficiente, exigindo $\sim N/2l$ saltos para enviar uma mensagem em uma rede com N nós.

Estágio II. A segunda parte de nossa explicação descreve o algoritmo Pastry completo e mostra como um roteamento eficiente pode ser obtido com a ajuda de tabelas de roteamento.

Cada nó Pastry mantém uma tabela de roteamento estruturada em árvore, fornecendo GUIDs e endereços IP para um conjunto de nós espalhados pela faixa inteira de 2^{128} valores de GUID possíveis, com densidade de cobertura maior para os GUIDs numericamente mais próximos de si próprios.

A Figura 10.7 mostra a estrutura da tabela de roteamento para um nó específico, e a Figura 10.8 ilustra as ações do algoritmo de roteamento. A tabela de roteamento é estruturada como segue: os GUIDs são vistos como valores hexadecimais, e a tabela classifica os GUIDs com base em seus prefixos hexadecimais. A tabela tem tantas linhas quantos são os dígitos hexadecimais em um GUID; portanto, para o protótipo de sistema Pastry que estamos descrevendo, existem $128/4 = 32$ linhas.

Para manipular uma mensagem M endereçada para um nó D (onde $R[p,i]$ é o elemento na coluna i , linha p da tabela de roteamento):

1. *Se* ($L_{-1} < D < L_1$) { // o destino está dentro do conjunto de folhas ou é o nó corrente.
 2. Encaminha M para o elemento L_i do conjunto de folhas com GUID mais próximo a D ou o nó corrente A .
 3. } *senão* { // usa a tabela de roteamento para enviar M para um nó com um GUID mais próximo
 4. Localiza p , o comprimento do prefixo comum mais longo de D e A ; e i , o $(p+1)$ -ésimo dígito hexadecimal de D .
 5. *Se* ($R[p,i] \neq \text{null}$) encaminha M para $R[p,i]$ // direciona M para um nó com um prefixo comum mais longo.
 6. *senão* { // não existe nenhuma entrada na tabela de roteamento
 7. Encaminha M para qualquer nó em L , ou R , com um prefixo comum de comprimento p , mas com um GUID numericamente mais próximo.
- }
- }

Figura 10.9 Algoritmo de roteamento do Pastry.

Qualquer linha n contém 15 entradas – uma para cada valor possível do n -ésimo dígito hexadecimal, excluindo o valor no GUID do nó local. Cada entrada na tabela aponta, potencialmente, para um dos muitos nós cujos GUIDs têm o prefixo relevante.

O processo de roteamento em qualquer nó A utiliza a informação de sua tabela de roteamento R e o conjunto de folhas L para lidar com todas as requisições de uma aplicação e com todas as mensagens recebidas de outro nó, de acordo com o algoritmo mostrado na Figura 10.9.

Podemos ter certeza de que o algoritmo terá êxito no envio de M para seu destino, pois as linhas 1, 2 e 7 executam as ações mencionadas no Estágio I de nossa descrição anterior, e mostramos que esse é um algoritmo de roteamento completo, embora ineficiente. Os passos restantes são projetados para usar a tabela de roteamento para melhorar o desempenho do algoritmo, reduzindo o número de etapas ou saltos intermediários exigidos.

As linhas 4 e 5 entram em ação quando D não cai no intervalo numérico do conjunto de folhas do nó corrente e as entradas da tabela de roteamento relevante estão disponíveis. A seleção de um destino para a próxima etapa envolve a comparação dos dígitos hexadecimais de D com os de A (o GUID do nó corrente), da esquerda para a direita, para descobrir o comprimento p de seu prefixo comum mais longo. Esse comprimento é usado, então, como um deslocamento de linha, junto ao primeiro dígito não correspondente de D , como deslocamento de coluna, para acessar o elemento exigido da tabela de roteamento. A construção da tabela garante que esse elemento (se não estiver vazio) contém o endereço IP de um nó cujo GUID tem $p+1$ dígitos de prefixo em comum com D .

A linha 7 é usada quando D cai fora do intervalo numérico do conjunto de folhas e não existe uma entrada relevante da tabela de roteamento. Esse caso é raro; ele surge apenas quando os nós falharam recentemente e a tabela ainda não foi atualizada. O algoritmo de roteamento é capaz de prosseguir, varrendo o conjunto de folhas e a tabela de roteamento, e encaminhando M para outro nó cujo GUID tenha p dígitos de prefixo correspondentes, mas seja numericamente mais próximo a D . Se esse nó estiver em L , então estamos seguindo o procedimento do Estágio I, ilustrado na Figura 10.6. Se ele estiver em R , então deve ser mais próximo a D do que qualquer nó em L ; portanto, estamos aperfeiçoando o Estágio I.

Integração de nós • Os nós novos usam um protocolo de associação para adquirir seus conteúdos de tabela de roteamento e conjunto de folhas e para notificar outros nós das alterações que devem fazer em suas tabelas. Primeiramente, o novo nó calcula um GUID conveniente (normalmente, aplicando a função de resumo SHA-1 na sua chave pública) e, então, estabelece contato com um nó Pastry vizinho. (Aqui, usamos o termo *vizinho* para nos referirmos à distância da rede; isto é, um pequeno número de saltos (*hops*) de rede, ou atraso de transmissão baixo; veja o quadro intitulado *Algoritmo do vizinho mais próximo*, a seguir.)

Suponha que o GUID do novo nó seja X e que o nó mais próximo que ele contata tenha o GUID A . O nó X envia uma mensagem de requisição *join* (associação) para A , fornecendo X como destino. A envia a mensagem *join*, via Pastry, da maneira normal. O Pastry direcionará a mensagem *join* para o nó existente cujo GUID é numericamente mais próximo a X ; vamos chamar esse nó de destino de Z .

A , Z e todos os nós (B, C, \dots) pelos quais a mensagem *join* é direcionada em seu caminho até Z acrescentam uma etapa adicional no algoritmo normal de roteamento Pastry que resulta na transmissão do conteúdo da parte relevante de suas tabelas de roteamento e conjuntos de folhas para X . O nó X examina essas informações e constrói sua própria tabela de roteamento e conjunto de folhas a partir delas, solicitando, se necessário, informações adicionais de outros nós.

Para ver como X constrói sua tabela de roteamento, note que a primeira linha da tabela depende do valor do GUID de X e que, para minimizar as distâncias de roteamento, a tabela deve ser construída para direcionar as mensagens por meio de nós vizinhos, quando possível. Contudo, A é vizinho de X ; portanto, a primeira linha da tabela de A é uma boa escolha inicial para a primeira linha da tabela de X , X_0 . Por outro lado, a tabela de A provavelmente não é relevante para a segunda linha X_1 , pois os GUIDs de X e de A podem não compartilhar o mesmo primeiro dígito hexadecimal. Porém, o algoritmo de roteamento garante que os GUIDs de X e de B compartilham o mesmo primeiro dígito, e isso significa que a segunda linha da tabela de roteamento de B , B_1 , é um valor inicial conveniente para X_1 . Analogamente, C_2 é conveniente para X_2 , e assim por diante.

Além disso, recordando as propriedades dos conjuntos de folhas, note que, como o GUID de Z é numericamente mais próximo ao de X , o conjunto de folhas de X deve ser semelhante ao de Z . Na verdade, o conjunto de folhas ideal de X será diferente do de Z apenas por um membro. Portanto, o conjunto de folhas de Z é aceito como uma aproximação inicial adequada que será otimizada por meio da interação com seus vizinhos, conforme descrito a seguir, na seção que trata de tolerância a falhas.

Finalmente, uma vez que X tiver construído seu conjunto de folhas e sua tabela de roteamento, da maneira descrita anteriormente, ele enviará seus conteúdos para todos os nós identificados no conjunto de folhas e na tabela de roteamento, e eles ajustarão suas próprias tabelas para incorporar o novo nó. A tarefa inteira de incorporação de um novo nó na infraestrutura Pastry exige a transmissão de $O(\log N)$ mensagens.

Algoritmo do vizinho mais próximo

O novo nó deve ter o endereço de pelo menos um nó Pastry existente, mas pode não ser um vizinho próximo. Para garantir que nós mais próximos sejam conhecidos, o Pastry inclui um algoritmo do “vizinho mais próximo”. Isso é feito medindo-se recursivamente o atraso da viagem de ida e volta de uma mensagem de teste (*probe*), enviada periodicamente para cada membro do conjunto de folhas do nó Pastry mais próximo conhecido, no momento.

Falha ou saída de nós • Os nós na infraestrutura Pastry podem falhar, ou sair sem aviso. Um nó Pastry é considerado defeituoso quando seus vizinhos imediatos (no espaço de GUID) não podem mais se comunicar com ele. Quando isso ocorre, é necessário reparar os conjuntos de folhas que contêm o GUID do nó defeituoso.

Para reparar seu conjunto de folhas L , o nó que descobre a falha procura um nó ativo próximo ao nó defeituoso em L e solicita uma cópia do conjunto de folhas desse nó, L' . L' conterá uma sequência de GUIDs que sobrepõem parcialmente os que estão em L , incluindo um que tenha um valor apropriado para substituir o nó defeituoso. Outros nós da vizinhança são, então, informados da falha e executam um procedimento semelhante. Esse procedimento de reparo garante que os conjuntos de folhas sejam reparados, a não ser que l nós de numeração adjacente falhem simultaneamente.

Os reparos nas tabelas de roteamento são feitos quando descobertos. O roteamento de mensagens pode prosseguir com algumas entradas da tabela de roteamento que não estão mais ativas – tentativas de roteamento malsucedidas resultam no uso de uma entrada diferente da mesma linha de uma tabela de roteamento.

Localidade • A estrutura de roteamento Pastry é altamente redundante: existem muitas rotas entre cada par de nós. A construção das tabelas de roteamento tem como objetivo levar em conta essa redundância para reduzir os tempos de transmissão de mensagem reais, explorando as propriedades de localidade dos nós na rede de transporte subjacente (que normalmente é um subconjunto de nós na Internet).

Lembremos que cada linha em uma tabela de roteamento contém 16 entradas. As entradas da i -ésima linha fornecem os endereços de 16 nós, com GUIDs com $i-1$ dígitos hexadecimais iniciais correspondentes ao GUID do nó corrente, e um i -ésimo dígito que recebe cada um dos valores hexadecimais possíveis. Uma sobreposição Pastry preenchida adequadamente conterá muito mais nós do que podem caber em uma tabela de roteamento individual, e quando uma nova tabela de roteamento está sendo construída, é feita uma escolha para cada posição entre várias candidatas (extraídas das informações de roteamento fornecidas pelos outros nós), com base em um algoritmo *Proximity Neighbour Selection* (*seleção de vizinho por proximidade*) [Gummadi *et al.* 2003]. É usada uma métrica de localidade (número de próximos saltos IP ou latência medida) para comparar as candidatas, e o nó mais próximo disponível é escolhido. Como as informações disponíveis não são abrangentes, esse mecanismo não pode produzir roteamentos globalmente perfeitos, mas simulações têm mostrado que ele resulta em rotas que, em média, são apenas cerca de 30–50% mais longas do que a ótima.

Tolerância a falhas • Conforme descrito anteriormente, o algoritmo de roteamento Pastry presume que todas as entradas nas tabelas de roteamento e conjuntos de folhas se referem a nós ativos e funcionando corretamente. Todos os nós enviam *mensagens de pulsação* (isto é, mensagens enviadas em intervalos de tempo fixos para indicar que o remetente está vivo) para os nós vizinhos em seus conjuntos de folhas, mas as informações sobre nós defeituosos detectadas dessa maneira podem não ser disseminadas rápido o suficiente para eliminar erros de roteamento. Isso também não leva em conta nós mal-intencionados que podem tentar interferir no roteamento correto. Para superar esses problemas, espera-se que os clientes que dependam do envio confiável da mensagem empreguem um mecanismo de entrega pelo menos uma vez (veja a Seção 5.3.1), e a repitam várias vezes, na ausência de uma resposta. Isso fornecerá ao Pastry uma janela de tempo mais longa para detectar e reparar falhas de nó.

Para tratar de todas as falhas, ou nós mal-intencionados restantes, um pequeno grau de aleatoriedade é introduzido no algoritmo de seleção de rota descrito na Figura 10.9. Basicamente, o passo da linha 5 da Figura 10.9 é modificado para uma pequena pro-

porção de casos selecionados aleatoriamente para produzir um prefixo comum que seja menor do que o comprimento máximo. Isso resulta no uso de um roteamento extraído de uma linha anterior da tabela de roteamento, produzindo um roteamento menos perfeito, mas diferente da versão padrão do algoritmo. Com essa variação aleatória no algoritmo de roteamento, as retransmissões do cliente acabam por ter sucesso, mesmo na presença de um pequeno número de nós mal-intencionados.

Dependabilidade • Os autores do Pastry desenvolveram uma versão atualizada, chamada MSPastry [Castro *et al.* 2003], que usa o mesmo algoritmo de roteamento e métodos de gerenciamento de nós semelhantes, mas que inclui algumas medidas de dependabilidade adicionais e algumas otimizações de desempenho nos algoritmos de gerenciamento de nó.

As medidas de dependabilidade incluem o uso de mensagens de confirmação em cada salto no algoritmo de roteamento. Se o nó que está fazendo o envio não receber uma mensagem de confirmação após um tempo limite especificado, ele seleciona uma rota alternativa e retransmite a mensagem. O nó que deixou de enviar a confirmação é, então, registrado como uma suspeita de falha.

Para detectar nós defeituosos, cada nó Pastry envia periodicamente uma mensagem de pulsação para seu vizinho imediato à esquerda (isto é, com um GUID menor) no conjunto de folhas. Cada nó também registra o tempo da última mensagem de pulsação recebida de seu vizinho imediato à direita (com um GUID maior). Se o intervalo de tempo desde a última pulsação ultrapassar o tempo limite, o nó de detecção começa um procedimento de reparo que envolve entrar em contato com os nós restantes do conjunto de folhas, com uma notificação sobre o nó defeituoso e uma requisição de substitutos sugeridos. Mesmo no caso de várias falhas simultâneas, esse procedimento termina com todos os nós no lado do nó defeituoso tendo conjuntos de folhas que contêm os l nós ativos com os GUIDs mais próximos.

Vimos que o algoritmo de roteamento pode funcionar corretamente usando apenas um conjuntos de folhas; mas a manutenção das tabelas de roteamento é importante para o desempenho. Nós com suspeita de defeito nas tabelas de roteamento são examinados de maneira semelhante ao conjunto de folhas, e se eles deixam de responder, suas entradas na tabela de roteamento são substituídas por uma alternativa conveniente, obtida de um nó vizinho. Além disso, um protocolo de *fofoca* (*gossip*, ver Seção 18.4.1) simples é usado para trocar informações da tabela de roteamento periodicamente entre os nós, visando reparar entradas defeituosas e evitar a lenta deterioração das propriedades da localidade. O protocolo de *fofoca* é executado aproximadamente a cada 20 minutos.

Avaliação • Castro e seus colegas realizaram uma exaustiva avaliação de desempenho do MSPastry, tendo como objetivo determinar o impacto sobre o desempenho e a confiança da taxa de associação/saída de nós e dos mecanismos de dependabilidade associados [Castro *et al.* 2003].

A avaliação foi realizada com a execução do sistema MSPastry por meio de um simulador, funcionando em uma única máquina, que simulava uma grande rede de nós com a passagem de mensagens substituída por atrasos de transmissão simulados. A simulação modelava de forma realista o comportamento de associação/saída de nós e atrasos de transmissão de IP baseados em parâmetros de instalações reais.

Todos os mecanismos de dependabilidade do MSPastry foram incluídos, com intervalos realistas para mensagens de teste (*probe*) e pulsação. O trabalho de simulação foi validado pela comparação com medidas tiradas com o MSPastry executando uma carga de uma aplicação real em uma rede interna com 52 nós.

Resumimos aqui apenas os principais resultados.

Dependabilidade: com uma taxa de perda de mensagens de IP de 0%, o MSPastry deixou de enviar 1,5 requisições em 100.000 (supostamente devido à indisponibilidade de nós de destino) e todas as requisições que foram enviadas chegaram ao nó correto.

Com uma taxa de perda de mensagens de IP de 5%, o MSPastry perdeu cerca de 3,3 requisições em 100.000 e 1,6 requisições em 100.000 foram enviadas para o nó errado. O uso de confirmações por saltos intermediários no MSPastry garante que todas as mensagens perdidas, ou direcionadas para o lugar errado, sejam finalmente retransmitidas e cheguem ao nó correto.

Desempenho: a métrica usada para avaliar o desempenho do MSPastry é chamada de *penalidade de atraso relativa* (*RDP, Relative Delay Penalty*) [Chu *et al.* 2000] ou *distensão*. A RDP é uma medida direta do custo extra acarretado no emprego de uma camada de sobreposição de roteamento. É a relação entre o atraso médio no envio de uma requisição pela sobreposição de roteamento e no envio de uma mensagem semelhante entre os mesmos dois nós por meio de UDP/IP. Os valores de RDP observados para o MSPastry sob cargas simuladas variaram de ~1,8, com perda zero de mensagens de rede, a ~2,2, com 5% de perda de mensagens de rede.

Sobrecargas: a carga de rede extra gerada pelo *tráfego de controle* – mensagens envolvidas na manutenção de conjuntos de folhas e tabelas de roteamento – foi de menos de duas mensagens por minuto por nó. Tanto a RDP como o tráfego de controle aumentaram significativamente para sessões com duração de menos de cerca de 60 minutos, devido às sobrecargas de configuração iniciais.

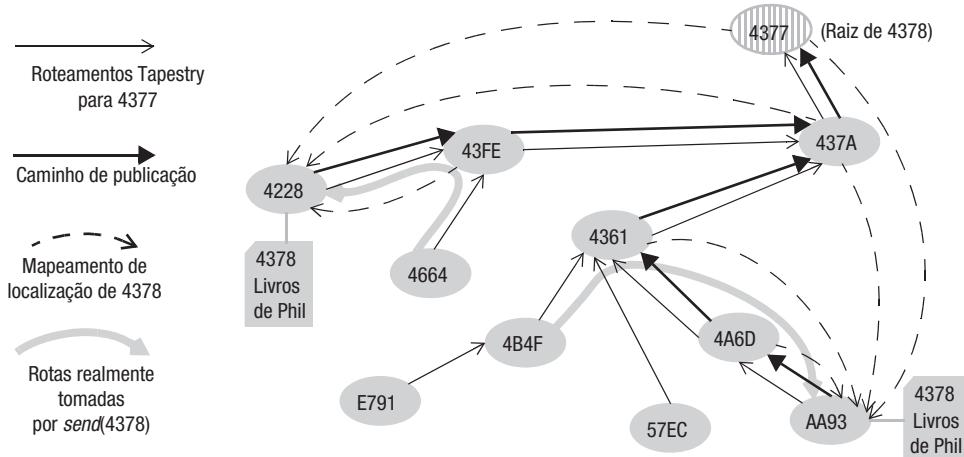
De modo geral, esses resultados mostram que podem ser construídas camadas de sobreposição de roteamento que obtêm bom desempenho e alta confiança com milhares de nós operando em ambientes reais. Mesmo com duração média de sessão menor do que 60 minutos e altas taxas de erro de rede, o sistema degrada pouco, continuando a fornecer um serviço eficiente.

Otimizando a latência de pesquisa de busca de sobreposição • Zhang *et al.* [2005a] mostraram que o desempenho da pesquisa de busca de uma classe importante de redes de sobreposição (incluindo Pastry, Chord e Tapestry) pode ser significativamente melhorado com a inclusão de um algoritmo de aprendizado simples, que mede as latências realmente experimentadas no acesso aos nós de sobreposição e, assim, modifica as tabelas de sobreposição de roteamento de modo incremental para otimizar as latências de acesso.

10.5.2 Tapestry

O Tapestry implementa uma tabela de resumo distribuída e direciona as mensagens para os nós com base nos GUIDs associados aos recursos, usando roteamento baseado em prefixo de maneira semelhante ao Pastry. No entanto, a API do Tapestry oculta das aplicações a tabela de resumo distribuída atrás da interface DOLR, como aquela que aparece na Figura 10.5. Os nós que contêm recursos usam a primitiva *publish(GUID)* para torná-los conhecidos do Tapestry; os proprietários de recursos continuam responsáveis por seu armazenamento. Os recursos replicados são publicados com o mesmo GUID em cada nó que contém uma réplica, resultando em várias entradas na estrutura de roteamento do Tapestry.

Isso proporciona às aplicações Tapestry uma flexibilidade adicional: elas podem colocar réplicas próximas (em distância da rede) aos usuários frequentes dos recursos para reduzir as latências e minimizar as cargas de rede, ou para garantir tolerância a falhas de rede e de nós. Contudo, essa distinção entre o Pastry e o Tapestry não é funda-



Réplicas do arquivo *Livros de Phil* ($G=4378$) são contidas nos nós 4228 e AA93. O nó 4377 é o nó raiz do objeto 4378. Os roteamentos do Tapestry mostrados são algumas das entradas nas tabelas de roteamento. Os caminhos de publicação mostram as rotas seguidas pelas mensagens de publicação contendo mapeamentos de localização armazenados em cache para o objeto 4378. Os mapeamentos de localização são usados subsequentemente para direcionar as mensagens enviadas para 4378.

Figura 10.10 Roteamento do Tapestry. De [Zhao et al. 2004].

mental: as aplicações Pastry podem obter flexibilidade semelhante fazendo os objetos associados aos GUIDs simplesmente agirem como *proxies* de objetos em nível de aplicação mais complexos, e o Tapestry pode ser usado para implementar uma tabela de resumo distribuída em termos de sua API DOLR [Dabek et al. 2003].

No Tapestry são usados identificadores de 160 bits para fazer referência tanto a objetos como aos nós que executam ações de roteamento. Os identificadores são *NodeIds*, quando se referem aos computadores que executam operações de roteamento, ou *GUIDs*, quando se referem aos objetos. Para qualquer recurso com GUID G existe um nó-raiz único com GUID R_G numericamente mais próximo a G . Os nós H que contêm réplicas de G ativam $publish(G)$ periodicamente para garantir que os nós recém-chegados saibam da existência de G . A cada invocação de $publish(G)$, uma mensagem de publicação é direcionada ao invocador para o nó R_G . Na recepção de uma mensagem de publicação, R_G insere (G, IP_H) , o mapeamento entre G e o endereço IP do nó que está fazendo o envio, em sua tabela de roteamento, e cada nó ao longo do caminho de publicação armazena o mesmo mapeamento na cache. Esse processo está ilustrado na Figura 10.10. Quando os nós contêm vários mapeamentos (G, IP) para o mesmo GUID, eles são ordenados pela distância da rede (tempo da viagem de ida e volta) até o endereço IP. Para objetos replicados, isso implica seleção da réplica do objeto mais próxima disponível como destino das mensagens subsequentes enviadas ao objeto.

Zhao et al. [2004] fornecem detalhes completos dos algoritmos de roteamento e do gerenciamento de tabelas de roteamento do Tapestry em face da chegada e saída de nós. Esse artigo inclui amplos dados de avaliação de desempenho, baseados em simulação de redes Tapestry de larga escala, mostrando que seu desempenho é semelhante ao do Pastry. Na Seção 10.6.2, descreveremos o sistema de armazenamento de arquivos Ocean-Store, que foi construído e implantado sobre o Tapestry.

	<i>Peer-to-peer estruturados</i>	<i>Peer-to-peer não estruturados</i>
<i>Vantagens</i>	Há garantia na localização de objetos (supondo que eles existam) e podem oferecer limites de tempo e complexidade nessa operação; sobrecarga de mensagens relativamente baixa.	Organizam a si próprios e são naturalmente resilientes às falhas de nó.
<i>Desvantagens</i>	Necessidade de manter as estruturas da sobreposição que são frequentemente complexas e que podem ser difíceis e dispendiosas de obter, especialmente em ambientes altamente dinâmicos.	Probabilísticos e, assim, não podem oferecer garantias absolutas sobre a localização de objetos; propensos à sobreulação de troca de mensagens excessiva, o que pode afetar a escalabilidade.

Figura 10.11 Sistemas *peer-to-peer* estruturados versus não estruturados.

10.5.3 De *peer-to-peer* estruturado para não estruturado

A discussão até aqui se concentrou exclusivamente nas estratégias conhecidas como *peer-to-peer estruturadas*. Nas estratégias estruturadas, existe uma política global generalizada governando a topologia da rede, o posicionamento de objetos na rede e as funções de roteamento ou pesquisa usadas para localizar objetos na rede. Em outras palavras, há uma estrutura de dados (distribuída) específica servindo de base para a sobreposição associada e um conjunto de algoritmos operando sobre essa estrutura de dados. Isso pode ser visto claramente nos exemplos do Pastry e do Tapestry, baseados na tabela de resumo distribuída subjacente e nas estruturas em anel associadas. Graças à estrutura imposta, esses algoritmos são eficientes e oferecem limites de tempo para a localização de objetos, mas ao custo da manutenção das estruturas associadas, muitas vezes em ambientes altamente dinâmicos.

Por causa dessa necessidade de manutenção, também foram desenvolvidas estratégias *peer-to-peer não estruturadas*. Nas estratégias não estruturadas, não há controle global sobre a topologia ou sobre o posicionamento de objetos dentro da rede. Em vez disso, a sobreposição é criada de maneira *ad hoc*, com cada nó que ingressa na rede seguindo algumas regras locais simples para estabelecer a conectividade. Em particular, ao ingressar, um nó estabelece contato com um conjunto de *vizinhos*, sabendo que os vizinhos também vão se conectar a outros e assim por diante, formando uma rede fundamentalmente descentralizada e que organiza a si própria; portanto, resiliente às falhas de nó. Para localizar determinado objeto é, então, necessário fazer uma busca na topologia de rede resultante; claramente essa estratégia não pode oferecer nenhuma garantia de que possa encontrar o objeto, sendo que o desempenho será imprevisível. Além disso, há o risco real de gerar tráfego de mensagens excessivo para localizar objetos.

Um resumo das vantagens relativas dos sistemas *peer-to-peer* estruturados e não estruturados é fornecido na Figura 10.11. É interessante pensar que, apesar dos inconvenientes aparentes dos sistemas *peer-to-peer* não estruturados, essa estratégia é dominante na Internet, particularmente no suporte para compartilhamento de arquivos *peer-to-peer* (com sistemas como Gnutella, FreeNet e BitTorrent adotando todos estratégias não estruturadas). Conforme será visto, foram feitos avanços significativos nesses sistemas para melhorar o desempenho das estratégias não estruturadas, e esse trabalho é importante, dada a quantidade de tráfego gerado por compartilhamento de arquivos *peer-to-peer* na Internet (por exemplo, um estudo feito nos anos 2008 e 2009 indicou que as

aplicações de compartilhamento de arquivos *peer-to-peer* foram responsáveis por cerca de 43% a 70% de todo tráfego da Internet, dependendo da parte do mundo considerada [www.ipoque.com]).

Estratégias de busca eficientes • No compartilhamento de arquivos *peer-to-peer*, todos os nós da rede oferecem arquivos para o ambiente maior. Conforme mencionado anteriormente, o problema de localizar um arquivo é mapeado em uma pesquisa de busca na rede inteira para localizar os arquivos adequados. Se fosse implementado ingenuamente, isso seria feito de modo a inundar a rede com requisições. É exatamente essa a estratégia adotada nas primeiras versões do Gnutella. Em particular, no Gnutella 0.4, cada nó encaminha uma requisição para cada um de seus vizinhos, os quais, por sua vez, passam isso para seus vizinhos e assim por diante, até que uma correspondência seja encontrada. Cada busca também era restrita, com um campo de tempo de vida (*time-to-live*) limitando o número de saltos. Quando o Gnutella 0.4 foi implantado, a conectividade média era em torno de 5 vizinhos por nó. Essa estratégia é simples, mas não escala e inunda a rede rapidamente com tráfego relacionado à pesquisa de busca.

Vários refinamentos foram desenvolvidos para pesquisa de busca em redes não estruturadas [Lv *et al.* 2002, Tsoumakos e Roussopoulos 2006], incluindo:

Pesquisa em anel expandida: nesta estratégia, o nó iniciador realiza uma série de buscas com valores cada vez maiores do campo de tempo de vida (*time-to-live*), reconhecendo que um número significativo de requisições será satisfeita localmente (especialmente se for acoplada a uma estratégia de replicação eficiente, conforme discutido a seguir).

Caminhadas aleatórias (random walks): com caminhadas aleatórias, o nó iniciador define vários caminhantes que seguem seus próprios trajetos aleatórios no grafo interconectado oferecido pela sobreposição não estruturada.

Fococa: nas estratégias de *fofoca*, um nó envia uma requisição para determinado vizinho com certa probabilidade e, assim, as requisições se propagam pela rede de maneira semelhante a um vírus em uma população (por isso, os protocolos de *fofoca* também são referidos como *protocolos epidêmicos*). A probabilidade pode ser fixa para determinada rede ou calculada dinamicamente com base na experiência anterior e/ou contexto atual. (Note que *fofoca* é uma técnica comum em sistemas distribuídos; mais aplicações podem ser encontradas nos Capítulos 6 e 18.)

Essas estratégias podem reduzir significativamente a sobrecarga de pesquisa de busca em redes não estruturadas e, assim, aumentar a escalabilidade dos algoritmos. Tais estratégias também são frequentemente suportadas por técnicas de *replicação* apropriadas. Replicando-se o conteúdo por vários pares, a probabilidade de descoberta eficiente de arquivos específicos é significativamente aumentada. As técnicas incluem replicação de arquivo inteiro e a dispersão de fragmentos de arquivos pela Internet – uma estratégia usada de maneira eficaz no BitTorrent, por exemplo, para reduzir a carga sobre qualquer par no *download* de arquivos grandes (consulte o Capítulo 20).

Estudo de caso: Gnutella • O Gnutella foi lançado originalmente em 2000 e, desde então, tem-se desenvolvido para ser um dos aplicativos de compartilhamento de arquivos *peer-to-peer* dominante e mais influente. Conforme mencionado anteriormente, de início o protocolo adotou uma estratégia de inundação bastante simples e que não escalava bem. Em resposta a isso, o Gnutella 0.6 introduziu várias modificações que melhoraram significativamente o desempenho de seu protocolo.

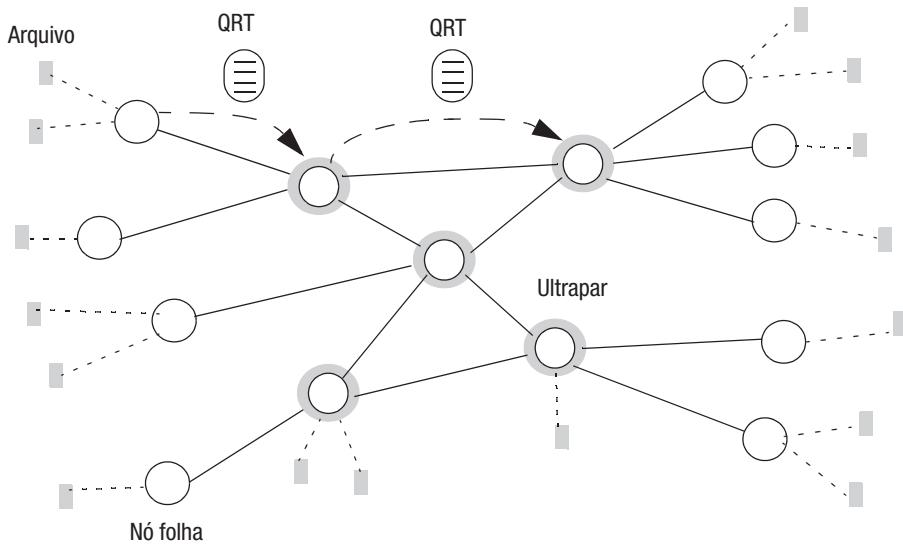


Figura 10.12 Principais elementos do protocolo Gnutella.

O primeiro aperfeiçoamento importante foi mudar de uma arquitetura *peer-to-peer* pura, em que todos os nós são iguais, para uma em que todos os pares ainda cooperam para oferecer o serviço, mas alguns nós, designados para ter recursos adicionais, são eleitos como *ultrapares* e formam a parte principal da rede, com os outros pares assumindo o papel de nós folha (ou folhas). As folhas se conectam com um pequeno número de ultrapares, os quais se conectam em grande quantidade com outros ultrapares (com mais de 32 conexões cada um). Isso reduz substancialmente o número máximo de saltos exigidos para uma pesquisa de busca exaustiva. Esse estilo de arquitetura *peer-to-peer* é referida como arquitetura híbrida e também é a estratégia adotada no Skype (conforme discutido na Seção 4.5.2).

O outro aprimoramento importante foi a introdução de um protocolo QRP (Query Routing Protocol), projetado para reduzir o número de consultas feitas pelos nós. O protocolo é baseado na troca de informações sobre os arquivos contidos nos nós e, assim, só encaminham as consultas para os caminhos nos quais o sistema acredita que vai haver um resultado positivo. Em vez de compartilhar informações sobre os arquivos diretamente, o protocolo produz um conjunto de números a partir do resumo feito nas palavras individuais presentes em um nome de arquivo. Por exemplo, um nome de arquivo como “Capítulo 10 sobre P2P” vai ser representado por quatro números, digamos <65, 47, 09, 76>. Um nó produz uma tabela QRT (Query Routing Table) contendo os valores de resumo representando cada um dos arquivos contidos nesse nó, a qual ele envia para todos os ultrapares associados. Então, os ultrapares produzem suas próprias Query Routing Tables com base em uma união de todas as entradas de todas as folhas conectadas, junto às entradas dos arquivos contidos nesse nó, e trocam isso com outros ultrapares conectados. Desse modo, os ultrapares podem determinar quais caminhos oferecem uma rota válida para determinada consulta, reduzindo significativamente o volume de tráfego desnecessário. Mais especificamente, um ultrapar encaminhará uma consulta para um nó se uma correspondência for encontrada (indicando que o nó tem o arquivo) e fará a mesma verificação antes de passá-la para outro ultrapar, se esse for o último salto (destino) para o arquivo. Note que, para evitar a sobrecar-

ga de ultrapares, os nós enviam uma consulta para um ultrapar por vez e, então, esperam um período de tempo especificado para ver se obtêm uma resposta positiva.

Por fim, uma consulta no Gnutella contém o endereço de rede do ultrapar iniciador, o que implica que, uma vez encontrado um arquivo, ele pode ser enviado diretamente para o ultrapar associado (usando UDP), evitando uma passagem inversa pelo grafo.

Os principais elementos associados ao Gnutella 0.6 estão resumidos na Figura 10.12.

10.6 Estudo de caso: Squirrel, OceanStore, Ivy

As camadas de sobreposição de roteamento descritas na seção anterior foram exploradas em várias experiências e cenários de utilização, e os aplicativos resultantes foram extensivamente avaliados. Escolhemos três deles para estudar melhor, o serviço de uso de cache Web Squirrel, baseado no Pastry, e os sistemas de armazenamento de arquivo OceanStore e Ivy.

10.6.1 Cache Web Squirrel

Os autores do Pastry desenvolveram o serviço *peer-to-peer* de cache Web Squirrel para utilização em redes locais de computadores pessoais [Iyer *et al.* 2002]. Em redes locais médias e grandes, o uso de cache Web normalmente é realizado com um computador servidor dedicado ou com um *cluster*. O sistema Squirrel executa a mesma tarefa, explorando armazenamento e recursos computacionais já disponíveis em computadores *desktop* da rede local. Primeiramente, daremos uma breve descrição geral da operação de um serviço de uso de cache Web e, em seguida, descreveremos em linhas gerais o projeto do Squirrel e examinaremos sua eficácia.

Uso de cache Web • Os navegadores Web geram pedidos HTTP *GET* para objetos de Internet, como páginas HTML, imagens, etc. Esses objetos podem ser providos a partir da cache de um navegador na máquina cliente, *cache Web proxy* (por um serviço sendo executado em outro computador na mesma rede local, ou em um nó vizinho na Internet), ou do servidor Web *de origem* – o servidor cujo nome de domínio é incluído nos parâmetros da requisição GET – dependendo de qual contenha uma cópia atualizada do objeto. As caches local e de *proxy* contêm, cada uma, um conjunto de objetos recentemente recuperados, organizados para uma pesquisa rápida pelo URL. Alguns objetos não podem ser colocados na cache, pois são gerados dinamicamente pelo servidor em resposta a cada requisição.

Quando a cache de um navegador, ou a cache Web do *proxy*, recebe uma requisição *GET*, existem três possibilidades: o objeto solicitado não pode ser atingido, há uma falta na cache ou o objeto é encontrado na cache. Nos dois primeiros casos, a requisição é encaminhada para o próximo nível, em direção ao servidor Web de origem. Quando o objeto solicitado é encontrado em uma cache, a atualidade da cópia colocada na cache deve ser testada.

Os objetos Web são armazenados em servidores Web e em servidores de cache com alguns valores adicionais de metadados, incluindo um caminho de tempo fornecendo a *data da última modificação T* e possivelmente um *tempo de vida t* ou uma *eTag* (um código de resumo calculado a partir do conteúdo de uma página Web). Esses itens de metadados são fornecidos pelo servidor de origem quando um objeto é retornado para um cliente.

Para objetos que têm um tempo de vida *t* associado, o objeto é considerado atual se *T+t* for posterior ao tempo corrente real. Para objetos sem tempo de vida, é usado um valor estimado para *t* (frequentemente de apenas alguns segundos). Se o resultado dessa avaliação de atualidade for positivo, o objeto colocado na cache será retornado para o

cliente, sem contato com o servidor Web de origem. Caso contrário, uma requisição *GET* condicional (*cGET*) será emitida para validação no próximo nível. Existem dois tipos básicos de requisição *cGET*: uma requisição *If-Modified-Since*, contendo a indicação de tempo da última modificação conhecida, e uma requisição *If-None-Match*, contendo uma *eTag* representando o conteúdo do objeto. Essa requisição *cGET* pode ser servida por outra cache Web ou pelo servidor de origem. Uma cache Web que recebe uma requisição *cGET*, e não tem uma cópia atualizada do objeto, encaminha a requisição para o servidor Web de origem. A resposta contém o objeto inteiro, ou uma mensagem *not-modified* se o objeto colocado na cache estiver intacto.

Quando um objeto recentemente modificado é recebido do servidor de origem, se for o caso, ele pode ser adicionado ao conjunto de objetos da cache local (removendo objetos mais antigos que ainda são válidos, se necessário), junto a um carimbo de tempo, um tempo de vida e uma *eTag*, se estiver disponível.

O esquema descrito anteriormente é a base da operação dos serviços de uso de cache Web de *proxies* centralizados, implantados na maioria das redes locais que suportam grandes números de clientes Web. Normalmente, as caches Web de *proxy* são implementadas como um processo *multithreaded*, executando em um único computador dedicado, ou um conjunto de processos executando em um *cluster* de computadores, e, em ambos os casos, exigem uma quantidade substancial de recursos computacionais dedicados.

Squirrel • O serviço de uso de cache Web Squirrel executa as mesmas funções, usando uma pequena parte dos recursos de cada computador cliente em uma rede local. A função de resumo segura SHA-1 é aplicada ao URL de cada objeto armazenado na cache para produzir um GUID Pastry de 128 bits. Como o GUID não é usado para validar seu conteúdo, ele não precisa ser baseado no conteúdo do objeto inteiro, como acontece nos outros aplicativos Pastry. Os autores do Squirrel baseiam sua justificativa para isso no “argumento do princípio fim-a-fim” (Seção 2.3.3), afirmando que a autenticidade de uma página Web pode ser comprometida em muitos pontos de sua jornada do servidor até o cliente; a autenticação de páginas armazenadas na cache acrescenta pouco em qualquer garantia global de autenticidade – o protocolo HTTPS (incorporando Transport Layer Security, Seção 11.6.3) deve ser usado para se obter uma garantia muito melhor para as interações que o exigem.

Na implementação mais simples do Squirrel – que se mostrou a mais eficiente –, o nó cujo GUID é numericamente mais próximo ao GUID de um objeto se torna o *nó de base* desse objeto, responsável por conter toda cópia do objeto colocada na cache.

Os nós clientes são configurados de forma a incluir um processo Squirrel local *proxy*, que assume a responsabilidade pelo armazenamento na cache local e remota de objetos Web. Se uma cópia atualizada de um objeto solicitado não está na cache local, o Squirrel direciona uma requisição *Get*, ou uma requisição *cGet* (quando existe uma cópia antiga do objeto na cache local), por meio do Pastry, para o nó de base. Se o nó de base tiver uma cópia atualizada, ele responderá diretamente para o cliente com uma mensagem *not-modified*, ou com uma cópia atualizada, conforme for apropriado. Se o nó de base tiver uma cópia antiga, ou não tiver nenhuma cópia do objeto, ele emitirá uma requisição *cGet* ou *Get* para o servidor de origem, respectivamente. Se o servidor de origem puder, responderá com uma mensagem *not-modified*, ou com uma cópia do objeto. No primeiro caso, o nó de base revalida sua entrada de cache e encaminha uma cópia do objeto para o cliente. No último caso, ele encaminha uma cópia do novo valor para o cliente e armazena uma cópia em sua cache local, caso o objeto possa ser colocado na cache.

Avaliação do Squirrel • O Squirrel foi avaliado por meio de simulação, usando cargas modeladas derivadas de traços existentes da atividade de caches Web de *proxies* centrali-

zados, em dois ambientes de trabalho reais dentro da Microsoft, um com 105 clientes ativos (em Cambridge) e o outro com mais de 36.000 (em Redmond). A avaliação comparou o desempenho de uma cache Web Squirrel com uma centralizada, sob três aspectos:

A redução na largura de banda externa total usada: a largura de banda externa total é inversamente relacionada à taxa de acertos, pois são apenas as faltas de cache que geram requisições para servidores Web externos. As taxas de acerto observadas para servidores de cache Web centralizados foram de 29% (para Redmond) e 38% (para Cambridge). Quando os mesmos registros de atividade foram usados para gerar uma carga simulada para a cache Squirrel, com cada cliente contribuindo com 100 MB de armazenamento em disco, foram obtidas taxas de acerto muito semelhantes, de 28% (Redmond) e 37% (Cambridge). Conclui-se que a largura de banda externa seria reduzida por uma proporção semelhante.

A latência percebida pelos usuários para acesso a objetos Web: o uso de uma sobreposição de roteamento resultou em várias transferências de mensagem (saltos de roteamento) na rede local para transmitir uma requisição de um cliente para o nó responsável por armazenar o objeto relevante (o nó de base) na cache. Os números médios de saltos de roteamento observados na simulação foram de 4,11 para enviar uma requisição GET, no caso de Redmond, e 1,8 no caso de Cambridge, enquanto apenas uma única transferência de mensagem é exigida para acessar um serviço de cache centralizado.

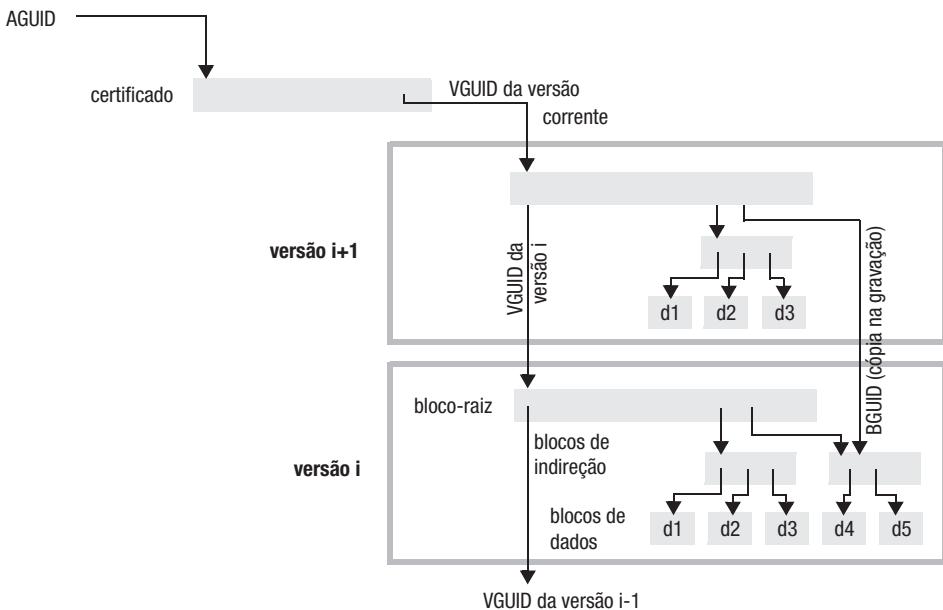
No entanto, as transferências locais levam apenas alguns milisegundos com um *hardware Ethernet* moderno, incluindo o tempo de configuração da conexão TCP, enquanto as transferências de mensagem TCP remotas na Internet exigem de 10 a 100 ms. Portanto, os autores do Squirrel argumentam que a latência para acesso a objetos encontrados em cache é superada pela latência muito maior do acesso a objetos não encontrados em cache, proporcionando uma experiência para o usuário semelhante àquela apresentada com uma cache centralizada.

A carga computacional e de armazenamento imposta sobre os nós clientes: em cada nó, o número médio de requisições de cache recebidas de outros nós durante o período inteiro da avaliação foi extremamente baixo, com apenas 0,31 por minuto (Redmond), indicando que a proporção global de recursos de sistema consumidos é extremamente baixa.

Com base nas medidas descritas anteriormente, os autores do Squirrel concluíram que seu desempenho é comparável ao de uma cache centralizada. O Squirrel obtém uma redução na latência observada para acesso à página Web próxima daquela que pode ser obtida por um servidor de cache centralizado, com uma cache dedicada de tamanho semelhante. A carga adicional imposta sobre os nós clientes é baixa e provavelmente será imperceptível para os usuários. O sistema Squirrel foi subsequentemente implantado como cache Web principal em uma rede local com 52 máquinas clientes, usando o Squirrel, e os resultados confirmaram suas conclusões.

10.6.2 Sistema de armazenamento de arquivos OceanStore

Os desenvolvedores do Tapestry projetaram e construíram um protótipo de sistema de armazenamento de arquivos *peer-to-peer*. Ao contrário do Past, ele suporta o armazenamento de arquivos mutáveis. O projeto OceanStore [Kubiatowicz *et al.* 2000, Kubiatowicz 2003, Rhea *et al.* 2001, Rhea *et al.* 2003] tem como objetivo fornecer um recurso de armazenamento persistente de grande escala, escalável por incrementos, para objetos



A versão $i+1$ foi atualizada nos blocos $d1, d2$ e $d3$. O certificado e os blocos-raízes incluem alguns metadados, não mostrados. Todas as setas não rotuladas são BGUIDs.

Figura 10.13 Organização do armazenamento de objetos do OceanStore.

de dados mutáveis, com persistência a longo prazo e confiabilidade em um ambiente de recursos de rede e computacionais em constante mudança. O OceanStore se destina a ser usado em uma variedade de aplicações, incluindo a implementação de um serviço de arquivos do tipo do NFS, hospedagem de correio eletrônico, bancos de dados e outras aplicações envolvendo o compartilhamento e o armazenamento persistente de grandes quantidades de objetos de dados.

O projeto inclui suporte para o armazenamento replicado de objetos de dados mutáveis e imutáveis. O mecanismo para manutenção da consistência entre as réplicas pode ser personalizado de acordo com as necessidades da aplicação, de uma maneira que foi inspirada pelo sistema Bayou (Seção 18.4.2). A privacidade e a integridade são obtidas por meio da criptografia dos dados e pelo uso de um protocolo de acordo bizantino (veja a Seção 15.5) para atualizações nos objetos replicados. Isso é necessário porque a confiabilidade dos nós individuais não pode ser pressuposta.

Foi construído um protótipo do OceanStore, chamado Pond [Rhea *et al.* 2003]. Ele é suficientemente completo para suportar aplicativos e seu desempenho foi avaliado em uma variedade de comparativos para validar o projeto OceanStore e verificar seu desempenho com estratégias mais tradicionais. No restante desta seção, fornecemos um panorama do projeto OceanStore/Pond e resumimos os resultados da avaliação.

O Pond usa o mecanismo de sobreposição de roteamento Tapestry para colocar blocos de dados em nós distribuídos por toda a Internet e para enviar requisições a eles.

Organização do armazenamento • Os objetos de dados OceanStore/Pond são análogos a arquivos, com seus dados armazenados em um conjunto de blocos. Contudo, cada objeto é representado como uma sequência ordenada de versões imutáveis que são (em princípio)

Nome	Significado	Descrição
BGUID	GUID do bloco	Código de resumo seguro de um bloco de dados
VGUID	GUID da versão	BGUID do bloco-raiz de uma versão
AGUID	GUID ativo	Identifica exclusivamente todas as versões de um objeto

Figura 10.14 Tipos de identificador usados no OceanStore.

mantidas para sempre. Toda atualização em um objeto resulta na geração de uma nova versão. As versões compartilham os blocos inalterados, seguindo a técnica da cópia na escrita para criar e atualizar objetos, descrita na Seção 7.4.2. Assim, uma pequena diferença entre as versões exige apenas uma pequena quantidade de armazenamento adicional.

Os objetos são estruturados de uma maneira que lembra o sistema de armazenamento do Unix, com os blocos de dados organizados e acessados por meio de um bloco de metadados chamado de bloco-raiz e por blocos de procedimento indireto adicionais, se necessário (cf. *i-nodes* Unix). Outro nível de procedimento indireto é usado para associar um nome textual persistente, ou outro nome visível externamente (por exemplo, o nome de caminho de um arquivo), à sequência de versões de um objeto de dados. A Figura 10.13 ilustra essa organização. Os GUIDs são associados ao objeto (um AGUID), ao bloco-raiz de cada versão do objeto (um VGUID), a blocos de procedimento indireto e a blocos de dados (BGUIDs). Várias réplicas de cada bloco são armazenadas nos pares de nós, selecionados de acordo com os critérios de localidade e disponibilidade de armazenamento, e seus GUIDs são publicados (usando a primitiva *publish()* da Figura 10.5) em cada um dos nós que contêm uma réplica para que o Tapestry possa ser usado pelos clientes para acessar os blocos.

São usados três tipos de GUIDs, conforme resumido na Figura 10.14. Os dois primeiros são GUIDs do tipo normalmente atribuído a objetos armazenados no Tapestry – eles são calculados a partir do conteúdo do bloco relevante, usando uma função de resumo segura para que eles possam ser usados posteriormente para autenticar e verificar a integridade do conteúdo. Os blocos a que eles fazem referência são necessariamente imutáveis, pois qualquer alteração no conteúdo de um bloco invalidaria o uso do GUID como prova de autenticação.

O terceiro tipo de identificador usado é o AGUID. Ele se refere (indiretamente) ao fluxo inteiro de versões de um objeto, permitindo que os clientes acessem a versão corrente do objeto ou qualquer versão anterior. Como os objetos armazenados são mutáveis, os GUIDs usados para identificá-los não podem ser derivados de seus conteúdos, pois isso tornaria os GUIDs mantidos em índices, etc., obsoletos quando um objeto mudasse.

Em vez disso, quando um novo objeto de armazenamento é criado, um AGUID permanente é gerado pela aplicação de uma função de resumo segura em um nome específico da aplicação (por exemplo, o nome de arquivo) fornecido pelo cliente que está criando o objeto e de uma chave pública que representa o proprietário do objeto (veja a Seção 11.2.5). Em um aplicativo de sistema de armazenamento, um AGUID seria mantido nos diretórios para cada nome de arquivo.

A associação entre um AGUID e a sequência de versões do objeto que ele identifica é gravada em um certificado assinado, que é armazenado e replicado por um esquema de replicação de cópia primária (também chamada de replicação passiva, veja a Seção 18.3.1). O certificado inclui o VGUID da versão corrente, e o bloco-raiz de cada versão contém o VGUID da versão anterior, de modo que existe um encadeamento de referê-

cias permitindo que os clientes que contenham um certificado percorram o encadeamento de versões inteiro (Figura 10.13). Um certificado assinado é necessário para garantir que a associação seja autêntica e que foi feita por um principal (agente) autorizado. Espera-se que os clientes verifiquem isso. Quando é criada uma nova versão de um objeto, é gerado um novo certificado contendo o VGUID da nova versão, junto a um carimbo de tempo e um número de sequência de versão.

O modelo de confiança dos sistemas *peer-to-peer* exige que a construção de cada novo certificado seja consentida (conforme descrito a seguir) entre um pequeno conjunto de nós chamados de *anel interno*. Quando um novo objeto é armazenado no OceanStore, é selecionado um conjunto de nós para atuar como anel interno desse objeto. Eles usam a primitiva *publish()* do Tapestry para tornar o AGUID do objeto conhecido para o Tapestry. Os clientes podem, então, usar o Tapestry para direcionar requisições do certificado do objeto para um dos nós do anel interno.

O novo certificado substitui a cópia primária antiga mantida em cada nó do anel interno e é disseminado para um número maior de cópias secundárias. Fica a cargo dos clientes determinar com que frequência eles verificam a existência de uma nova versão (por exemplo, a maioria das instalações de NFS opera com uma janela de consistência de 30 segundos entre cliente e servidor; veja a Seção 12.3).

Como sempre acontece nos sistemas *peer-to-peer*, a confiança não pode ser depositada em um nó individual. A atualização de cópias primárias exige acordo de consenso entre os nós do anel interno. Eles usam uma versão de uma máquina de estados, baseada no algoritmo de acordo bizantino, descrito por Castro e Liskov [2000], para atualizar o objeto e assinar o certificado. O uso de um protocolo de acordo bizantino garante que o certificado seja mantido corretamente, mesmo que alguns membros do anel interno falhem ou se comportem de maneira mal-intencionada. Como os custos computacionais e de comunicação do acordo bizantino sobem com o quadrado do número de nós envolvidos, o número de nós no anel interno é mantido pequeno, e o certificado resultante é replicado usando-se o esquema de cópia primária.

A realização de uma atualização também envolve a verificação dos direitos de acesso e a serialização de todas as outras escritas pendentes. Uma vez concluído o processo de atualização da cópia primária, os resultados são disseminados nas réplicas secundárias armazenadas nos nós de fora do anel interno, usando uma árvore de roteamento *multicast* gerenciada pelo Tapestry.

Devido à sua natureza de somente leitura, os blocos de dados são replicados por um mecanismo diferente, de armazenamento mais eficiente. Ele é baseado na divisão de cada bloco em m fragmentos de igual tamanho, os quais são codificados usando *códigos de obliteração* [Weatherspoon e Kubiatowicz 2002] em n fragmentos, onde $n > m$. A principal propriedade do código de obliteração é que é possível reconstruir um bloco a partir de quaisquer m fragmentos. Em um sistema que usa código de obliteração, todos os objetos de dados permanecem disponíveis com a perda de até $n-m$ nós. Na implementação Pond, $m = 16$ e $n = 32$, de modo que, dobrando o custo de armazenamento, o sistema pode tolerar a falha de até 16 nós, sem perda de dados. Os fragmentos são armazenados na rede usando o Tapestry para colocá-los e recuperá-los.

Esse alto nível de tolerância a falhas e disponibilidade de dados é obtido a certo custo na reconstrução de blocos a partir dos fragmentos do código de obliteração. Para minimizar o impacto disso, os blocos inteiros também são armazenados na rede usando o Tapestry. Como podem ser reconstruídos a partir de seus fragmentos, esses blocos são tratados como uma cache – eles não são tolerantes a falhas e podem ser eliminados quando for exigido espaço de armazenamento.

Fase	Rede local		Rede remota		
	NFS Linux	Pond	NFS Linux	Pond	Operações predominantes
1	0,0	1,9	0,9	2,8	Leitura e escrita
2	0,3	11,0	9,4	16,8	Leitura e escrita
3	1,1	1,8	8,3	1,8	Leitura
4	0,5	1,5	6,9	1,5	Leitura
5	2,6	21,0	21,5	32,0	Leitura e escrita
Total	4,5	37,2	47,0	54,9	

Os valores mostram tempos em segundos para executar as diferentes fases do benchmark do AFS. Ele tem cinco fases: (1) cria subdiretórios recursivamente; (2) copia uma árvore de diretórios; (3) examina o status de todos os arquivos da árvore sem examinar seus dados; (4) examina cada byte de dados em todos os arquivos; e (5) compila e vincula os arquivos.

Figura 10.15 Avaliação de desempenho do protótipo Pond simulando o NFS.

Desempenho • O Pond foi desenvolvido como protótipo para provar a viabilidade de um serviço de arquivo *peer-to-peer* com escalabilidade, em vez de uma implementação de produção. Ele é implementado em Java e inclui quase tudo do projeto descrito anteriormente. Ele foi avaliado em diversos comparativos (*benchmarks*) destinados a isso e em uma simulação simples de cliente e servidor NFS em termos de objetos OceanStore. Os desenvolvedores testaram a simulação de NFS usando o *benchmark* do Andrew [Howard *et al.* 1988], que simula uma carga de trabalho de desenvolvimento de *software*. A tabela da Figura 10.15 mostra os resultados deste último. Eles foram obtidos usando-se um PC Pentium III de 1 GHz executando Linux. Os testes de rede local foram realizados usando uma Ethernet Gigabit e os resultados de rede de longa distância foram obtidos usando-se dois conjuntos de nós ligados pela Internet.

As conclusões tiradas pelos autores foram que o desempenho do OceanStore/Pond ao operar sobre uma rede de longa distância (isto é, a Internet) ultrapassa em muito o do NFS para leitura, e está dentro de um fator de três em relação ao NFS para atualização de arquivos e diretórios; os resultados em rede local são substancialmente piores. De modo geral, os resultados sugeriram que um serviço de arquivo *peer-to-peer* com escala de Internet, baseado no projeto OceanStore, seria uma solução eficiente para a distribuição de arquivos que não mudam com muita rapidez (como as cópias de páginas Web armazenadas em cache). Seu potencial para ser empregado como uma alternativa para o NFS é questionável, mesmo para a rede de longa distância, e é claramente não competitivo para uso puramente local.

Esses resultados foram obtidos com blocos de dados armazenados sem fragmentação e replicação baseada em código de obliteração. O uso de chaves públicas contribui substancialmente para o custo computacional da operação do Pond. Os valores mostrados são para chaves de 512 bits, cuja segurança é boa, mas menos que ótima. Os resultados para chaves de 1024 bits foram consideravelmente piores, principalmente para os testes comparativos que envolviam atualizações de arquivo. Também foram realizadas medidas de avaliação específicas, como, por exemplo, o impacto do processo de acordo bizantino sobre a latência das atualizações. Eles estão no intervalo de 100 ms a 10 segundos. Um teste de desempenho de saída de atualização atingiu um máximo de 100 atualizações/segundo.

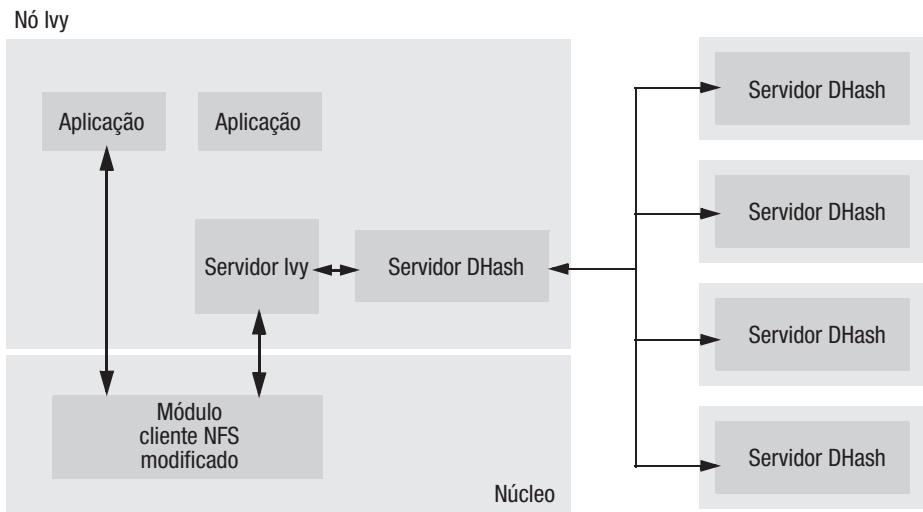


Figura 10.16 Arquitetura do sistema Ivy.

10.6.3 Sistema de arquivos Ivy

Assim como o OceanStore, o Ivy [Muthitacharoen *et al.* 2002] é um sistema de arquivos de leitura/escrita que suporta vários clientes implementados sobre uma camada de sobreposição de roteamento e um sistema de armazenamento de dados distribuído endereçado por funções de resumo. Ao contrário do OceanStore, o sistema de arquivos Ivy simula um servidor Sun NFS. O Ivy armazena o estado dos arquivos como *logs* das requisições de atualização de arquivos emitidas por clientes Ivy e reconstrói os arquivos percorrendo os *logs*, quando não é capaz de atender a uma requisição de acesso a partir de sua cache local. Os registros do *log* são mantidos no serviço de armazenamento distribuído endereçado por função de resumo DHash [Dabek *et al.* 2001]. (Os *logs* foram usados pela primeira vez para gravar atualizações de arquivo no sistema operacional distribuído Sprite [Rosenblum e Oosterhout 1992], conforme brevemente descrito na Seção 12.5, mas eles foram usados simplesmente para otimizar o desempenho de atualização do sistema de arquivos.)

O projeto do Ivy resolve vários problemas não solucionados anteriormente, que surgem da necessidade de hospedar arquivos em máquinas parcialmente confiáveis, ou não confiáveis, incluindo:

- A manutenção de metadados de arquivo consistentes (cf. o conteúdo de *i-node* nos sistemas de arquivos Unix/NFS) com atualizações de arquivo potencialmente concorrentes em diferentes nós. Não são usadas travas (*locks*), pois a falha de nós, ou de conectividade da rede, poderia causar um bloqueio indefinido.
- Confiança parcial entre os participantes e a vulnerabilidade a ataques das máquinas dos participantes. A recuperação das falhas de integridade causadas por tais ataques é baseada na noção de modos de visualização do sistema de arquivos. Um modo de visualização é uma representação do estado, construído a partir de *logs*, das atualizações feitas por um conjunto de participantes. Participantes podem ser removidos e um modo de visualização pode ser recalculado sem suas atualizações. Assim, um sistema de arquivos compartilhado é visto como o resultado da integração de todas as atualizações realizadas por um conjunto de participantes (selecionados dinamicamente).

- Operação continuada durante particionamentos da rede, os quais podem resultar em atualizações conflitantes nos arquivos compartilhados. As atualizações conflitantes são resolvidas usando-se métodos relacionados àqueles utilizados no sistema de arquivos Coda (Seção 18.4.3).

O Ivy implementa em cada nó cliente uma API baseada no protocolo NFS (semelhante ao conjunto de operações listadas na Seção 12.3, Figura 12.9). Os nós clientes incluem um processo servidor Ivy que usa DHash para armazenar e acessar registros de *log* em nós de uma rede local ou remota, com base em chaves (GUIDs), calculadas como a função de resumo do conteúdo do registro (veja a Figura 10.16). O DHash implementa uma interface de programação como a que aparece na Figura 10.4 e replica todas as entradas em vários nós para ter boa capacidade de recuperação e disponibilidade. Os autores do Ivy mencionam que, em princípio, o DHash poderia ser substituído por outro sistema de armazenamento distribuído endereçado com código de resumo, como o Pastry, o Tapestry ou o CAN.

Um sistema de armazenamento de arquivos Ivy consiste em um conjunto de *logs* de atualização, um por participante. Cada participante do Ivy anexa apenas em seu próprio *log*, mas pode ler todos os *logs* que compreendem o sistema de arquivos. As atualizações são armazenadas em *logs*, por participante separados, de modo que podem ser desfeitas no caso de brechas de segurança ou falhas de consistência.

Um *log* do Ivy é uma lista encadeada reversa, ordenada no tempo, de entradas de *log*. Cada entrada é um registro com um carimbo tempo de uma requisição de cliente para alterar o conteúdo, ou metadados, de um arquivo ou diretório. O DHash usa a função de resumo SHA-1 de 160 bits de um registro como chave para colocar e recuperar o registro. Cada participante também mantém um bloco DHash mutável (chamado de cabeçalho de *log*), que aponta para o registro de *log* mais recente do participante. Os blocos mutáveis recebem um par de chaves públicas de criptografia de seus proprietários. O conteúdo do bloco é assinado com a chave privada e, portanto, pode ser autenticado com a chave pública correspondente. O Ivy usa vetores de versão (isto é, carimbo de tempo vetoriais; veja a Seção 14.4) para impor uma ordem total nas entradas de *log*, ao ler vários *logs*.

O DHash armazena os registros de *log* empregando como chave o valor resultante de aplicação de uma função de resumo SHA-1 em seu conteúdo. Os registros de *log* são encadeados ordenados pelo carimbo de tempo, usando a chave DHash como vínculo. O cabeçalho de *log* contém a chave da entrada do *log* mais recente. Para armazenar e recuperar cabeçalhos de *log*, um par de chaves públicas é calculado pelo proprietário do *log*. O valor da chave pública é usado como sua chave DHash, e a chave privada é usada pelo proprietário para assinar o cabeçalho de *log*. Qualquer participante que tenha a chave pública pode recuperar o cabeçalho de *log* e usá-lo para acessar todos os registros do *log*.

Supondo, por enquanto, um sistema de arquivos composto de um único *log*, o método de execução canônico de uma requisição para ler uma sequência de bytes de um arquivo exige uma varredura sequencial do *log* para localizar os registros que contêm atualizações da parte relevante do arquivo. Os *logs* têm comprimento ilimitado, mas a varredura termina quando é encontrado o primeiro registro (ou registros) que cobre a sequência de bytes solicitada.

O algoritmo canônico para acessar um sistema de arquivos multusuário, com múltiplos *logs*, envolve a comparação de carimbos de tempo vetoriais nos registros de *log* e determinação da ordem das atualizações (pois um relógio global não pode ser pressuposto).

O tempo necessário para executar esse processo para uma operação simples, como a requisição de uma operação *read*, é potencialmente muito longo. Ele é reduzido para uma duração mais tolerável, e previsível, pelo uso de uma combinação de caches locais e instantâneos. Instantâneos (*snapshots*) são representações do sistema de arquivos cal-

culadas e armazenadas de forma local em cada participante, como um subproduto de seu uso dos *logs*. Eles constituem uma representação temporária do sistema de arquivos, no sentido de que podem ser invalidados se um participante for ejetado do sistema.

A consistência de atualização é *fechar-para-abrir* (*close-to-open*), isto é, as atualizações realizadas em um arquivo por uma aplicação não são visíveis para outros processos até que o arquivo seja fechado. O uso de um modelo de consistência *fechar-para-abrir* permite que as operações *write* em um arquivo sejam salvas no nó cliente até que o arquivo seja fechado, então o conjunto inteiro de operações *write* é gravado como um único registro de *log*, e um novo registro de cabeçalho de *log* é gerado e gravado (uma extensão do protocolo NFS permite que a ocorrência de uma operação *close* na aplicação seja notificada para o servidor Ivy).

Como existe um servidor Ivy separado em cada nó, e cada um armazena suas atualizações de forma autônoma em um *log* separado, sem coordenação com os outros servidores, a serialização das atualizações deve ser feita no momento em que os *logs* são lidos para construir o conteúdo dos arquivos. Os vetores de versão gravados nos registros de *log* podem ser usados para ordenar a maioria das atualizações, mas atualizações conflitantes são possíveis e devem ser resolvidas por métodos automáticos, ou manuais específicos da aplicação, como é feito no Coda (Seção 18.4.3).

A integridade dos dados é obtida por uma combinação dos mecanismos que já mencionamos: os registros de *log* são imutáveis e seus endereços são um resumo seguro de seu conteúdo; os cabeçalhos de *log* são conferidos pela verificação de uma assinatura de chave pública de seu conteúdo. Porém, o modelo de confiança permite a possibilidade de que um participante mal-intencionado possa ter acesso a um sistema de arquivos. Por exemplo, ele poderia excluir arquivos dos quais obteve direitos de acesso indevidamente. Quando isso é detectado, o participante mal-intencionado é ejetado do modo de visualização; seu *log* não é mais usado para calcular o conteúdo do sistema de arquivos, e os arquivos que já foram excluídos são novamente visíveis no novo modo de visualização.

Os autores do Ivy usaram um *benchmark* modificado do Andrew [Howard *et al.* 1988] para comparar o desempenho do Ivy com um servidor NFS padrão em ambientes de rede local e remota. Eles consideram (*i*) o Ivy usando servidores DHash locais comparados com um único servidor NFS local e (*ii*) o Ivy usando servidores DHash localizados em vários *sites* remotos da Internet comparados com um único servidor NFS remoto. Eles também consideraram as características de desempenho como uma função dos números de participantes em um modo de visualização, o número de participantes escrevendo concorrentemente e o número de servidores DHash usados para armazenar os *logs*.

Eles descobriram que os tempos de execução do Ivy apresentava um fator duas vezes o tempo do NFS, para a maioria dos testes, e não passavam um fator três vezes em nenhum dos casos restantes. Os tempos de execução para implantação em redes de longa distância ultrapassaram os do caso local por um fator de 10 ou mais, mas taxas semelhantes são obtidas para um servidor NFS remoto. Detalhes completos da avaliação de desempenho podem ser encontrados no artigo sobre o Ivy [Muthitacharoen *et al.* 2002]. Contudo, deve-se notar que o NFS não foi projetado para uso remoto; o Andrew File System e outros sistemas baseados em servidor, desenvolvidos mais recentemente, como o xFS [Anderson *et al.* 1996], apresentam desempenho mais alto em implantação remota e poderiam ter constituído bases melhores para a comparação. A principal contribuição do Ivy está em sua nova estratégia para o gerenciamento da segurança e da integridade em um ambiente de confiança parcial – uma característica inevitável de sistemas distribuídos muito grandes, que abrangem muitas organizações e jurisdições.

10.7 Resumo

As arquiteturas *peer-to-peer* surgiram pela primeira vez para suportar compartilhamento de dados em grande escala, com o uso do Napster e de seus derivados para compartilhamento de música digital. O fato de que grande parte de seu uso conflitava com as leis de direitos autorais não diminui sua importância técnica, embora tivessem inconvenientes que restringiam sua implantação em aplicações nas quais garantias da integridade dos dados e disponibilidade não eram importantes.

A pesquisa subsequente resultou no desenvolvimento de plataformas de *middleware peer-to-peer* que enviam requisição para objetos de dados que estão localizados na Internet. Nas estratégias estruturadas, os objetos são endereçados usando-se GUIDs, os quais são nomes puros não contendo nenhuma informação de endereçamento IP. Os objetos são armazenados em nós de acordo com alguma função de mapeamento específica para cada sistema de *middleware*. O envio é realizado por sobreposição de roteamento no *middleware*, a qual mantém tabelas de roteamento e encaminha requisições ao longo de uma rota determinada pelo cálculo da distância, de acordo com a função de mapeamento escolhida. Nas estratégias não estruturadas, os próprios nós formam uma rede *ad hoc* e, então, propagam as pesquisas de busca pelos vizinhos para encontrar os recursos apropriados. Várias estratégias foram desenvolvidas para melhorar o desempenho dessa função de busca e aumentar a escalabilidade global do sistema.

As plataformas de *middleware* acrescentam garantias de integridade baseadas no uso de uma função de resumo segura para gerar os GUIDs e garantias de disponibilidade baseadas na replicação de objetos em vários nós e nos algoritmos de roteamento tolerantes a falhas.

As plataformas foram implantadas em várias aplicações piloto de larga escala, refinadas e avaliadas. Os resultados recentes da avaliação indicam que a tecnologia está pronta para implantação em aplicações que envolvem grandes números de usuários compartilhando muitos objetos de dados. As vantagens dos sistemas *peer-to-peer* incluem:

- sua capacidade de explorar recursos ociosos (armazenamento, processamento) nos nós;
- sua escalabilidade para suportar grandes números de clientes e nós com excelente harmonização das cargas nos enlaces de rede e nos recursos computacionais dos nós;
- as propriedades de organização automática das plataformas de *middleware*, que resultam em custos de suporte amplamente independentes dos números de clientes e nós implantados.

As deficiências e assuntos de pesquisa atuais incluem:

- seu uso para o armazenamento de dados mutáveis é relativamente dispendioso, comparado a um serviço centralizado confiável;
- a principal promessa de que eles proporcionam anonimato de clientes e de nós ainda não resultou em fortes garantias de anonimato.

Exercícios

- 10.1 Os primeiros aplicativos de compartilhamento de arquivo, como o Napster, eram restritos em sua escalabilidade pela necessidade de manter um índice central de recursos e dos nós que os continham. Quais outras soluções para o problema da indexação você pode identificar?
páginas 428–430, 435, Seção 18.4

- 10.2 O problema de manter índices de recursos disponíveis é dependente do aplicativo. Considere a conveniência de cada uma de suas resposta para o Exercício 10.1 para:
- i) compartilhamento de arquivos de música e mídia;
 - ii) armazenamento a longo prazo de material arquivado, como conteúdo de periódico ou jornal;
 - iii) armazenamento na rede de arquivos de leitura-escrita de propósito geral.
- 10.3 Quais são as principais garantias que os usuários esperam que os servidores convencionais (por exemplo, servidores Web ou servidores de arquivos) ofereçam? *Seção 1.5.5*
- 10.4 As garantias oferecidas pelos servidores convencionais podem ser violadas como resultado de:
- i) dano físico no nó;
 - ii) erros ou inconsistências dos administradores de sistema ou seus gerentes;
 - iii) ataques bem-sucedidos contra a segurança do *software* de sistema;
 - iv) erros de *hardware* ou *software*.

Cite dois exemplos de possíveis incidentes para cada tipo de violação. Quais deles poderiam ser descritos como uma brecha de confiança ou ação criminal? Elas seriam brechas de confiança se ocorressem em um computador pessoal que estivesse contribuindo com alguns recursos para um serviço *peer-to-peer*? Por que isso é relevante para os sistemas *peer-to-peer*? *Seção 11.1.1*

- 10.5 Normalmente, os sistemas *peer-to-peer* dependem de sistemas de computador *não confiáveis* e *voláteis* para a maioria de seus recursos. A confiança é um fenômeno social com consequências técnicas. A volatilidade (isto é, a disponibilidade imprevisível) também acontece frequentemente devido às ações humanas. Elabore suas respostas para o Exercício 10.4 discutindo as maneiras possíveis pelas quais cada uma delas provavelmente vai diferir de acordo com os seguintes atributos dos computadores usados:
- i) posse;
 - ii) localização geográfica;
 - iii) conectividade da rede;
 - iv) país ou jurisdição.

O que isso sugere a respeito dos planos para a colocação de objetos de dados em um serviço de armazenamento *peer-to-peer*?

- 10.6 Avalie a disponibilidade e a confiabilidade dos computadores pessoais em seu ambiente. Você deve estimar:

Tempo de funcionamento: as horas por dia em que o computador está operando e conectado na Internet.

Consistência do software: o *software* é gerenciado por um técnico competente?

Segurança: o computador está totalmente protegido contra falsificação por parte de seus usuários ou outras pessoas?

Com base em sua avaliação, discuta a viabilidade de executar um serviço de compartilhamento de dados no conjunto de computadores que você avaliou e descreva, em linhas gerais, os problemas que devem ser tratados em um serviço de compartilhamento de dados *peer-to-peer*. *páginas 431–432*

- 10.7 Explique como o uso de funções de resumo seguro de um objeto, para identificar e direcionar mensagens para ele, garante que seja a prova de falsificação. Quais propriedades são exigidas da função de resumo? Como a integridade pode ser mantida, mesmo que uma proporção substancial dos nós pares seja destruída? *páginas 426, 453, Seção 11.4.3*
- 10.8 Frequentemente, argumenta-se que os sistemas *peer-to-peer* podem oferecer anonimato para:
i) clientes acessando recursos e;
ii) os nós que dão acesso aos recursos.
Discuta cada uma dessas proposições. Sugira uma maneira pela qual a resistência a ataques sobre o anonimato poderia ser melhorada. *página 429*
- 10.9 Os algoritmos de roteamento escolhem o próximo salto de acordo com uma estimativa da distância em algum espaço de endereçamento. Tanto o Pastry como o Tapestry usam espaços de endereço lineares e circulares, nos quais uma função baseada na diferença numérica aproximada entre GUIDs determina sua separação. O Kademia usa a operação XOR dos GUIDs. Como isso ajuda na manutenção das tabelas de roteamento? A operação XOR fornece propriedades apropriadas para uma métrica de distância? *páginas 435, [Maymounkov e Mazieres 2002]*
- 10.10 Quando o serviço de uso de cache Web *peer-to-peer* Squirrel foi avaliado pela simulação, 4,11 saltos eram necessários, em média, para direcionar uma requisição de uma entrada de cache ao simular o tráfego de Redmond, enquanto apenas 1,8 foram exigidos para o tráfego de Cambridge. Explique esse fato e mostre que isso sustenta o desempenho teórico reivindicado para o Pastry. *páginas 436, 450*
- 10.11 Nos sistemas *peer-to-peer* não estruturados, aprimoramentos significativos nos resultados de consultas podem ser conseguidos com a adoção de estratégias de busca específicas. Compare as estratégias de pesquisa em anel expandida e *random walk*, destacando quando é provável que cada uma delas seja eficiente. *página 446*

11

Segurança

- 11.1 Introdução
- 11.2 Visão geral das técnicas de segurança
- 11.3 Algoritmos de criptografia
- 11.4 Assinaturas digitais
- 11.5 Criptografia na prática
- 11.6 Estudos de caso: Needham–Schroeder, Kerberos, TLS, 802.11 WiFi
- 11.7 Resumo

Há uma necessidade generalizada de medidas para garantir a privacidade, a integridade e a disponibilidade dos recursos em sistemas distribuídos. Os ataques contra a segurança assumem as formas de intromissão, mascaramento, falsificação e negação de serviço. Os projetistas de sistemas distribuídos seguros devem conviver com interfaces de serviço expostas e redes inseguras em ambientes onde os invasores provavelmente têm conhecimento dos algoritmos usados.

A criptografia fornece a base para a autenticação de mensagens, assim como sua privacidade e integridade; são necessários protocolos de segurança cuidadosamente projetados para explorá-la. A escolha dos algoritmos de criptografia e o gerenciamento de chaves são fundamentais para a eficácia, o desempenho e a utilidade dos mecanismos de segurança. A criptografia de chave pública facilita a distribuição de chaves, mas seu desempenho é inadequado para a cifragem de grandes volumes de dados, situação em que a criptografia de chave secreta é mais conveniente. Protocolos mistos, como o TLS (Transport Layer Security), estabelecem um canal seguro usando criptografia de chave pública e negociam chaves secretas; depois, essas chaves são utilizadas nas trocas de dados subsequentes.

A informação digital pode ser assinada, produzindo certificados digitais. Os certificados permitem que seja estabelecida confiança entre usuários e organizações.

Na conclusão deste capítulo são apresentados, como estudos de caso, as abordagens e mecanismos de segurança empregados em Kerberos, TLS/SSL e 802.11 WiFi.

11.1 Introdução

Na Seção 2.4.3, apresentamos um modelo simples para examinar os requisitos de segurança em sistemas distribuídos. Concluímos que a necessidade de mecanismos de segurança em sistemas distribuídos surge do desejo de compartilhar recursos. (Geralmente, os recursos que não são compartilhados podem ser protegidos por meio de seu isolamento do acesso externo.) Se considerarmos os recursos compartilhados como objetos, então o requisito é proteger, contra todas as formas de ataque concebíveis, todos os processos que encapsulam objetos compartilhados e todos os canais de comunicação usados para interagir com eles. O modelo apresentado na Seção 2.4.3 forneceu um bom ponto de partida para a identificação dos requisitos de segurança. Eles podem ser resumidos como segue:

- Os processos encapsulam recursos (tanto objetos em nível de linguagem de programação como recursos definidos pelo sistema) e permitem que os clientes os acessem por meio de interfaces. Os principais (usuários ou outros processos) são autorizados a trabalhar com os recursos. Os recursos devem ser protegidos contra acesso não autorizado (Figura 2.17).
- Os processos interagem por meio de uma rede que é compartilhada por muitos usuários. Inimigos (invasores) podem acessar a rede. Eles podem copiar, ou tentar ler, qualquer mensagem transmitida pela rede, e podem injetar na rede mensagens arbitrárias endereçadas para qualquer destino, dando a entender que são provenientes de qualquer fonte (Figura 2.18).

A necessidade de proteger a integridade e a privacidade da informação e de outros recursos pertencentes a indivíduos e organizações é generalizada, tanto no mundo físico quanto no digital. Ela surge do desejo de compartilhar recursos. No mundo físico, as organizações adotam *políticas de segurança* que proporcionam o compartilhamento de recursos dentro de limites especificados. Por exemplo, uma empresa pode permitir a entrada em seus prédios apenas de seus funcionários e visitantes autorizados. Uma política de segurança para documentos pode especificar grupos de funcionários que podem acessar classes de documentos, ou pode ser definida para documentos e usuários individuais.

As políticas de segurança são impostas com a ajuda de *mecanismos de segurança*. Por exemplo, o acesso a um prédio pode ser controlado por uma recepcionista que distribui crachás para os visitantes autorizados, reforçado pela presença de um guarda de segurança e por travas eletrônicas nas portas. O acesso a documentos em papel normalmente é controlado pela ocultação e pela distribuição restrita. No mundo eletrônico, a distinção entre políticas e mecanismos de segurança é igualmente importante; sem ela, seria difícil determinar se um sistema em particular é seguro. As políticas de segurança são independentes da tecnologia usada, assim como, por si só, o uso de uma tranca em uma porta não garante a segurança de um prédio, a não ser que exista uma política para sua utilização (por exemplo, dizendo que a porta será trancada quando ninguém estiver guardando a entrada). Analogamente, os mecanismos de segurança que vamos descrever não garantem, em si, a segurança de um sistema. Na Seção 11.1.2 esboçaremos os requisitos de segurança em vários cenários simples de comércio eletrônico, ilustrando a necessidade de políticas nesse contexto.

O assunto deste capítulo é o uso de mecanismos para proteger dados e outros recursos em sistemas distribuídos e, ao mesmo tempo, possibilitar interações entre os computadores envolvidos pelas políticas de segurança. Os mecanismos que vamos descrever são projetados para fazer valer as políticas de segurança contra os ataques mais determinados.

O papel da criptografia • A criptografia digital fornece a base para a maioria dos mecanismos de segurança em sistemas de computação, mas é importante notar que segurança e criptografia são assuntos distintos. Criptografia é a arte de codificar a informação em um formato que apenas os destinatários apropriados possam acessá-la. A criptografia também pode ser empregada para fornecer uma prova da autenticidade da informação, de maneira semelhante ao uso de assinaturas nas transações convencionais.

A criptografia tem uma história longa e fascinante. A necessidade militar de comunicação segura, e a necessidade correspondente de um inimigo de interceptá-la e decodificá-la, levou ao investimento de muito esforço intelectual por parte de alguns dos melhores matemáticos de sua época. Os leitores que estiverem interessados em explorar essa história encontrarão uma leitura cativante nos livros sobre o assunto de David Kahn [Kahn 1967, 1983, 1991] e de Simon Singh [Singh 1999]. Whitfield Diffie, um dos inventores da criptografia de chave pública, escreveu com conhecimento de causa sobre a história e a política recentes da criptografia [Diffie 1988, Diffie e Landau 1998].

Contudo, foi apenas recentemente que a criptografia saiu do esconderijo a que era obrigada a ficar anteriormente pelos poderes políticos e militares que controlavam seu desenvolvimento e seu uso. Atualmente, ela é assunto livre de pesquisas feitas por uma comunidade ampla e ativa, com resultados publicados em muitos livros, periódicos e conferências. A publicação do livro *Applied Cryptography*, de Schneier [Schneier 1996], foi um marco na abertura do conhecimento no setor. Esse foi o primeiro livro a publicar muitos algoritmos importantes com código-fonte – um passo corajoso, pois quando a primeira edição foi lançada, em 1994, o status jurídico de tal publicação não estava claro. O livro de Schneier continua sendo a referência definitiva sobre a maioria dos aspectos da criptografia moderna. Um livro mais recente, do qual Schneier é coautor [Ferguson e Schneier 2003], fornece uma introdução excelente sobre criptografia e sistemas de computação e contém um panorama discursivo de praticamente todos os algoritmos e técnicas importantes em uso atualmente, incluindo vários publicados desde o lançamento do primeiro livro de Schneier. Além disso, Menezes *et al.* [1997] fornecem um bom manual prático, com uma forte base teórica, e a *Network Security Library* [www.secinf.net] é uma excelente fonte *online* de conhecimento prático e experiência.

O livro *Security Engineering*, de Ross Anderson [Anderson 2008], também é importante. Ele está repleto de lições práticas sobre o projeto de sistemas seguros, extraídas de situações reais e falhas na segurança de sistemas.

Essa nova abertura é, em grande parte, o resultado do tremendo crescimento no interesse por aplicações não militares da criptografia e dos requisitos de segurança dos sistemas computacionais distribuídos. Isso resultou na existência, pela primeira vez, de uma comunidade independente de pesquisadores em criptografia fora do domínio militar.

Ironicamente, a abertura da criptografia para acesso e uso público resultou em uma grande melhoria nas técnicas de criptografia, tanto em seu poder de suportar ataques de inimigos quanto na conveniência com que elas podem ser implantadas. A criptografia de chave pública é um dos frutos dessa abertura. Como outro exemplo, notamos que o algoritmo de criptografia DES, que foi adotado e usado pelos órgãos militares e governamentais dos Estados Unidos, era inicialmente um segredo militar. Sua publicação final e os esforços bem-sucedidos de violá-la resultaram no desenvolvimento de algoritmos de criptografia de chave secreta muito mais poderosos.

Outra reviravolta útil foi o desenvolvimento de uma terminologia e de uma estratégia comuns. Um exemplo desta última é a adoção de um conjunto de nomes familiares para os protagonistas (principais) envolvidos nas transações que precisam se tornar seguras. O uso de nomes de pessoas para principais e invasores ajuda a esclarecer e a

Alice	Primeiro participante
Bob	Segundo participante
Carol	Participante em protocolos de três ou quatro partes
Dave	Participante em protocolos de quatro partes
Eve	Bisbilhoteiro
Mallory	Invasor mal-intencionado
Sara	Um servidor

Figura 11.1 Nomes comumente usados para protagonistas nos protocolos de segurança.

reavivar as descrições dos protocolos de segurança e dos ataques em potencial sobre eles, o que é um passo importante na identificação de suas deficiências. Os nomes mostrados na Figura 11.1 são usados extensivamente na literatura sobre segurança, e vamos usá-los livremente aqui. Não conseguimos descobrir suas origens; a ocorrência mais antiga que conhecemos está no artigo original sobre criptografia de chave pública RSA [Rivest *et al.* 1978]. Um divertido comentário sobre seu uso pode ser encontrado em Gordon [1984].

11.1.1 Ameaças e ataques

Algumas ameaças são óbvias – por exemplo, na maioria dos tipos de rede local é fácil construir e executar um programa em um computador conectado à rede para obter cópias das mensagens transmitidas entre outros computadores. Outras ameaças são mais sutis – se os clientes não conseguem autenticar servidores, um programa poderia se passar por um servidor de arquivos autêntico e, com isso, obter cópias de informações confidenciais que os clientes enviam inadvertidamente para ele armazenar.

Além do perigo da perda, ou de danos na informação ou nos recursos, por meio de violações diretas, pedidos indenizatórios podem ser feitos contra o proprietário de um sistema que não esteja totalmente seguro. Para evitar tais reivindicações, o proprietário deve estar em uma posição favorável para invalidá-las, mostrando que seu sistema é seguro contra tais violações ou produzindo um registro de todas as transações durante o período em questão. Um caso comum é o problema do saque fantasma em caixas eletrônicos. A melhor resposta que um banco pode dar a tal reivindicação é fornecer um registro da transação com a assinatura digital do correntista, de uma forma que não possa ser falsificada por outra pessoa.

O principal objetivo da segurança é restringir o acesso às informações e aos recursos apenas para os principais que estejam autorizados a ter acesso. As ameaças contra a segurança caem em três classes amplas:

Vazamento: a aquisição de informações por destinatários não autorizados;

Falsificação: a alteração não autorizada da informação;

Vandalismo: interferência na operação correta de um sistema, sem ganho para o invasor.

Os ataques nos sistemas distribuídos dependem da obtenção do acesso aos canais de comunicação existentes, ou do estabelecimento de novos canais que sejam mascarados como conexões autorizadas. (Usamos o termo *canal* para nos referirmos a qualquer me-

canismo de comunicação entre processos.) Os métodos de ataque podem, ainda, ser classificados de acordo com a maneira como um canal é empregado inadequadamente:

Intromissão: obter cópias de mensagens sem autorização.

Mascaramento: enviar ou receber mensagens usando a identidade de outro principal, sem sua autorização.

Falsificação de mensagem: interceptar mensagens e alterar seu conteúdo, antes de passá-las para o destinatário desejado. O *ataque do homem no meio* (*man-in-the-middle*) é uma forma de falsificação de mensagem na qual o invasor intercepta a primeira mensagem em uma troca de chaves de criptografia para estabelecer um canal seguro. O invasor substitui por chaves comprometidas que permitem a ele decodificar as mensagens subsequentes, antes de recodificá-las com as chaves corretas e passá-las adiante.

Reprodução: armazenar mensagens interceptadas e enviá-las em uma data posterior. Este ataque pode ser eficiente mesmo com mensagens autenticadas e codificadas.

Negação de serviço: saturar um canal, ou outro recurso, com mensagens para impedir (negar) o acesso de outras pessoas.

Teoricamente, esses são os perigos; mas como os ataques são realizados na prática? Os ataques bem-sucedidos dependem da descoberta de brechas na segurança dos sistemas. Infelizmente, elas são muito comuns nos sistemas atuais e não são, necessariamente, pouco conhecidas. Cheswick e Bellovin [1994] identificam 42 deficiências que consideram riscos sérios em sistemas e componentes normalmente usados na Internet. Elas variam desde adivinhação de senha até ataques nos programas que executam o protocolo de sincronização de tempo ou no envio e recepção de correspondência eletrônica. Algumas delas têm levado a ataques bem-sucedidos e bastante divulgados [Stoll 1989, Spafford 1989] e muitas têm sido exploradas para propósitos nocivos ou criminosos.

Quando a Internet e os sistemas conectados a ela foram projetados, a segurança não era uma prioridade. Os projetistas provavelmente não tinham nenhuma ideia da escala com a qual a Internet cresceria, e o projeto básico de sistemas operacionais, como o UNIX, é anterior ao advento das redes de computadores. Conforme veremos, a incorporação de medidas de segurança precisa ser cuidadosamente pensada no estágio de projeto básico, e o material deste capítulo se destina a fornecer a base para isso.

Focalizamos as ameaças aos sistemas distribuídos que surgem da exposição de seus canais de comunicação e suas interfaces. Para muitos sistemas, essas são as únicas ameaças que precisam ser consideradas (além das que surgem de erro humano – os mecanismos de segurança não podem atuar contra uma senha mal escolhida, ou que seja negligentemente revelada). Entretanto, nos sistemas que incluem códigos móveis e nos sistemas cuja segurança é particularmente sensível ao vazamento de informações, existem mais fontes de ameaças.

Ameaças em código móvel • Várias linguagens de programação desenvolvidas recentemente foram projetadas para permitir que programas sejam carregados em um processo a partir de um servidor remoto e depois executados localmente. Nesse caso, as interfaces internas e os objetos dentro de um processo em execução podem ficar expostos a ataques feitos a partir de um código móvel.

Java é a linguagem mais usada que permite esse tipo de facilidade, e os projetistas prestam considerável atenção ao projeto, à construção da linguagem e aos mecanismos

de carga remota, em um esforço para restringir a exposição (o modelo *sandbox* de proteção contra código móvel).

A máquina virtual Java, a JVM (Java Virtual Machine), foi projetada tendo o código móvel em vista. Ela fornece para cada aplicativo seu próprio ambiente de execução. Cada ambiente tem um gerenciador de segurança que determina quais recursos estão disponíveis para esse aplicativo. Por exemplo, o gerenciador de segurança pode impedir que um aplicativo leia e escreva em arquivos, ou pode dar a ele acesso limitado às conexões de rede. Uma vez configurado, o gerenciador de segurança não pode ser substituído. Quando um usuário executa um programa, como um navegador, que carrega código móvel por *download* para ser executado localmente, em seu nome, ele não tem nenhum bom motivo para acreditar que o código vá se comportar de maneira responsável. Na verdade, existe o perigo de fazer *download* e executar um código nocivo que remova arquivos ou acesse informações privadas. Para proteger os usuários contra código não confiável, a maioria dos navegadores especifica que os *applets* não podem acessar arquivos locais, impressoras ou soquetes de rede. Alguns aplicativos baseados em código móvel são capazes de atribuir vários níveis de confiança no código carregado por *download*. Nesse caso, os gerenciadores de segurança são configurados de forma a dar mais direitos de acessos aos recursos locais.

A JVM adota mais duas medidas para proteger o ambiente local:

1. As classes carregadas por *download* são armazenadas separadamente das classes locais, impedindo que elas substituam as classes locais com versões espúrias.
2. É verificada a validade dos *bytecodes*. Um *bytecode* Java válido é composto de instruções da máquina virtual Java de um conjunto específico. As instruções também são verificadas para garantir que não produzirão certos erros quando o programa for executado, como o acesso a endereços de memória inválidos.

A segurança da linguagem Java foi o assunto de muitas pesquisas, no curso das quais se tornou claro que os mecanismos originais adotados não estavam livres de brechas [McGraw e Felden 1999]. As brechas identificadas foram corrigidas e o sistema de proteção Java foi refinado para permitir que código móvel acessasse recursos locais, quando autorizado a fazer isso [[java.sun.com V](http://java.sun.com/V)].

Apesar da inclusão de mecanismos de verificação de tipo e de validação de código, os mecanismos de segurança incorporados nos sistemas de código móvel ainda não possuem eficientemente o mesmo nível de confiança que aqueles usados para proteger canais e interfaces de comunicação. Isso porque a construção de um ambiente para a execução de programas apresenta muitas oportunidades de erros, e é difícil ter certeza de que todos foram evitados. Volpano e Smith [1999] apontaram que uma estratégia alternativa, baseada em provas de que o comportamento do código móvel é sadio, poderia ser uma solução melhor.

Vazamento de informações • Se a transmissão de uma mensagem entre dois processos puder ser observada, uma informação poderá ser colhida a partir de sua simples existência – por exemplo, uma avalanche de mensagens para um operador de bolsa sobre uma ação em particular poderia indicar um alto nível de negociação dessa ação. Existem formas muito mais sutis de vazamento de informações, algumas nocivas e outras provenientes de erro involuntário. O potencial de vazamento surge quando os resultados de uma computação podem ser observados. Nos anos 70, foi feito um trabalho na prevenção desse tipo de ameaça à segurança [Denning e Denning 1977]. A estratégia adotada foi atribuir níveis de segurança às informações e aos canais, e analisar o fluxo de informa-

ções com o objetivo de garantir que informações de mais alto nível não pudessem fluir para canais de mais baixo nível. Um método para o controle seguro de fluxos de informação foi descrito pela primeira vez por Bell e LaPadula [1975]. A extensão dessa estratégia para sistemas distribuídos com desconfiança mútua entre os componentes é o assunto de pesquisa recente [Myers e Liskov 1997].

11.1.2 Tornando transações eletrônicas seguras

Muitos usos da Internet na indústria, no comércio e em várias outras áreas envolvem transações que dependem fundamentalmente da segurança. Por exemplo:

E-mail: embora os sistemas de *e-mail* não tenham incluído originalmente suporte para segurança, existem muitos empregos de *e-mail* no qual o conteúdo das mensagens deve ser mantido em segredo (por exemplo, no envio de um número de cartão de crédito), ou que o conteúdo e o remetente de uma mensagem devem ser autenticados (por exemplo, ao enviar um lance por *e-mail* em um leilão). A segurança com criptografia baseada nas técnicas descritas neste capítulo é, agora, incluída em muitos clientes de correio eletrônico.

Aquisição de bens e serviços: atualmente, tais transações são muito comuns. Os compradores escolhem bens e pagam por eles usando a Web; os bens são entregues por meio de um mecanismo de distribuição apropriado. *Software* e outros produtos digitais (como gravações e vídeos) podem ser distribuídos por meio de *download* pela Internet. Bens materiais, como livros, CDs e quase todos os outros tipos de produtos, também são comercializados por vendedores na Internet; eles são fornecidos por meio de um serviço de distribuição.

Transações bancárias: atualmente, os bancos eletrônicos oferecem aos usuários praticamente todas as facilidades fornecidas pelos bancos convencionais. Eles podem consultar seus saldos e extratos, transferir dinheiro entre contas, estabelecer pagamentos automáticos regulares, etc.

Microtransações: a Internet se presta ao fornecimento de pequenos volumes de informação e outros serviços para muitos clientes. Por exemplo, atualmente, a maioria das páginas da Web não é paga, mas o desenvolvimento da Web como um meio de publicação de alta qualidade depende, com certeza, do grau com que os fornecedores de informação podem receber pagamentos dos consumidores dessa informação. O uso da Internet para voz e videoconferência fornece outro exemplo de serviço que provavelmente será fornecido somente quando for pago pelos usuários finais. O preço de tais serviços pode chegar a apenas uma fração de um centavo, e os custos e taxas relacionados ao pagamento devem ser proporcionalmente baixos. Em geral, esquemas baseados no envolvimento de um banco ou operadora de cartão de crédito para cada transação não podem conseguir isso.

Transações como essas só podem ser realizadas com segurança quando são protegidas por políticas e mecanismos de segurança apropriados. O comprador deve ser protegido contra a exposição de seus códigos de crédito (número de cartão) durante a transmissão e contra um vendedor fraudulento, que receba o pagamento sem nenhuma intenção de fornecer os bens. Os vendedores devem receber o pagamento antes de liberar os bens e, para produtos carregados por *download*, devem garantir que apenas o cliente que pagou obtenha os dados de forma que possam ser utilizados. A proteção exigida deve ser obtida a um custo que seja razoável em comparação com o valor da transação.

Políticas de segurança para vendedores e compradores na Internet levam aos seguintes requisitos para tornar seguras as compras na Web:

1. Autenticação do vendedor para o comprador, para que este possa ter confiança de que está em contato com um servidor operado pelo vendedor com quem pretende negociar.
2. Impedir que o número do cartão de crédito e outros detalhes do pagamento do comprador caiam nas mãos de outra pessoa e garantir que eles sejam transmitidos do comprador para o vendedor sem alteração.
3. Se os bens estiverem em uma forma conveniente para *download*, certificar-se de que seu conteúdo seja enviado para o comprador, sem alteração e sem exposição para terceiros.

A identidade do comprador normalmente não é exigida pelo vendedor (exceto para o propósito de enviar bens, no caso de não poderem ser carregados por *download*). O vendedor desejará verificar se o comprador tem fundos suficientes para pagar pela compra, mas isso normalmente é feito pela exigência do pagamento pelo banco do comprador, antes do envio dos bens.

As necessidades de segurança das transações bancárias usando uma rede aberta são semelhantes às das transações de compra, com o comprador, como correntista, e o banco como vendedor, mas aqui certamente há necessidade de:

4. Autenticar a identidade do correntista para o banco, antes de dar a ele acesso à sua conta.

Note que, nessa situação, é importante para o banco garantir que o correntista não possa negar sua participação em uma transação. *Não repúdio* ou *irretratabilidade* é o nome dado a esse requisito.

Além dos requisitos anteriores, que são impostos pelas políticas de segurança, existem alguns do sistema em si. Eles surgem da própria escala da Internet, que torna impraticável exigir que os compradores estabeleçam relacionamentos especiais com os vendedores (por exemplo, registrando chaves de criptografia para uso posterior, etc.). Deve ser possível para um comprador concluir uma transação segura com um vendedor, mesmo que não tenha havido nenhum contato anterior entre o comprador e o vendedor, e sem o envolvimento de terceiros. Técnicas como o uso de *cookies* – registros de transações anteriores armazenados no computador do usuário – têm deficiências de segurança óbvias; *desktop* e *dispositivos* móveis são frequentemente localizados em ambientes físicos inseguros.

Devido à importância da segurança para o comércio na Internet e ao rápido crescimento desse comércio, optamos por ilustrar o uso de técnicas de segurança com criptografia descrevendo, na Seção 11.6, o protocolo de segurança padrão *de facto* usado na maior parte do comércio eletrônico – o TLS (Transport Layer Security). Uma descrição do Millicent, um protocolo especificamente projetado para microtransações, pode ser encontrada no endereço www.cdk5.net/security (em inglês).

O comércio na Internet é uma aplicação importante das técnicas de segurança, mas certamente não é a única. Ela é necessária onde quer que computadores sejam usados por indivíduos ou organizações para armazenar e comunicar informações importantes. O uso de *e-mail* codificado para comunicação privada entre indivíduos é um caso questionável que tem sido o assunto de considerável discussão política. Vamos nos referir a esse debate na Seção 11.5.2.

11.1.3 Projeto de sistemas seguros

Nos últimos anos, passos enormes foram dados no desenvolvimento de técnicas de criptografia e em sua aplicação, apesar de o projeto de sistemas seguros continuar sendo uma tarefa inherentemente difícil. No centro desse dilema está o fato de que o objetivo do projetista é excluir *todos* os ataques e brechas possíveis. A situação é semelhante à do programador cujo objetivo deve ser excluir todos os erros de seu programa. Em nenhum dos casos há um método concreto para garantir os objetivos durante o projeto. Projetase para os melhores padrões disponíveis e aplica-se análise e verificações informais. Uma vez concluído o projeto, uma opção é a validação formal. O trabalho feito na validação formal dos protocolos de segurança produziu alguns resultados importantes [Lampson *et al.* 1992, Schneider 1996, Abadi e Gordon 1999]. Uma descrição de um dos primeiros passos nessa direção, a lógica de autenticação BAN [Burrows *et al.* 1990] e sua aplicação podem ser encontradas no endereço www.cdk5.net/security.

Segurança significa evitar desastres e minimizar acidentes. Ao se projetar tendo em vista a segurança é necessário supor o pior. O quadro a seguir mostra um conjunto de suposições e diretrizes de projeto bastante úteis. Essas suposições sustentam o pensamento por trás das técnicas que vamos descrever neste capítulo.

Para demonstrar a validade dos mecanismos de segurança empregados em um sistema, os projetistas do sistema devem, primeiro, construir uma lista de ameaças – os métodos pelos quais as políticas de segurança poderiam ser violadas – e mostrar que cada uma delas é evitada pelos mecanismos empregados. Essa demonstração pode assumir a forma de argumento informal; ou melhor, pode assumir a forma de uma prova lógica.

Provavelmente, nenhuma lista de ameaças será exaustiva; portanto, métodos de auditoria também devem ser usados para detectar violações em aplicações sensíveis à segurança. Eles são muito simples de implementar, caso seja sempre gravado um registro (*log*) seguro das ações em um sistema sensível quanto a segurança, com detalhes dos usuários que estão executando as ações e suas permissões.

Um *log* de segurança conterá uma sequência de registros com carimbo de tempo das ações dos usuários. No mínimo, os registros incluirão a identidade de um principal, a operação executada (por exemplo, excluir arquivo, atualizar registro de conta), a identidade do objeto que sofreu a operação e um carimbo de tempo. Quando violações particulares forem suspeitas, os registros podem ser ampliados para incluir utilização de recurso físico (largura de banda de rede, periféricos) ou o processo de registro pode ter como objetivo operações sobre objetos em particular. A análise subsequente pode ser estatística ou baseada em pesquisa. Mesmo quando nenhuma violação é suspeita, com o passar do tempo, as estatísticas podem ser comparadas para ajudar a descobrir tendências ou eventos incomuns.

O projeto de sistemas seguros é um exercício de ponderação dos custos em relação às ameaças. A série de técnicas que podem ser implantadas para proteger processos e tornar a comunicação entre processos segura é forte o suficiente para suportar quase qualquer ataque, mas seu uso acarreta custos e inconveniências:

- há um custo (em esforço computacional e na utilização da rede) por seu uso. Os custos devem ser ponderados em relação às ameaças;
- medidas de segurança especificadas de forma inadequada podem excluir usuários legítimos da execução das ações necessárias.

Tais compromissos são difíceis de identificar sem comprometer a segurança, e pode parecer que estão em conflito com o conselho do primeiro parágrafo desta subseção, mas o poder das técnicas de segurança pode ser quantificado e selecionado com base no custo

estimado de um ataque a elas. Um exemplo são as técnicas de custo relativamente baixo empregadas no protocolo Millicent para pequenas transações comerciais, descrito no endereço www.cdk5.net/security.

Como ilustração de dificuldades e acidentes que podem advir no projeto de sistemas seguros, examinaremos, na Seção 11.6.4, aquelas que surgiram no projeto de segurança incorporado no padrão de interligação em rede IEEE 802.11 WiFi.

11.2 Visão geral das técnicas de segurança

O objetivo desta seção é apresentar ao leitor algumas das técnicas e mecanismos mais importantes para tornar seguros sistemas e aplicativos distribuídos. Aqui, os descreveremos informalmente, reservando as descrições mais rigorosas para as Seções 11.3 e 11.4. Vamos usar os nomes dos principais (protagonistas) apresentados na Figura 11.1 e as notações para itens cifrados e assinados mostradas na Figura 11.2.

Suposições de pior caso e diretrizes de projeto

As interfaces são expostas: os sistemas distribuídos são compostos de processos que oferecem serviços ou compartilham informações. Suas interfaces de comunicação são necessariamente abertas (para permitir que novos clientes as acessem) – um invasor pode enviar uma mensagem para qualquer interface.

As redes são inseguras: por exemplo, as fontes de mensagem podem ser falsificadas – as mensagens podem parecer terem sido enviadas por Alice, quando na verdade foram enviadas por Mallory. Pode haver *spoofing* dos endereços dos *computadores* – Mallory pode se conectar na rede com o mesmo endereço de Alice e receber cópias das mensagens destinadas a ela.

Límite do tempo de vida e escopo de cada segredo: quando uma chave secreta é gerada pela primeira vez, podemos ter certeza de que ela não foi comprometida. Quanto mais a utilizamos e mais ela se torna conhecida, maior é o risco. O uso de segredos, como senhas e chaves secretas compartilhadas, deve ter um período de validade (limite do tempo de vida) e o compartilhamento deve ser restrito.

Algoritmos e código de programa estão disponíveis para os invasores: quanto maior e mais amplamente distribuído é um segredo, maior é o risco de sua exposição. Os algoritmos de criptografia secretos são totalmente inadequados para os ambientes atuais de redes de larga escala. A melhor prática é publicar os algoritmos usados para criptografia e autenticação, e contar com o segredo das chaves de criptografia. Isso ajuda a garantir que os algoritmos sejam poderosos, lançando-os abertamente para o escrutínio de outras pessoas.

Os invasores podem ter acesso a recursos importantes: o custo do poder de computação está diminuindo rapidamente. Devemos supor que os invasores terão acesso aos maiores e mais poderosos computadores que serão projetados durante o ciclo de vida de um sistema e, então, acrescentar algumas ordens de grandeza para admitir evoluções inesperadas.

Minimização da base confiável: as partes de um sistema que são responsáveis pela implementação de sua segurança e de *todos os componentes de hardware e software com os quais elas contam*, precisam ser confiáveis – isso é frequentemente referido como *base de computação confiável*. Qualquer defeito ou erro de programação nessa base confiável pode produzir deficiências de segurança; portanto, devemos ter como objetivo minimizar seu tamanho. Por exemplo, não se deve confiar em programas aplicativos para proteger dados de seus usuários.

K_A	Chave secreta de Alice
K_B	Chave secreta de Bob
K_{AB}	Chave secreta compartilhada entre Alice e Bob
$K_{A\text{priv}}$	Chave privada de Alice (conhecida apenas por Alice)
$K_{A\text{pub}}$	Chave pública de Alice (publicada por Alice para todos lerem)
$\{M\}_K$	Mensagem M cifrada com a chave K
$[M]_K$	Mensagem M assinada com a chave K

Figura 11.2 Notações de criptografia.

11.2.1 Criptografia

Criptografia é o processo de codificar uma mensagem de maneira a ocultar seu conteúdo. A criptografia moderna inclui vários algoritmos de segurança para cifrar e decifrar mensagens baseados no uso de segredos chamados *chaves*. Uma chave de criptografia é um parâmetro usado em um algoritmo de criptografia de tal maneira que a criptografia não possa ser revertida sem o conhecimento da chave.

Existem duas classes principais de algoritmo de criptografia em uso geral. A primeira usa *chaves secretas compartilhadas* – o remetente e o destinatário devem compartilhar o conhecimento da chave e ela não deve ser revelada a mais ninguém. A segunda classe de algoritmos de criptografia usa *pares de chave pública/privada* – o remetente de uma mensagem usa uma *chave pública* – que já foi publicada pelo destinatário – para cifrar a mensagem. O destinatário usa uma *chave privada* correspondente, para decifrar a mensagem. Embora muitos principais possam inspecionar a chave pública, apenas o destinatário pode decifrar a mensagem, pois somente ele possui a chave privada.

As duas classes de algoritmo de criptografia são extremamente úteis e são amplamente usadas na construção de sistemas distribuídos seguros. Normalmente, os algoritmos de criptografia de chave pública exigem de 100 a 1.000 vezes mais poder de processamento do que os algoritmos de chave secreta, mas existem situações em que sua conveniência supera essa desvantagem.

11.2.2 Usos da criptografia

A criptografia desempenha três papéis importantes na implementação de sistemas seguros. Os apresentaremos aqui em esboço, por meio de alguns cenários simples. Em seções posteriores deste capítulo, descreveremos esses e outros protocolos com mais detalhes, tratando de alguns problemas não resolvidos que são apenas destacados aqui.

Em todos os nossos cenários a seguir, podemos supor que Alice, Bob e todos os outros participantes já concordaram com os algoritmos de criptografia que desejam usar e têm implementações deles. Também supomos que as chaves secretas ou privadas que eles detêm podem ser armazenadas com segurança para impedir que invasores as obtenham.

Segredo e integridade • A criptografia é usada para manter o segredo e a integridade da informação, quando ela é exposta a ataques em potencial; por exemplo, durante a transmissão em redes vulneráveis à intromissão e à falsificação da mensagem. Esse uso da criptografia corresponde à sua função tradicional em atividades militares e de inteligência. Ele explora o fato de que uma mensagem cifrada com uma chave de criptografia em particular só pode ser decifrada por um destinatário que conheça a chave correspondente para decifrar. Assim, ele

mantém o segredo da mensagem cifrada, desde que a chave para decifrar não seja *comprometida* (exposta a quem não é participante da comunicação) e que o algoritmo de criptografia seja poderoso o suficiente para anular todas as tentativas possíveis de violá-lo. A criptografia também mantém a integridade da informação cifrada, já que é possível incluir e verificar algum tipo de informação redundante, como uma soma de verificação.

Cenário 1. Comunicação secreta com uma chave secreta compartilhada: Alice deseja enviar secretamente algumas informações para Bob. Alice e Bob compartilham uma chave secreta K_{AB} .

1. Alice usa K_{AB} e uma função de criptografia consensual $C(K_{AB}, M)$ para cifrar e enviar qualquer número de mensagens $\{M_i\}K_{AB}$ para Bob. (Alice pode usar K_{AB} , desde que seja seguro supor que K_{AB} não foi comprometida.)
2. Bob lê as mensagens cifradas e as decifra usando a função correspondente $D(K_{AB}, M)$.

Agora, Bob pode ler a mensagem original M . Se a mensagem fizer sentido quando for decifrada por Bob, ou melhor, se ela incluir algum valor acordado entre Alice e Bob, como uma soma de verificação da mensagem, então Bob saberá que a mensagem é de Alice e que não foi falsificada. Porém, ainda existem alguns problemas:

Problema 1: como Alice pode enviar uma chave compartilhada K_{AB} para Bob com segurança?

Problema 2: como Bob sabe que qualquer $\{M_i\}$ não é uma cópia de uma mensagem cifrada anteriormente por Alice que foi capturada por Mallory e reproduzida posteriormente? Mallory não precisa ter a chave K_{AB} para realizar esse ataque – ele pode simplesmente copiar o padrão de bits que representa a mensagem e enviá-la para Bob posteriormente. Por exemplo, se a mensagem fosse um pedido para pagar alguém em dinheiro, Mallory poderia enganar Bob, fazendo-o pagar duas vezes.

Posteriormente neste capítulo, vamos mostrar como esses problemas podem ser resolvidos.

Autenticação • A criptografia é usada no suporte de mecanismos de autenticação da comunicação entre pares de principais. Um principal que cifra uma mensagem com êxito, usando uma chave em particular, pode supor que a mensagem é autêntica se ela contiver uma soma de verificação correta, ou algum outro valor esperado, se for usado o modo de criptografia com encadeamento de blocos (Seção 11.3). Ele pode concluir que o remetente da mensagem possuía a chave de cifragem correspondente e, daí, deduzir a identidade do remetente, caso a chave seja conhecida apenas pelas duas partes. Assim, se as chaves forem mantidas em segredo, uma ação de decifrar bem-sucedida autenticará a mensagem como sendo proveniente de um remetente em particular.

Cenário 2. Comunicação autenticada com um servidor: Alice deseja acessar arquivos mantidos por Bob em um servidor de arquivos na rede local da organização onde ela trabalha. Sara é um servidor de autenticação gerenciado com segurança. Sara distribui senhas para os usuários e contém as chaves secretas correntes de todos os principais do sistema a que atende (geradas pela aplicação de alguma função de transformação na senha do usuário). Por exemplo, ele conhece a chave K_A de Alice e K_B de Bob. Em nosso cenário, nos referimos a um *tíquete*. Um tíquete é um item cifrado emitido por um servidor de autenticação que contém a identidade do principal para quem ele é emitido e uma chave compartilhada, gerada para a sessão de comunicação corrente.

1. Alice envia uma mensagem (não cifrada) para Sara, informando sua identidade e solicitando um tíquete para acessar Bob.
2. Sara envia uma resposta para Alice, cifrada com K_A , consistindo em um tíquete (a ser enviado para Bob com cada pedido de acesso a arquivo) cifrado com K_B e uma nova chave secreta K_{AB} para uso na comunicação com Bob. Então, a resposta recebida por Alice é como a seguinte: $\{\{Ticket\}K_B, K_{AB}\}K_A$.
3. Alice decifra a resposta usando K_A (que ela gera a partir de sua senha, usando a mesma função de transformação; a senha não é transmitida pela rede e, uma vez que tenha sido usada, ela é excluída do armazenamento local para evitar seu comprometimento). Se Alice tiver a chave K_A , derivada da senha correta, ela obterá um tíquete válido para usar o serviço de Bob e uma nova chave secreta para uso na comunicação com Bob. Alice não pode decifrar nem falsificar o tíquete, pois ele está cifrado por K_B . Se o destinatário não for Alice, então ele não saberá a senha de Alice; portanto, não poderá decifrar a mensagem.
4. Alice envia o tíquete para Bob, junto a sua identidade e um pedido R para acessar um arquivo: $\{Ticket\}K_B, Alice, R$.
5. O tíquete, originalmente criado por Sara, é na verdade $\{\{K_{AB}, Alice\}K_B$. Bob decifra o tíquete usando sua chave K_B . Portanto, Bob obtém a identidade autêntica de Alice (garantido pelo conhecimento da chave de Alice, compartilhada entre Alice e Sara) e uma nova chave secreta compartilhada K_{AB} para uso na interação com Alice. (Elas são chamadas de *chave de sessão*, pois podem ser usadas com segurança por Alice e Bob para uma sequência de interações.)

Esse cenário é uma versão simplificada do protocolo de autenticação originalmente desenvolvido por Roger Needham e Michael Schroeder [1978] e usado subsequentemente no sistema Kerberos, desenvolvido e usado no MIT [Steiner *et al.* 1988], que será descrito na Seção 11.6.2. Em nossa descrição simplificada desse protocolo, não há nenhuma proteção contra a reprodução de mensagens de autenticação antigas. Essa e algumas outras deficiências serão tratadas em nossa descrição do protocolo Needham–Schroeder completo, na Seção 11.6.1.

O protocolo de autenticação que descrevemos depende do conhecimento anterior, por parte do servidor de autenticação Sara, das chaves K_A e K_B de Alice e de Bob. Isso é possível em uma única organização, em que Sara é executado em um computador fisicamente seguro e gerenciado por um principal confiável que gera os valores iniciais das chaves e os transmite para os usuários por meio de um canal seguro separado. No entanto, ele não é apropriado para comércio eletrônico ou outras aplicações remotas, em que o uso de um canal separado é extremamente inconveniente e o requisito de uma terceira pessoa confiável, irreal. A criptografia de chave pública nos salva desse dilema.

A utilidade dos desafios (challenges): um aspecto importante de Needham e Schroeder, 1978, foi a percepção de que a senha de um usuário não precisa ser enviada para um serviço de autenticação (e, assim, exposta na rede) sempre que for usada. Em vez disso, eles apresentaram o conceito de *desafio*. Isso pode ser visto no passo 2 de nosso cenário anterior, em que o servidor Sara emite um tíquete para Alice, *cifrado na chave secreta K_A de Alice*. Isso constitui um desafio, pois Alice não pode usar o tíquete, a não ser que consiga decifrá-lo, e ela só pode fazer isso se puder determinar K_A , que é derivada da senha de Alice. Um impostor dizendo-se ser Alice seria anulado nesse ponto.

Cenário 3. Comunicação autenticada com chaves públicas: supondo que Bob tenha gerado um par de chaves pública/privada, o seguinte diálogo permite que Bob e Alice estabeleçam uma chave secreta compartilhada K_{AB} :

1. Alice acessa um serviço de distribuição de chaves para obter um *certificado de chave pública*, fornecendo a chave pública de Bob. Ele é chamado de certificado porque é assinado por uma autoridade confiável – uma pessoa ou organização amplamente conhecida como sendo confiável. Após verificar a assinatura, ela lê a chave pública K_{Bpub} de Bob no certificado. (Discutiremos a construção e o uso de certificados de chave pública na Seção 11.2.3.)
2. Alice cria uma nova chave compartilhada K_{AB} e a cifra usando K_{Bpub} com um algoritmo de chave pública. Ela envia o resultado para Bob, junto a um nome (*keyname*) que identifica exclusivamente um par de chaves pública/privada (pois Bob pode ter vários deles). Então, Alice envia *keyname*, $\{K_{AB}\}K_{Bpub}$.
3. Bob seleciona a chave privada K_{Bpriv} correspondente em seu conjunto de chaves privadas e a utiliza para decifrar a mensagem e obter K_{AB} . Note que a mensagem de Alice para Bob poderia ter sido corrompida, ou falsificada, quando estava em trânsito. A consequência seria simplesmente que Bob e Alice não compartilhariam a mesma chave K_{AB} . Se isso for um problema, pode ser resolvido pela adição de um valor, ou de um *string*, previamente combinado na mensagem, como os nomes ou os endereços de *e-mail* de Bob e de Alice, os quais Bob pode verificar após decifrá-la.

O cenário anterior ilustra o uso da criptografia de chave pública para distribuir uma chave secreta compartilhada. Essa técnica é conhecida como *protocolo de criptografia misto* e é muito utilizada, pois explora recursos úteis tanto dos algoritmos de criptografia de chave pública como os de chave secreta.

Problema: essa troca de chave é vulnerável a ataques de homem no meio (*man-in-the-middle*). Mallory pode interceptar o pedido inicial de Alice, para o serviço de distribuição de chaves para obter o certificado de chave pública de Bob, e enviar uma resposta contendo sua própria chave pública. Então, ele poderá interceptar todas as mensagens subsequentes. Em nossa descrição anterior, nós nos defendemos desse ataque exigindo que o certificado de Bob fosse assinado por uma autoridade conhecida. Para se proteger desse ataque, Alice deve garantir que o certificado de chave pública de Bob seja assinado com uma chave pública (conforme descrito a seguir) que ela tenha recebido de uma maneira totalmente segura.

Assinaturas digitais • A criptografia é usada para implementar um mecanismo conhecido como *assinatura digital*. Isso simula a função das assinaturas convencionais, verificando com um outro elemento se uma mensagem, ou um documento, é uma cópia inalterada do que foi produzido pelo signatário.

As técnicas de assinatura digital são baseadas em um vínculo irreversível na mensagem ou documento de um segredo conhecido apenas pelo signatário. Isso pode ser obtido cifrando-se a mensagem – ou melhor, uma forma compactada da mensagem chamada *resumo* (*digest*), usando uma chave conhecida apenas pelo signatário. Um resumo é um valor de comprimento fixo, calculado pela aplicação de uma *função de resumo segura* (*secure digest function*). A função de resumo segura é semelhante a uma função de soma de verificação, mas é muito improvável que ela produza um valor de resumo semelhante para duas mensagens diferentes. O resumo resultante, cifrado, atua como uma assinatura que acompanha a mensagem. Geralmente, é usada criptografia de chave pública para

1. <i>Tipo de certificado:</i>	Número de conta
2. <i>Nome:</i>	Alice
3. <i>Conta:</i>	6262626
4. <i>Autoridade certificadora:</i>	Banco Bob
5. <i>Assinatura:</i>	$\{Digest(field\ 2 + field\ 3)\} K_{Bpriv}$

Figura 11.3 Certificado da conta bancária de Alice.

isso: o criador gera uma assinatura com sua chave privada; a assinatura pode ser decifrada por qualquer destinatário usando a chave pública correspondente. Há um requisito adicional: o verificador deve ter certeza de que a chave pública é realmente do principal que diz ser o signatário – isso é tratado com o uso de certificados de chave pública, descritos na Seção 11.2.3.

Cenário 4. Assinaturas digitais com uma função de resumo segura: Alice quer assinar um documento M para que todo destinatário subsequente possa verificar a autoria. Assim, quando Bob acessar posteriormente o documento assinado, após recebê-lo por meio de qualquer rota e de qualquer fonte (por exemplo, ele poderia ser enviado em uma mensagem ou ser recuperado de um banco de dados), poderá verificar se Alice é a autora.

1. Alice calcula um resumo de comprimento fixo do documento $Digest(M)$.
2. Alice cifra o resumo com sua chave privada, anexa-a a M e torna o resultado M , $\{Digest(M)\} K_{Apriv}$ disponível para os usuários pretendidos.
3. Bob obtém o documento assinado, extrai M e calcula $Digest(M)$.
4. Bob decifra $\{Digest(M)\} K_{Apriv}$ usando a chave pública K_{Apub} de Alice e compara o resultado com seu $Digest(M)$ calculado. Se eles corresponderem, a assinatura é válida.

11.2.3 Certificados

Um certificado digital é um documento contendo uma declaração (normalmente curta) assinada por um principal. Ilustraremos o conceito com um cenário.

Cenário 5. O uso de certificados: Bob é um banco. Quando seus clientes estabelecem contato, eles precisam ter certeza de que estão falando com Bob (o banco), mesmo que nunca tenham entrado em contato com ele antes. Bob precisa autenticar seus clientes, antes de dar a eles acesso às suas contas.

Por exemplo, Alice poderia achar útil obter um certificado de seu banco, informando o número de sua conta (Figura 11.3). Alice poderia usar esse certificado para fazer compras como forma de garantir que tem uma conta no Banco Bob. O certificado é assinado na chave privada K_{Bpriv} de Bob. Uma vendedora, Carol, pode aceitar tal certificado para cobrar produtos na conta de Alice, desde que ela possa validar a assinatura no campo 5. Para isso, Carol precisa ter a chave pública de Bob e certificar-se de que ela é autêntica para evitar a possibilidade de que Alice possa assinar um certificado falso, associando seu nome à conta de outra pessoa. Para realizar esse ataque, Alice simplesmente geraria um novo par de chaves K'_{pub} , K'_{priv} e as usaria para gerar um certificado falsificado, dando a entender que seria proveniente do Banco Bob.

1. <i>Tipo de certificado:</i>	Chave pública
2. <i>Nome:</i>	Banco Bob
3. <i>Chave pública:</i>	K_{Bpub}
4. <i>Autoridade certificadora:</i>	Fred – Federação dos bancos
5. <i>Assinatura:</i>	$\{Digest(field\ 2 + field\ 3)\}K_{Fpriv}$

Figura 11.4 Certificado de chave pública do Banco Bob.

O que Carol precisa é de um certificado informando a chave pública de Bob, assinado por uma autoridade conhecida e confiável. Vamos supor que Fred represente a Federação dos Bancos, cuja função, entre outras, é certificar as chaves públicas dos bancos. Então, Fred emitiria um certificado de chave pública para Bob (Figura 11.4).

É claro que esse certificado depende da autenticidade da chave pública K_{Fpub} de Fred; portanto, temos um problema de autenticidade recursivo – Carol só pode contar com esse certificado se puder ter certeza de que conhece a chave pública K_{Fpub} autêntica de Fred. Podemos quebrar essa recursividade garantindo que Carol obtenha K_{Fpub} por algum meio no qual ela possa ter confiança – ela poderia recebê-la por intermédio de um representante de Fred, ou poderia receber uma cópia assinada dele, ou ainda de alguém que conhece e em quem confia, dizendo que a recebeu diretamente de Fred. Nossa exemplo ilustra um encadeamento de certificados – neste caso, com dois vínculos.

Já mencionamos um dos problemas que surgem com os certificados – a dificuldade de escolher uma autoridade confiável a partir da qual um encadeamento de autenticações possa começar. Raramente a confiança é absoluta; portanto, a escolha de uma autoridade deve depender do objetivo do certificado. Outros problemas surgem com o risco das chaves privadas serem comprometidas (descobertas) e com o comprimento permitido para um encadeamento de certificados – quanto maior o encadeamento, maior o risco de um vínculo fraco.

Desde que se tenha o cuidado de tratar dessas questões, os encadeamentos de certificados são uma base importante para o comércio eletrônico e outros tipos de transação real. Eles ajudam a tratar do problema da escala: existem seis bilhões de pessoas no mundo; então, como podemos construir um ambiente eletrônico no qual possamos estabelecer as credenciais de todas elas?

Os certificados podem ser usados para estabelecer a autenticidade de muitos tipos de declaração. Por exemplo, os membros de um grupo ou associação talvez quisessem manter uma lista de *e-mails* aberta apenas para os membros do grupo. Uma boa maneira de fazer isso seria o gerente do quadro de associados (Bob) emitir um certificado de associado ($S, Bob, \{Digest(S)\}K_{Bpriv}$) para cada membro, onde S é uma declaração da forma *Alice é membro da Sociedade dos Amigos* e K_{Bpriv} é a chave privada de Bob. Um membro solicitando sua inscrição na lista de *e-mails* da Sociedade dos Amigos teria que fornecer uma cópia desse certificado para o sistema de gerenciamento da lista, o qual verificaria o certificado antes de efetivar a inscrição.

Para tornar os certificados úteis, duas coisas são necessárias:

- um formato padrão e uma representação para eles, de modo que os emitentes e os usuários do certificado possam ter êxito em construí-los e interpretá-los;
- acordo sobre a maneira pela qual os encadeamentos de certificados são construídos e, em particular, sobre a noção de autoridade confiável.

Vamos voltar a esses requisitos na Seção 11.4.4.

Às vezes, há necessidade de revogar um certificado – por exemplo, Alice poderia deixar de ser membro da Sociedade dos Amigos, mas ela e outros membros provavelmente continuariam a manter cópias armazenadas de seus certificados de associados. Seria dispendioso, ou mesmo impossível, rastrear e excluir todos esses certificados, e não é fácil invalidar um certificado – seria necessário notificar todos os destinatários possíveis sobre um certificado revogado. A solução usual para esse problema é incluir uma data de expiração no certificado. Quem receber um certificado expirado deve rejeitá-lo, e o dono do certificado deve solicitar sua renovação. Se for exigida uma revogação mais rápida, deve-se recorrer a mecanismos mais complicados.

11.2.4 Controle de acesso

Esboçaremos aqui os conceitos nos quais é baseado o controle de acesso aos recursos em sistemas distribuídos e as técnicas pelas quais ele é implementado. A base conceitual para proteção e controle de acesso foi estabelecida muito claramente em um artigo clássico de Lampson [1971], e detalhes de implementações não distribuídas podem ser encontrados em muitos livros sobre sistemas operacionais [Stallings 2008].

Historicamente, a proteção de recursos em sistemas distribuídos é específica ao serviço. Os servidores recebem mensagens de pedido da forma *<op, principal, recurso>*, onde *op* é a operação solicitada, *principal* é uma identidade ou um conjunto de credenciais do principal que está fazendo o pedido e *recurso* identifica o recurso no qual a operação deve ser aplicada. O servidor deve primeiro autenticar a mensagem de requisição e as credenciais do principal e, depois, aplicar o controle de acesso, recusando qualquer pedido para o qual o principal solicitante não tenha os direitos de acesso necessários para efetuar a operação solicitada no recurso especificado.

Nos sistemas distribuídos orientados a objetos pode haver muitos tipos de objeto nos quais o controle de acesso deve ser aplicado; e as decisões frequentemente são específicas do aplicativo. Por exemplo, Alice só pode fazer um saque por dia em sua conta bancária, enquanto Bob pode fazer três. As decisões de controle de acesso normalmente são deixadas para o código em nível de aplicativo, mas é fornecido suporte genérico para grande parte do mecanismo que suporta as decisões. Isso inclui a autenticação de principais, a assinatura e autenticação das requisições e o gerenciamento de credenciais e informações sobre os direitos de acesso.

Domínios de proteção • Um domínio de proteção é um ambiente de execução compartilhado por um conjunto de processos: ele contém um conjunto de pares *<recurso, direitos>*, listando os recursos que podem ser acessados por todos os processos em execução dentro do domínio e especificando as operações permitidas em cada recurso. Normalmente, um domínio de proteção é associado a um principal – quando um usuário se conecta, sua identidade é autenticada e um domínio de proteção é criado para os processos que ele executará. Conceitualmente, o domínio contém todos os direitos de acesso que o principal possui, incluindo todos aqueles que ele adquire por meio da participação como membro de vários grupos. Por exemplo, no UNIX, o domínio de proteção de um processo é determinado pelos identificadores de usuário e grupo anexados ao processo no momento do *login*. Os direitos são especificados em termos das operações permitidas. Por exemplo, um arquivo poderia ser lido e gravado por um processo e somente lido por outro.

O domínio de proteção é apenas uma abstração. Duas implementações alternativas são comumente usadas em sistemas distribuídos. São elas as *listas de capacidades* (*capabilities*) e as *listas de controle de acesso* (*ACL*, *Access Control Lists*).

Listas de capacidades: um conjunto de capacidades é mantido por processo, de acordo com o domínio em que ele estiver localizado. A capacidade é um valor binário que atua como uma chave de acesso, permitindo que seu possuidor execute certas operações em um recurso especificado. Para uso em sistemas distribuídos, nos quais as capacidades devem ser impossíveis de falsificar, elas assumem uma forma como a seguinte:

<i>Identificador de recurso</i>	Um identificador exclusivo para o recurso pretendido
<i>Operações</i>	Uma lista das operações permitidas no recurso
<i>Código de autenticação</i>	Uma assinatura digital tornando a capacidade impossível de ser falsificada

Os serviços só fornecem capacidades para os clientes quando estes tiverem sido autenticados como pertencentes ao domínio de proteção alegado. A lista de operações na capacidade é um subconjunto das operações definidas para o recurso pretendido e, frequentemente, é codificada como um mapa de bits. Diferentes capacidades são usadas para diferentes combinações de direitos de acesso para o mesmo recurso.

Quando as capacidades são usadas, os pedidos dos clientes assumem a forma $<op, userid, capacidade>$. Isto é, eles incluem uma capacidade para o recurso a ser acessado, em vez de um simples identificador, dando ao servidor uma prova imediata de que o cliente está autorizado a acessar o recurso identificado pela capacidade, com as operações especificadas por ela. Uma verificação de controle de acesso em uma requisição que esteja acompanhado por uma capacidade envolve a sua validação e a verificação de que a operação solicitada está no conjunto permitido pela capacidade. Essa característica é a principal vantagem das capacidades – elas constituem uma chave de acesso autossuficiente, exatamente como a chave física de uma fechadura de porta é a chave de acesso para o prédio que a fechadura protege.

As capacidades compartilham dois inconvenientes das chaves de uma fechadura física:

Roubo de chave: qualquer um que possua a chave de um prédio pode usá-la para ganhar acesso, seja proprietário autorizado da chave ou não – a pessoa pode ter roubado a chave ou a obtido de alguma maneira fraudulenta.

O problema da revogação: a autorização para manter uma chave muda com o tempo. Por exemplo, o portador pode deixar de ser funcionário do dono do prédio, mas poderia guardar a chave, ou uma cópia dela, e usá-la de uma maneira não autorizada.

As únicas soluções disponíveis para esses problemas das chaves físicas são (1) prender o portador da chave ilícita – o que nem sempre é possível a tempo de evitar que ele cause danos – ou (2) mudar a fechadura e redistribuir chaves para todos os proprietários – uma operação desleigante e dispendiosa.

Os problemas análogos das capacidades são claros:

- As capacidades podem, por falta de cuidado, ou como resultado de um ataque de intromissão, cair nas mãos de principais que não sejam aqueles para os quais elas foram emitidas. Se isso acontecer, os servidores não terão poderes para impedir que sejam usados ilicitamente.
- É difícil cancelar capacidades. O status do portador pode mudar e seus direitos de acesso devem mudar da mesma maneira, mas ele ainda pode usar a capacidade.

Soluções para esses dois problemas, baseadas na inclusão de informações identificando o portador e em tempos limites, em conjunto com listas de capacidades revogadas, têm sido propostas e desenvolvidas [Gong 1989, Hayton *et al.* 1998]. Embora aumentem a complexidade de um conceito simples, as capacidades continuam sendo uma técnica importante; por exemplo, elas podem ser usadas em conjunto com as listas de controle de acesso para otimizar o controle do acesso repetido a um mesmo recurso, e elas fornecem um mecanismo mais organizado para a implementação de delegação (veja a Seção 11.2.5).

É interessante notar a semelhança entre capacidades e certificados. Considere o certificado de posse de conta bancária de Alice, apresentado na Seção 11.2.3. Ele difere das capacidades descritas aqui somente porque não há uma lista de operações permitidas e porque o emitente é identificado. Os certificados e as capacidades podem ser conceitos indistintos em algumas circunstâncias. O certificado de Alice poderia ser considerado uma chave de acesso à conta bancária de Alice para executar todas as operações permitidas aos correntistas, desde que o solicitante possa comprovar que é mesmo Alice.

Listas de controle de acesso: cada recurso possui uma lista associada com entradas na forma $\langle\text{domínio}, \text{operações}\rangle$, fornecendo as operações permitidas para cada domínio que tenha acesso ao recurso. Um domínio pode ser especificado pelo identificador de um principal, ou ser uma expressão que pode ser usada para determinar a participação de um principal como membro do domínio. Por exemplo, *o proprietário deste arquivo* é uma expressão que pode ser avaliada pela comparação da identidade do principal solicitante com a identidade do proprietário armazenada em um arquivo.

Esse é o esquema adotado na maioria dos sistemas de arquivos, incluindo UNIX e Windows NT, em que um conjunto de bits de permissão de acesso é associado a cada arquivo, e os domínios para os quais as permissões são concedidas são definidos pela referência às informações de membro armazenadas em cada arquivo.

As requisições para servidores assumem a forma $\langle\text{op, principal, recurso}\rangle$. Para cada requisição, o servidor autentica o principal e verifica se a operação solicitada está incluída na entrada associada a este principal na lista de controle de acesso do recurso relevante.

Implementação • As assinaturas digitais, credenciais e certificados de chave pública fornecem a base de criptografia para o controle de acesso seguro. Canais seguros oferecem vantagens de desempenho, permitindo que vários pedidos sejam manipulados sem a necessidade de verificação repetida dos principais e das credenciais [Wobber *et al.* 1994].

Tanto CORBA como Java oferecem APIs de segurança. O suporte para controle de acesso é um de seus principais objetivos. A linguagem Java fornece suporte para objetos distribuídos para gerenciar seu próprio controle de acesso com as classes *Principal*, *Signer* e *ACL*, e métodos padrão para autenticação e suporte para certificados, validação de assinaturas e verificações de controle de acesso. Também são suportadas criptografia de chave secreta e de chave pública. Farley [1998] apresenta uma boa introdução para esses recursos da linguagem Java. A proteção dos programas Java que incluem código móvel é baseada no conceito de domínio de proteção – código local e código carregado por *download* recebem diferentes domínios de proteção para sua execução. Pode haver um domínio de proteção para cada origem de *download*, com direitos de acesso para diferentes conjuntos de recursos locais, dependendo do nível de confiança depositado no código carregado.

O CORBA oferece uma especificação para um serviço de segurança (*Security Service*) [Blakley 1999, OMG 2002b] com um modelo para ORBs, para fornecer comunicação segura, autenticação, controle de acesso com credenciais, ACLs e auditoria; esses itens estão mais bem descritos na Seção 8.3.

11.2.5 Credenciais

Credenciais são um conjunto de evidências fornecidas por um principal ao solicitar acesso a um recurso. No caso mais simples, um certificado de uma autoridade relevante informando a identidade do principal é suficiente, e isso seria usado para verificar as permissões do principal em uma lista de controle de acesso (veja a Seção 11.2.4). Frequentemente, isso é tudo que é exigido, ou fornecido, mas o conceito pode ser generalizado para lidar com requisitos muito mais sutis.

Não é conveniente exigir que os usuários interajam com o sistema e autentiquem a si mesmos sempre que sua autorização for exigida para executar uma operação em um recurso protegido. Em vez disso, é introduzida a noção de que uma credencial *representa* um principal. Assim, o certificado de chave pública de um usuário representa esse usuário – qualquer processo que receba um pedido autenticado com a chave privada do usuário pode supor que o pedido foi feito por esse usuário.

A ideia de *representa* pode ser levada bem mais adiante. Por exemplo, em uma tarefa cooperativa, poderia-se exigir que certas ações sigilosas só fossem executadas com a autorização de dois membros da equipe; nesse caso, o principal que está solicitando a ação envia sua própria credencial de identificação e uma credencial de endosso de outro membro da equipe, junto a uma indicação de que elas devem ser consideradas em conjunto na verificação das credenciais.

Analogamente, para votar em uma eleição, um voto seria acompanhado de um certificado de eleitor, assim como de um certificado de identificação. Um certificado de delegação permite que um principal atue em nome de outro e assim por diante. Em geral, uma verificação de controle de acesso envolve a avaliação de uma fórmula lógica combinando os certificados fornecidos. Lampson *et al.* [1992] desenvolveram uma lógica de autenticação abrangente para uso na avaliação da autoridade de *representa* realizada por um conjunto de credenciais. Wobber *et al.* [1994] descrevem um sistema que suporta exatamente essa estratégia geral. Mais trabalhos sobre formas úteis de credenciais para uso em tarefas cooperativas reais podem ser encontrados em Rowley [1998].

As credenciais baseadas na função (*role-based credentials*) parecem particularmente úteis no projeto de esquemas práticos de controle de acesso [Sandhu *et al.* 1996]. Conjuntos de credenciais baseadas na função são definidos para organizações, ou para tarefas cooperativas, e direitos de acesso em nível de aplicativo são construídos com referência a eles. Então, funções são atribuídas a principais específicos, por meio da geração de um certificado de função associando um principal a uma função nomeada em uma tarefa ou organização específica [Coulouris *et al.* 1998].

Delegação • Uma forma particularmente útil de credencial é aquela que autoriza um principal, ou um processo atuando para um principal, a executar uma ação com a autoridade de outro principal. Pode surgir uma necessidade de delegação em qualquer situação em que um serviço precise acessar um recurso protegido para completar uma ação em nome de seu cliente. Considere o exemplo de um servidor de impressão que aceita requisições para imprimir arquivos. Seria um desperdício de recursos copiar o arquivo; portanto, o nome do arquivo é passado para o servidor de impressão, e o arquivo é acessado pelo servidor em nome do usuário que está fazendo a requisição. Se o arquivo for protegido contra leitura, isso não funcionará, a não ser que o servidor de impressão possa adquirir direitos temporários para ler o arquivo. A delegação é um mecanismo projetado para resolver problemas como esse.

A delegação pode ser obtida usando-se um certificado de delegação ou uma capacidade. O certificado é assinado pelo principal solicitante e ele autoriza outro principal

(o servidor de impressão, em nosso exemplo) a acessar um recurso nomeado (o arquivo a ser impresso). Nos sistemas que as suportam, as capacidades podem obter o mesmo resultado sem a necessidade de identificar os principais – a capacidade de acessar um recurso pode ser passada em uma requisição para um servidor. A capacidade é um conjunto de direitos codificados, impossível de falsificar, para acessar o recurso.

Quando direitos são delegados, é comum restringi-los a um subconjunto dos direitos mantidos pelo principal emitente para que o principal delegado não possa fazer mal uso deles. Em nosso exemplo, o certificado poderia ter um tempo limitado para reduzir o risco de que o código do servidor de impressão fosse subsequentemente comprometido e o arquivo, exposto a terceiros. O serviço de segurança (*Security Service*) CORBA inclui um mecanismo para a delegação de direitos baseado em certificados, com suporte para a restrição dos direitos transmitidos.

11.2.6 Firewalls

Os *firewalls* foram apresentados e descritos na Seção 3.4.8. Eles protegem intranets, realizando ações de filtragem em comunicações recebidas e enviadas. Aqui, discutiremos suas vantagens e inconvenientes como mecanismos de segurança.

Em um mundo ideal, a comunicação sempre se daria entre processos mutuamente confiáveis e seriam sempre usados canais seguros. Existem muitos motivos pelos quais esse ideal não é atingido, alguns corrigíveis, mas outros inerentes à natureza aberta dos sistemas distribuídos, ou resultantes de erros que estão presentes na maior parte dos *softwares*. A facilidade com que as mensagens de requisição podem ser enviadas para qualquer servidor, em qualquer parte, e o fato de que muitos servidores não são projetados para suportar ataques maldosos de *hackers* ou erros acidentais torna fácil o vazamento de informações destinadas a serem confidenciais. Elementos indesejáveis também podem penetrar na rede de uma organização, permitindo que programas *worm* e vírus entrem em seus computadores. Veja [[web.mit.edu II](#)] para uma crítica mais detalhada dos *firewalls*.

Os *firewalls* produzem um ambiente de comunicação local no qual toda a comunicação externa é interceptada. As mensagens são encaminhadas para o destinatário local pretendido apenas para comunicações explicitamente autorizadas.

O acesso às redes internas pode ser controlado por *firewalls*, mas o acesso a serviços públicos na Internet é irrestrito, pois seu objetivo é oferecer serviços para uma ampla gama de usuários. O uso de *firewalls* não oferece nenhuma proteção contra ataques internos em uma organização e é deficiente em seu controle para acesso externo. Há necessidade de mecanismos de segurança mais refinados, permitindo que usuários individuais compartilhem informações com outros usuários selecionados, sem comprometer a privacidade e a integridade. Abadi *et al.* [1998] descrevem uma estratégia para o uso de acesso a dados privados da Web para usuários externos, baseada em um mecanismo de *túnel* Web que pode ser integrado com um *firewall*. Para usuários confiáveis e autenticados, ela oferece acesso a servidores Web internos por intermédio de um *proxy* seguro baseado no protocolo HTTPS (HTTP sobre TLS).

Os *firewalls* não são particularmente eficazes contra ataques de negação de serviço, como aquele baseado em *spoofing* de IP que descrevemos na Seção 3.4.2. O problema é que a avalanche de mensagens gerada por tais ataques sobrecarrega qualquer ponto de defesa único, como um *firewall*. Qualquer solução para avalanches de mensagens recebidas deve ser aplicada em um nível mais alto. As soluções baseadas em mecanismos de qualidade de serviço para restringir o fluxo de mensagens da rede a um patamar que o destino possa tratá-las parecem os mais promissoras.

11.3 Algoritmos de criptografia

Uma mensagem é cifrada pelo remetente aplicando alguma regra para transformar a mensagem de *texto puro* (qualquer sequência de bits) em um *texto cifrado* (uma sequência de bits diferente). O destinatário deve conhecer a regra inversa para transformar o texto cifrado no texto puro original. Outros principais não podem decifrar a mensagem, a menos que conheçam a regra inversa. A transformação da criptografia é definida com duas partes, uma *função C* e uma *chave K*. A mensagem M cifrada resultante é escrita como $\{M\}_K$.

$$C(K, M) = \{M\}_K$$

A função de criptografia C define um algoritmo que transforma itens de dados de texto puro em dados cifrados, combinando-os com a chave e transpondo-os de uma maneira fortemente dependente do valor da chave. Podemos considerar um algoritmo de criptografia como a especificação de uma grande família de funções, das quais um membro em particular é selecionado por determinada chave. A decifração é feita usando-se a função inversa D , que também recebe uma chave como parâmetro. Para a criptografia de chave secreta, a chave usada para decifrar é a mesma para cifrar:

$$D(K, E(K, M)) = M$$

Devido ao uso simétrico das chaves, a criptografia de chave secreta é frequentemente referida como *criptografia simétrica*, enquanto a criptografia de chave pública é referida como *assimétrica*, pois as chaves usadas para cifrar e decifrar são diferentes, conforme veremos a seguir. Na próxima seção, vamos descrever várias funções de criptografia amplamente usadas dos dois tipos.

Algoritmos simétricos • Se eliminarmos da consideração o parâmetro chave, definindo $F_K([M]) = C(K, M)$, então uma propriedade das funções fortes de criptografia é que $F_K([M])$ é relativamente fácil de calcular, enquanto a função inversa, $F_K^{-1}([M])$, é tão difícil de calcular que não é exequível. Tais funções são conhecidas como funções de mão única. A eficácia de qualquer método para cifrar informações depende do uso de uma função de criptografia F_K que tenha essa propriedade de mão única. É isso que faz a proteção contra ataques projetados para descobrir M , dado $\{M\}_K$.

No caso de algoritmos simétricos bem projetados, como os descritos na próxima seção, seu poder contra tentativas de descobrir K , dado um texto puro M e o texto cifrado correspondente $\{M\}_K$, depende do tamanho de K . Isso porque a forma geral mais eficiente de ataque é a mais grosseira, conhecida como *ataque de força bruta*. A estratégia da força bruta é examinar todos os valores possíveis de K , calculando $C(K, M)$ até que o resultado corresponda ao valor de $\{M\}_K$ que já é conhecido. Se K tiver N bits, então tal ataque exigirá 2^{N-1} iterações, em média, e um máximo de 2^N iterações, para encontrar K . Assim, o tempo para violar K é exponencial ao número de bits em K .

Algoritmos assimétricos • Quando um par de chaves pública/privada é usado, as funções de mão única são exploradas de outra maneira. A exequibilidade de um esquema de chave pública foi proposta pela primeira vez por Diffie e Hellman [1976], como um método de criptografia que elimina a necessidade de confiança entre as partes comunicantes. A base de todos os esquemas de chave pública é a existência de *funções de alçapão (trap-door functions)*. Uma função de alçapão é uma função de mão única com uma saída secreta – ela é fácil de calcular em uma direção, mas impossível de calcular seu inverso, a não ser que um segredo seja conhecido. Diffie e Hellman foram os primeiros a sugerir a

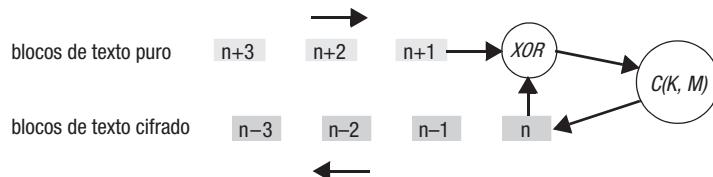


Figura 11.5 Encadeamento de blocos de cifra.

possibilidade de encontrar tais funções e usá-las de forma prática na criptografia. Desde então, vários esquemas de chave pública foram propostos e desenvolvidos. Todos dependem do uso de funções envolvendo grandes números, como funções de alcápão.

O par de chaves necessário para os algoritmos assimétricos é derivado de uma raiz comum. Para o algoritmo RSA, descrito na Seção 11.3.2, a raiz é um par de números primos muito grandes, escolhidos arbitrariamente. A derivação do par de chaves a partir da raiz é uma função de mão única. No caso do algoritmo RSA, os números primos são multiplicados – um cálculo que leva apenas alguns segundos, mesmo para números primos muito grandes. Evidentemente, o produto resultante, N , é muito maior do que os multiplicandos. Esse uso da multiplicação é uma função de mão única, pois é praticamente impossível, em termos computacionais, inferir os multiplicandos originais a partir do produto – isto é, decompor o produto em fatores.

Um dos pares de chaves é usado para criptografia. Para o RSA, a função de criptografia oculta o texto puro, tratando cada bloco de bits como um número binário e elevando-o à potência da chave, módulo N . O número resultante é o bloco de texto cifrado correspondente.

O tamanho de N , e pelo menos um dos pares de chaves, é muito maior do que o tamanho seguro para chaves simétricas para garantir que N não possa ser decomposto em fatores. Por esse motivo, o potencial de ataques de força bruta no RSA é pequeno; sua resistência aos ataques depende da impossibilidade de decompor N em fatores. Vamos discutir os tamanhos seguros para N na Seção 11.3.2.

Cifras de bloco • A maioria dos algoritmos de criptografia opera em blocos de dados de tamanho fixo; 64 bits é um tamanho popular para os blocos. Uma mensagem é subdividida em blocos; se necessário, o último bloco é preenchido para completar o comprimento padrão, e cada bloco é cifrado independentemente. O primeiro bloco estará disponível para transmissão assim que tiver sido cifrado.

Para uma cifra de bloco simples, o valor de cada bloco de texto cifrado não depende dos blocos precedentes. Isso constitui uma deficiência, pois um invasor pode reconhecer padrões repetidos e inferir seu relacionamento com o texto puro. A integridade das mensagens também não é garantida, a não ser que seja usada uma soma de verificação ou um mecanismo de resumo seguro. A maioria dos algoritmos de cifra de bloco emprega encadeamento de blocos de cifra, ou simplesmente CBC (Cipher Block Chaining) para superar essas deficiências.

Encadeamento de blocos de cifra: no modo de encadeamento de blocos de cifra, cada bloco de texto puro é combinado com o bloco de texto cifrado precedente, usando a operação ou-exclusivo (XOR) antes de ser cifrado (Figura 11.5). Na decifração, cada bloco é decifrado e aplicado nele uma função XOR com o bloco cifrado precedente (que deve ter sido armazenado para esse propósito), para obter o novo bloco de texto puro. Isso funciona porque a operação XOR é sua própria inversa – duas aplicações dela produzem o valor original.

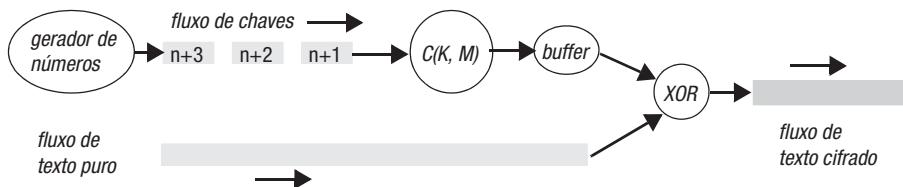


Figura 11.6 Cifra de fluxo.

O CBC se destina a impedir que partes idênticas de texto puro sejam cifradas em partes idênticas de texto cifrado. Contudo, existe uma deficiência no início de cada sequência de blocos – se abrirmos conexões cifradas para dois destinos e enviamos a mesma mensagem, as sequências cifradas de blocos serão as mesmas, e um intruso poderá obter algumas informações úteis a partir disso. Para impedir isso, precisamos inserir um trecho de texto puro diferente na frente de cada mensagem. Tal texto é chamado de *vetor de inicialização*. Um valor de hora constitui um bom vetor de inicialização, obrigando cada mensagem a começar com um bloco de texto puro diferente. Isso, combinado com a operação CBC, resultará em diferentes textos de cifra, mesmo para dois textos puros idênticos.

O uso do modo CBC está restrito à criptografia de dados que são transferidos em uma conexão confiável. A decifração falhará se quaisquer blocos de texto cifrado forem perdidos, pois o processo de decifração não conseguirá decifrar mais nenhum bloco. Portanto, ele é inconveniente para uso em aplicações como as descritas no Capítulo 18, nas quais alguns dados perdidos podem ser tolerados. Em tais circunstâncias, deve ser usada uma cifra de fluxo (*stream cipher*).

Cifras de fluxo • Para algumas aplicações, como codificar conversas telefônicas, a criptografia em blocos é inadequada, pois os fluxos de dados são produzidos em tempo real, em pequenos trechos. As amostras de dados podem ser de apenas 8 bits, ou mesmo de um bit, e seria um desperdício preencher cada uma delas com 64 bits antes de cífrá-las e transmiti-las. As cifras de fluxos são algoritmos de criptografia que podem fazer criptografia de forma incremental, convertendo texto puro em texto cifrado, um bit por vez.

Isso parece difícil de conseguir, mas na verdade é muito simples converter um algoritmo de cifra de bloco para usar como uma cifra de fluxo. O truque é construir um *gerador de fluxo de chaves*. Um fluxo de chaves é uma sequência de bits de comprimento arbitrário que pode ser usada para ocultar o conteúdo de um fluxo de dados, aplicando a operação XOR do fluxo de chaves com o fluxo de dados (Figura 11.6). Se o fluxo de chaves for seguro, então o fluxo de dados cifrado resultante também será.

A ideia é semelhante a uma técnica usada pelos serviços de inteligência para frustrar intrusos, em que um *ruído branco* é produzido para ocultar a conversa em uma sala, enquanto sua gravação é feita. Se o som da sala onde há a conversa e o ruído branco forem gravados separadamente, a conversa poderá ser reproduzida sem ruído, subtraindo-se a gravação do ruído branco da gravação da sala onde há a conversa.

Um gerador de fluxo de chaves pode ser construído pela iteração de uma função matemática sobre um intervalo de valores de entrada para produzir um fluxo contínuo de valores de saída. Os valores de saída são, então, concatenados para compor blocos de texto puro, e os blocos são cifrados usando uma chave compartilhada pelo remetente e pelo receptor. O fluxo de chaves pode ser ainda mais dissimulado pela aplicação do CBC. Os blocos cifrados resultantes são usados como fluxo de chaves. Em princípio, qualquer função que resulte em uma variedade de valores não inteiros diferentes serve, mas geralmente

um gerador de números aleatórios é usado com um valor inicial combinado entre o emissor e o receptor. Para manter a qualidade de serviço do fluxo de dados, os blocos de fluxo de chaves devem ser produzidos pouco antes de serem usados, e o processo que os gera não deve exigir um trabalho de processamento tal que o fluxo de dados seja retardado.

Assim, em princípio, fluxos de dados em tempo real podem ser cifrados com a mesma segurança que os dados armazenados em lotes, desde que esteja disponível um poder de processamento suficiente para cifrar o fluxo de chaves em tempo real. É claro que alguns dispositivos que poderiam tirar proveito da criptografia em tempo real, como os telefones móveis, não são equipados com processadores muito poderosos e, nesse caso, talvez seja necessário reduzir a segurança do algoritmo de fluxo de chaves.

Projeto de algoritmos de criptografia • Existem muitos algoritmos de criptografia bem projetados, de modo que $C(K, M) = \{M\}_K$ oculte o valor de M e torne praticamente impossível recuperar K mais rapidamente do que pela força bruta. Todos os algoritmos de criptografia contam com as manipulações de preservação das informações de M , usando princípios baseados na teoria da informação [Shannon 1949]. Schneier [1996] descreve os princípios da *mistura* e da *difusão* de Shannon para ocultar o conteúdo de um bloco de texto cifrado M , combinando-o com uma chave K de tamanho suficiente para submetê-lo à prova contra ataques de força bruta.

Mistura: operações não destrutivas, como XOR e deslocamento circular (*circular shifting*), são usadas para combinar cada bloco de texto puro com a chave, produzindo um novo padrão de bits que oculta o relacionamento entre os blocos em M e $\{M\}_K$. Se os blocos forem maiores do que alguns caracteres, isso anulará a análise baseada no conhecimento de frequências de caractere. (A máquina alemã Enigma, da Segunda Guerra Mundial, usava blocos de uma letra encadeados, e isso podia ser anulado pela análise estatística.)

Difusão: normalmente, existe repetição e redundância no texto puro. A difusão elimina os padrões regulares resultantes da transposição de partes de cada bloco de texto puro. Se for usado o CBC, a redundância também será distribuída por todo um texto maior. As cifras de fluxo não podem usar difusão, pois não existem blocos.

Nas duas próximas seções, descreveremos o projeto de vários algoritmos práticos importantes. Todos foram projetados de acordo com os princípios anteriores, foram sujeitos a análise rigorosa e são considerados seguros contra todos os ataques conhecidos com uma considerável margem de segurança. Com exceção do algoritmo TEA, que será apresentado para propósitos ilustrativos, os algoritmos descritos aqui estão entre os mais utilizados em aplicações em que é exigida uma segurança forte. Em alguns deles, permanecem algumas pequenas deficiências, ou áreas de preocupação; o espaço aqui não nos permite descrever todas essas preocupações e o leitor deve consultar Schneier [1996] para mais informações. Resumiremos e compararemos a segurança e o desempenho dos algoritmos na Seção 11.5.1.

Os leitores que não precisarem entender o funcionamento dos algoritmos de criptografia podem omitir as seções 11.3.1 e 11.3.2.

11.3.1 Algoritmos de chave secreta (simétricos)

Muitos algoritmos de criptografia foram desenvolvidos e publicados nos últimos anos. Schneier [1996] descreve mais de 25 algoritmos simétricos, muitos dos quais ele identifica como seguros contra ataques conhecidos. Aqui, temos espaço para descrever apenas três deles. Escolhemos o primeiro, o TEA, pela simplicidade de seu projeto e implementação, e o utilizamos para fornecer uma ilustração concreta da natureza de tais algoritmos. Vamos discutir os algoritmos DES e IDEA com menos detalhes. O DES foi, por

```

void encrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];
    unsigned long delta = 0x9e3779b9, sum = 0; int n;
    for (n= 0; n < 32; n++) {
        sum += delta;
        y += ((z << 4) + k[0]) ^ (z+sum) ^ ((z >> 5) + k[1]);
        z += ((y << 4) + k[2]) ^ (y+sum) ^ ((y >> 5) + k[3]);
    }
    text[0] = y; text[1] = z;
}

```

Figura 11.7 Função de criptografia TEA.

muitos anos, o padrão nacional americano, mas agora é de interesse principalmente histórico, pois suas chaves de 56 bits são pequenas demais para resistir aos ataques de força bruta feitos com os computadores modernos. O IDEA usa uma chave de 128 bits e é um dos algoritmos de criptografia de bloco simétrico mais eficazes, sendo uma boa escolha comprovada para criptografia de grandes volumes de dados.

Em 1997, o National Institute for Standards and Technology (NIST) dos Estados Unidos divulgou um convite para propostas de um algoritmo para substituir o DES como um novo padrão de criptografia avançada (AES, Advanced Encryption Standard) daquele país. Em outubro de 2000, foi escolhido o vencedor dentre 21 algoritmos enviados por profissionais da área de criptografia de 11 países. O algoritmo vencedor, Rijndael, foi escolhido por sua combinação de poder e eficiência. Mais informações sobre ele serão dadas a seguir.

TEA • Os princípios de projeto dos algoritmos simétricos apresentados anteriormente são bem ilustrados no *Tiny Encryption Algorithm*, desenvolvido na Universidade de Cambridge [Wheeler e Needham 1994]. A função de criptografia, programada em C, aparece em sua totalidade na Figura 11.7.

O algoritmo TEA usa várias passagens de adição de inteiros, combinados com XOR (o operador `^`) e deslocamentos lógicos em nível de bit (`<<` e `>>`) para obter mistura e difusão dos padrões de bit no texto puro. O texto puro é um bloco de 64 bits representado como dois inteiros de 32 bits no vetor `text[]`. A chave tem 128 bits de comprimento, representada como quatro inteiros de 32 bits.

Em cada uma das 32 passagens, as duas metades do texto são repetidamente combinadas com as partes deslocadas da chave e uma com as outras, nas linhas 5 e 6. O uso da operação XOR e das partes deslocadas do texto proporciona a mistura e o deslocamento. A troca das duas partes do texto proporciona a difusão. A constante `delta`, que não se repete, é combinada, em cada ciclo, com cada parte do texto para ocultar a chave, para evitar o caso de ela poder ser revelada por uma seção de texto que não varie. A função de decifração é a inversa da função de criptografia e aparece na Figura 11.8.

Esse programa curto fornece criptografia de chave secreta de forma segura e razoavelmente rápida. Ele é bem mais rápido do que o algoritmo DES, e o caráter conciso do programa se presta à otimização e à implementação em *hardware*. A chave de 128 bits é segura contra ataques de força bruta. Estudos realizados por seus autores e por outros pesquisadores revelaram apenas duas deficiências muito pequenas, as quais eles trataram em uma nota subsequente [Wheeler e Needham 1997].

Para ilustrar seu uso, a Figura 11.9 mostra um programa simples que utiliza TEA para cifrar ou decifrar dois arquivos previamente abertos (usando a biblioteca C `stdio`).

```

void decrypt(unsigned long k[], unsigned long text[]) {
    unsigned long y = text[0], z = text[1];
    unsigned long delta = 0x9e3779b9, sum = delta << 5; int n;

    for (n= 0; n < 32; n++) {
        z -= ((y << 4) + k[2]) ^ (y + sum) ^ ((y >> 5) + k[3]);
        y -= ((z << 4) + k[0]) ^ (z + sum) ^ ((z >> 5) + k[1]);
        sum -= delta;
    }
    text[0] = y; text[1] = z;
}

```

Figura 11.8 Função de decifração TEA.

DES • O DES (Data Encryption Standard) [National Bureau of Standards 1977] foi desenvolvido pela IBM e, subsequentemente, adotado como padrão nacional nos Estados Unidos para aplicações governamentais e comerciais. Nesse padrão, a função de criptografia mapeia uma entrada de texto puro de 64 bits em uma saída cifrada de 64 bits, usando uma chave de 56 bits. O algoritmo tem 16 estágios dependentes de chave, conhecidos como *passagens*, nos quais os dados a serem cifrados são rotacionados por um número de bits determinado pela chave e por três transposições independentes de chave. O algoritmo demorava muito para ser executado por *software* nos computadores dos anos 70 e 80 e, por isso, foi implementado diretamente em *hardware* VLSI. Por ser um chip VLSI, além de realizar rapidamente o cálculo DES, permitiu sua fácil incorporação em interfaces de rede e em outros *hardware* de comunicação.

Em junho de 1997, ele foi violado com êxito em um ataque de força bruta amplamente divulgado. O ataque foi realizado no contexto de uma competição para demonstrar a falta de segurança da criptografia com chaves menores do que 128 bits [www.rsasecurity.com III].

```

void tea(char mode, FILE *infile, FILE *outfile, unsigned long k[]) {
    /* mode é 'c' para cifrar, 'd' para decifrar, k[] é a chave.*/
    char ch, Text[8]; int i;

    while(!feof(infile)) {
        i = fread(Text, 1, 8, infile);           /* lê 8 bytes de infile para Text */
        if (i <= 0) break;
        while (i < 8) { Text[i++] = ' ';}      /* preenche último bloco com espaços */
        switch (mode) {
            case 'c':
                encrypt(k, (unsigned long*) Text); break;
            case 'd':
                decrypt(k, (unsigned long*) Text); break;
        }
        fwrite(Text, 1, 8, outfile);             /* grava 8 bytes de Text em outfile */
    }
}

```

Figura 11.9 TEA em uso.

Para isso, um consórcio de usuários da Internet executou um programa aplicativo cliente em vários computadores (PCs e outras estações de trabalho). Eles iniciaram com 1.000 máquinas clientes e aumentaram seu número até atingir 14.000 máquinas durante um período de 24h [Curtin e Dolske 1998].

O programa cliente tinha como objetivo violar uma chave particular usada em uma amostra conhecida de texto puro e cifrado e depois usá-la para decifrar uma mensagem de desafio secreta. Os clientes interagiam com um único servidor, o qual coordenava o trabalho deles, divulgando a cada cliente os intervalos de valores de chave a verificar e recebendo deles relatórios de progresso. O computador cliente típico executava o programa cliente em *background* e tinha um desempenho aproximadamente igual ao de um processador Pentium de 200 MHz. A chave foi violada em cerca de 12 semanas, após aproximadamente 25% dos 2^{56} ou 6×10^{16} valores possíveis terem sido verificados. Em 1998, foi desenvolvida uma máquina pela Electronic Frontier Foundation [EFF 1998] que conseguiu violar com êxito chaves DES em cerca de três dias.

Embora ainda seja usado em muitos aplicativos comerciais e outros, em sua forma básica, o DES deve ser considerado obsoleto para a proteção de informações, a não ser as de baixo valor. Uma solução frequentemente usada é conhecida como *triple-DES* (ou 3DES) [ANSI 1985, Schneier 1996]. Ela envolve a aplicação do DES três vezes, com duas chaves, K_1 e K_2 :

$$C_{3DES}(K_1, K_2, M) = C_{DES}(K_1, D_{DES}(K_2, C_{DES}(K_1, M)))$$

Isso proporciona uma resistência contra ataques de força bruta equivalente a um comprimento de chave de 112 bits, já prevendo um certo aumento futuro do poder computacional das máquinas. Entretanto, tal algoritmo tem o inconveniente do mal desempenho, resultante da aplicação tripla de um algoritmo que já é lento pelos padrões modernos.

IDEA • O International Data Encryption Algorithm foi desenvolvido, no início dos anos 90 [Lai e Massey 1990, Lai 1992], como um sucessor do DES. Assim como o TEA, ele usa uma chave de 128 bits para cifrar blocos de 64 bits. Seu algoritmo é baseado na álgebra de grupos e tem oito passagens de XOR, adição módulo 2^{16} e multiplicação. Tanto para o DES como para o IDEA, a mesma função é usada para cifrar e decifrar: uma propriedade útil para algoritmos que são implementados em *hardware*.

O poder do IDEA foi amplamente analisado e nenhuma deficiência significativa foi encontrada. Ele cifra e decifra com aproximadamente três vezes a velocidade do DES.

RC4 • O RC4 é uma cifra de fluxo desenvolvida por Ronald Rivest [Rivest 1992b]. As chaves podem ter qualquer comprimento de até 256 bytes. O RC4 é fácil de implementar [Schneier 1996, pp. 397–8] e cifra e decifra cerca de dez vezes mais rápido que o DES. Consequentemente, ele foi amplamente adotado em vários produtos, incluindo as redes IEEE 802.11 WiFi, mas uma deficiência que permitia aos invasores violar algumas chaves foi descoberta por Fluhrer *et al.* [2001] e isso levou a um reprojeto da segurança do padrão 802.11 (veja a Seção 11.6.4 para mais detalhes).

AES • O algoritmo Rijndael, selecionado para se tornar o algoritmo padrão de criptografia avançada pelo NIST dos Estados Unidos, foi desenvolvido por Joan Daemen e Vincent Rijmen [Daemen e Rijmen 2000, 2002]. A cifra tem comprimento de bloco e de chave variável, com especificações para chaves com comprimento de 128, 192 ou 256 bits para cifrar blocos com comprimento de 128, 192 ou 256 bits. Tanto o comprimento do bloco como o comprimento da chave podem ser ampliados por múltiplos de 32 bits. O número de passagens no algoritmo varia de 9 a 13, dependendo dos tamanhos da chave

e do bloco. O Rijndael pode ser implementado eficientemente em uma grande variedade de processadores e em *hardware*.

11.3.2 Algoritmos de chave pública (assimétricos)

Apenas alguns esquemas de chave pública foram desenvolvidos até agora. Eles dependem do uso de funções de alçapão em grandes números para produzir as chaves. As chaves K_c e K_d representam um par de números grandes, e a função de criptografia executa uma operação, como a exponenciação, em M , usando um deles. A decifração é uma função semelhante, usando a outra chave. Se a exponenciação usar aritmética de módulo, pode-se mostrar que o resultado é igual ao valor original de M ; isto é:

$$D(K_d, C(K_c, M)) = M$$

Um principal p que queira participar de uma comunicação segura com outros compõe um par de chaves, K_c e K_d , e mantém em segredo a chave de decifração K_d . A chave de criptografia K_c é tornada de conhecimento público para uso de quem quiser se comunicar com este principal p . A chave de criptografia K_c pode ser vista como uma parte da função de criptografia de mão única C , e a chave de decifração K_d é o conhecimento secreto que permite ao principal p reverter a criptografia. Qualquer portador de K_c (que está amplamente disponível) pode cifrar mensagens $\{M\} K_c$, mas somente o principal p que tem a K_d secreta pode reverter a operação.

O uso de funções de números grandes implica em elevados custos de processamento para o cálculo das funções C e D . Posteriormente, veremos que esse é um problema que faz com que as chaves públicas sejam usadas apenas nos estágios iniciais das sessões de comunicação seguras. Certamente, o algoritmo RSA é o algoritmo de chave pública mais conhecido, e o descreveremos com alguns detalhes aqui. Outra classe de algoritmos é baseada em funções derivadas do comportamento das curvas elípticas em um plano. Esses algoritmos oferecem, com o mesmo nível de segurança, a possibilidade de funções de criptografia e decifração menos dispendiosas, mas sua aplicação prática está menos avançada e vamos tratar deles apenas sucintamente.

RSA • O projeto de cifra de chave pública de Rivest, Shamir e Adelman (RSA) [Rivest *et al.* 1978] é baseado no uso do produto de dois números primos muito grandes (maiores do que 10^{100}), contando com o fato de que a determinação dos fatores primos de tais números grandes é tão difícil, em termos computacionais, que se torna efetivamente impossível de calcular.

Apesar das extensivas investigações, nenhuma falha foi encontrada e, agora, o RSA é amplamente usado. A seguir apresentamos um esboço do método. Para encontrar um par de chaves $\langle c, d \rangle$:

1. Escolha dois números primos grandes, P e Q (cada um maior do que 10^{100}) e forme

$$N = P \times Q$$

$$Z = (P - 1) \times (Q - 1)$$
2. Para d , escolha qualquer número que seja primo relativo de Z (isto é, de modo que d não tenha fatores comuns com Z).

Ilustramos os cálculos envolvidos usando valores inteiros pequenos para P e Q :

$$P = 13, Q = 17 \rightarrow N = 221, Z = 192$$

$$d = 5$$

3. Para encontrar c , resolva a equação:

$$c \times d = 1 \text{ mod } Z$$

Isto é, $c \times d$ é o menor elemento divisível por d na série $Z+1, 2Z+1, 3Z+1, \dots$

$$c \times d = 1 \text{ mod } 192 = 1, 193, 385, \dots$$

385 é divisível por d

$$c = 385/5 = 77$$

Para cifrar texto usando o método RSA, o texto puro é dividido em blocos iguais de k bits de comprimento, onde $2^k < N$ (isto é, de modo que o valor numérico de um bloco é sempre menor do que N ; nas aplicações práticas, k normalmente está no intervalo de 512 a 1024).

$$k = 7, \text{ pois } 2^7 = 128$$

A função para cifrar um único bloco de texto puro M é:

$$C'(c, N, M) = M^c \text{ mod } N$$

para uma mensagem M , o texto cifrado é $M^{77} \text{ mod } 221$

A função para decifrar um bloco de texto cifrado c , para produzir o bloco de texto puro original, é:

$$D'(d, N, c) = c^d \text{ mod } N$$

Rivest, Shamir e Adelman provaram que C' e D' são inversos mútuos (isto é, $C'(D'(x)) = D'(C'(x)) = x$) para todos os valores de P no intervalo $0 \leq P \leq N$.

Os dois parâmetros c, N podem ser considerados como uma chave para a função de criptografia e, analogamente, os parâmetros d, N representam uma chave para a função de decifração. Portanto, podemos escrever $K_c = \langle c, N \rangle$ e $K_d = \langle d, N \rangle$, e obtermos as funções de criptografia $C(K_c, M) = \{M\}_K$ (a notação aqui indicando que a mensagem cifrada só pode ser decifrada pelo portador da chave privada K_d) e $D(K_d, \{M\}_K) = M$.

É interessante notar uma deficiência em potencial de todos os algoritmos de chave pública – como a chave pública está disponível para invasores, eles podem gerar mensagens cifradas facilmente. Assim, eles podem tentar decifrar uma mensagem desconhecida, cifrando exaustivamente sequências de bits arbitrárias, até obterem uma correspondência com a mensagem de destino. Esse ataque, conhecido como *ataque de texto puro escolhido*, é anulado garantindo-se que todas as mensagens sejam maiores do que o comprimento da chave, de modo que essa forma de ataque de força bruta é menos viável do que um ataque direto na chave.

O destinatário da informação secreta deve publicar, ou distribuir de outra forma, o par $\langle c, N \rangle$, enquanto mantém d em segredo. A publicação de $\langle c, N \rangle$ não compromete o segredo de d , pois qualquer tentativa de determinar d exige o conhecimento dos números primos originais P e Q , e eles só podem ser obtidos pela decomposição dos fatores de N . A decomposição dos fatores de números grandes (lembremos que P e Q foram escolhidos para serem $> 10^{100}$; portanto, $N > 10^{200}$) é extremamente demorada, mesmo em computadores de desempenho muito elevado. Em 1978, Rivest *et al.* concluíram que decompor em fatores um número grande como 10^{200} demoraria mais de quatro bilhões de

anos, com o melhor algoritmo conhecido, em um computador que executasse um milhão de instruções por segundo. Um cálculo semelhante para os computadores atuais reduziria esse tempo para cerca de um milhão de anos.

A RSA Corporation publicou uma série de desafios para fatorar números de mais de 100 dígitos decimais [www.rsasecurity.com III]. Quando este livro estava sendo produzido, números de até 174 dígitos decimais (576 dígitos binários) tinham sido decompostos em fatores com êxito; portanto, claramente o uso do algoritmo RSA com chaves de 512 bits é inaceitavelmente deficiente para muitos propósitos. A RSA Corporation (dona das patentes do algoritmo RSA) recomenda um comprimento de chave de pelo menos 768 bits, ou cerca de 230 dígitos decimais, para segurança a longo prazo (~ 20 anos). Em algumas aplicações, são usadas chaves de até 2.048 bits.

Os cálculos anteriores pressupõem que são usados os melhores algoritmos de fatoração atualmente disponíveis. O RSA e outras formas de criptografia assimétrica que usam multiplicação de números primos como função de mão única serão vulneráveis se, e quando, um algoritmo de decomposição em fatores mais rápido for descoberto.

Algoritmos de curva elíptica • Um método para gerar pares de chaves pública/privada com base nas propriedades das curvas elípticas foi desenvolvido e testado. Os detalhes completos podem ser encontrados no livro de Menezes dedicado ao assunto [Menezes 1993]. As chaves são derivadas de um ramo diferente da matemática e, ao contrário do RSA, sua segurança não depende da dificuldade de decompor números grandes em fatores. As chaves obtidas são mais curtas, seguras, e as demandas de processamento para cifrar e decifrar são menores do que para o RSA. Os algoritmos de criptografia de curva elíptica provavelmente serão adotados mais amplamente no futuro, especialmente em sistemas como os que incorporam dispositivos móveis, os quais têm recursos de processamento limitados. A matemática relevante envolve algumas propriedades bastante complexas das curvas elípticas e está fora dos objetivos deste livro.

11.3.3 Protocolos de criptografia mistos

A criptografia de chave pública é conveniente para o comércio eletrônico porque não precisa de um mecanismo de distribuição de chaves seguro. (Há necessidade de autenticar as chaves públicas, mas isso é muito menos oneroso, exigindo apenas o envio de um certificado de chave pública com a chave.) No entanto, os custos de processamento da criptografia de chave pública são altos demais até para cifrar mensagens de tamanho médio, normalmente encontradas no comércio eletrônico. A solução adotada na maioria dos sistemas distribuídos de larga escala é usar um esquema de criptografia misto, no qual a criptografia de chave pública é usada para autenticar as partes e para cifrar a troca de chaves secretas, as quais são usadas para toda a comunicação subsequente. Vamos descrever a implementação de um protocolo misto no estudo de caso sobre TLS, na Seção 11.6.3.

11.4 Assinaturas digitais

Assinaturas digitais fortes são um requisito essencial para os sistemas seguros. Elas são necessárias para certificar determinadas informações; por exemplo, para fornecer declarações confiáveis, vinculando as identidades dos usuários às suas chaves públicas, ou vinculando alguns direitos de acesso e funções às identidades dos usuários.

A necessidade de assinaturas em muitos tipos de transações comerciais e pessoais é inquestionável. Assinaturas manuscritas têm sido usadas como uma maneira de verificação desde que os documentos existem. Elas são usadas para atender às necessidades dos destinatários do documento, de verificar se ele é:

Autêntico: convence o destinatário de que o signatário deliberadamente assinou o documento e que ele não foi alterado por ninguém.

Impossível de falsificar: fornece uma prova de que o signatário, e ninguém mais, deliberadamente assinou o documento. A assinatura não pode ser copiada, nem colocada em outro documento.

Impossível de repudiar: o signatário não pode negar que o documento foi assinado por ele.

Na realidade, nenhuma dessas propriedades desejáveis da assinatura é inteiramente obtida com as assinaturas convencionais – pois falsificações e cópias são difíceis de detectar, os documentos podem ser alterados após sua assinatura e os signatários, às vezes, são enganados, assinando um documento involuntária ou inconscientemente. Entretanto, desejamos conviver com sua imperfeição, devido à dificuldade de fraudar e ao risco de detecção. Assim como as assinaturas manuscritas, as assinaturas digitais dependem do vínculo de um atributo único e secreto do signatário com um documento. No caso das assinaturas manuscritas, o segredo é a caligrafia do signatário.

As propriedades dos documentos digitais, mantidas em arquivos armazenados ou em mensagens, são completamente diferentes dos documentos em papel. Os documentos digitais são extremamente fáceis de gerar, copiar e alterar. Simplesmente anexar a identidade do criador, seja como um *string* de texto, uma fotografia ou uma imagem manuscrita, não tem nenhum valor para propósitos de verificação.

O que é necessário é uma maneira de vincular irrevogavelmente a identidade de um signatário à sequência de bits inteira que representa um documento. Isso deve satisfazer o primeiro requisito acima, o da autenticidade. Assim como acontece com as assinaturas manuscritas, a data de um documento não é garantida pela assinatura. O destinatário de um documento assinado sabe apenas que ele foi assinado antes de tê-lo recebido.

A respeito do não repúdio, há um problema que não surge com as assinaturas manuscritas: e se o signatário revelar deliberadamente sua chave privada e, subsequentemente, negar que o assinou, alegando que existem outras pessoas que poderiam ter feito isso, porque a chave não era realmente privada? Foram desenvolvidos alguns protocolos para tratar desse problema, sob o título de *assinaturas digitais inegáveis* [Schneier 1996], mas eles aumentam consideravelmente a complexidade.

Um documento com uma assinatura digital pode ser consideravelmente mais resistente à falsificação do que outro com assinatura manuscrita. Mas a palavra “original” tem pouco significado com referência aos documentos digitais. Conforme veremos em nossa discussão sobre as necessidades do comércio eletrônico, as assinaturas digitais, por si sós, não podem, por exemplo, impedir um saque duplo no caixa eletrônico – outras medidas são necessárias para evitar isso. Vamos descrever agora duas técnicas para assinar documentos de forma digital, vinculando a identidade de um principal ao documento. Ambas dependem do uso de criptografia.

Assinatura digital • Um documento, ou mensagem eletrônica M , pode ser assinada por um principal A cifrando-se uma cópia de M com uma chave K_A e anexando-se uma cópia de texto puro de M e o identificador de A . O documento assinado consiste, então, em: $M, A, [M]_{K_A}$. A

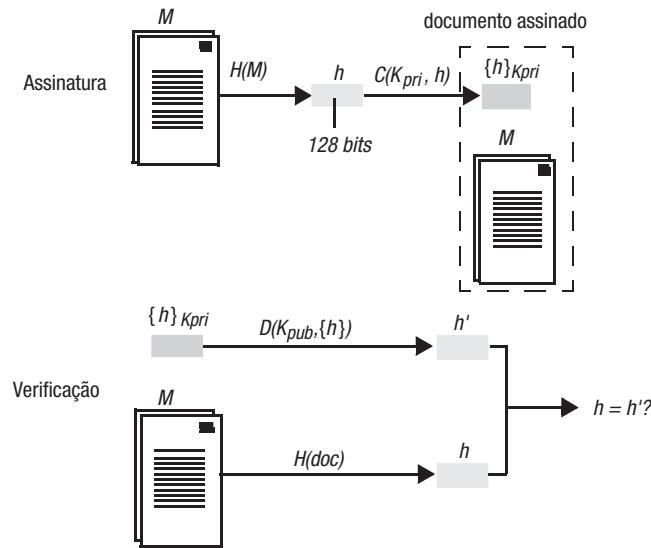


Figura 11.10 Assinatura digital com chaves públicas.

assinatura pode ser verificada por um principal que, subsequentemente, recebe o documento para conferir se ele foi originado por A e se seu conteúdo, M , não foi alterado.

Se uma chave secreta for usada para cifrar o documento, apenas os principais que compartilharem o segredo poderão verificar a assinatura. Contudo, se for usada criptografia de chave pública, então o signatário usará sua chave privada e qualquer um que possua a chave pública correspondente poderá verificar a assinatura. Isso é melhor que as assinaturas convencionais e atende a uma variedade mais ampla das necessidades do usuário. A verificação de assinaturas prossegue de diferentes formas, dependendo se é usada criptografia de chave secreta ou de chave pública para produzir a assinatura. Descreveremos os dois casos nas Seções 11.4.1 e 11.4.2.

Funções de resumo (digest) • As funções de resumo também são chamadas de *funções de hashing seguras* e denotadas como $H(M)$. Elas devem ser cuidadosamente projetadas para garantir que $H(M)$ seja diferente de $H(M')$ para todos os prováveis pares de mensagens M e M' . Se houver quaisquer pares de mensagens M e M' diferentes, tal que $H(M) = H(M')$, então um principal fraudulento poderia enviar uma cópia assinada de M , mas quando confrontada com ela, reivindicar que M' foi originalmente enviada e que ela deve ter sido alterada em trânsito. Discutiremos algumas funções de *hashing* seguras na Seção 11.4.3.

11.4.1 Assinaturas digitais com chaves públicas

A criptografia de chave pública é particularmente bem adaptada para a geração de assinaturas digitais, pois ela é relativamente simples e não exige nenhuma comunicação entre o signatário e o destinatário de um documento assinado ou outra pessoa qualquer.

O método para A assinar uma mensagem M e para B verificar a é o seguinte (e está ilustrado graficamente na Figura 11.10):

1. A gera um par de chaves K_{pub} e K_{priv} e publica a chave pública K_{pub} , colocando-a em um local bem conhecido.

2. A calcula o resumo de M , $H(M)$, usando uma função de resumo segura H e previamente combinada com B , o resumo é cifrado com a chave privada K_{priv} para produzir a assinatura $S = \{H(M)\}K_{priv}$.
3. A envia a mensagem assinada $[M]_K = M, S$ para B .
4. B decifra S usando K_{pub} e calcula o resumo de M , $H(M)$. Se eles corresponderem, a assinatura é válida.

O algoritmo RSA é muito conveniente para a construção de assinaturas digitais. Note que a chave *privada* do signatário é usada para cifrar a assinatura, diferentemente de quando o objetivo é transmitir informações em segredo, em que é usada a chave *pública* do destinatário para cifrar a mensagem. A explicação dessa diferença é simples e direta – uma assinatura deve ser criada usando-se um segredo conhecido apenas pelo signatário e ela deve estar acessível para todos para verificação.

11.4.2 Assinaturas digitais com chaves secretas

Não existe nenhum motivo técnico pelo qual um algoritmo de criptografia de chave secreta não possa ser usado para cifrar uma assinatura digital, mas para verificar tais assinaturas, a chave precisa ser revelada, e isso causa alguns problemas:

- O signatário deve fazer preparativos para o destinatário receber com segurança a chave secreta usada para assinatura.
- Talvez seja necessário verificar uma assinatura em vários contextos e em diferentes momentos – na hora da assinatura, o signatário pode não saber as identidades dos destinatários. Para resolver isso, a verificação poderia ser delegada a um terceiro confiável que possua as chaves secretas de todos os signatários, mas isso aumenta a complexidade do modelo de segurança e exige uma comunicação segura com o terceiro confiável.
- A exposição de uma chave secreta usada para assinar é indesejável, pois isso enfraquece a segurança das assinaturas feitas com essa chave – uma assinatura poderia ser falsificada por um possuidor da chave que não seja o proprietário dela.

Por todos esses motivos, o método de chave pública para geração e verificação de assinaturas oferece a solução mais conveniente na maioria das situações.

Uma exceção surge quando um canal seguro é usado para transmitir mensagens não cifradas, mas com a necessidade de verificar a autenticidade das mensagens. Como o canal fornece comunicação segura entre dois processos, uma chave secreta compartilhada pode ser estabelecida usando-se o método misto apresentado na Seção 11.3.3, e com ela pode-se produzir assinaturas de baixo custo. Essas assinaturas são chamadas de *códigos de autenticação de mensagem (MAC, Message Authentication Codes)* para refletir seu propósito mais limitado – elas autenticam a comunicação entre pares de principais com base em um segredo compartilhado.

Uma técnica de assinatura de baixo custo, baseada em chaves secretas compartilhadas, que tem segurança adequada para muitos propósitos, está ilustrada na Figura 11.11 e delineada a seguir. O método depende da existência de um canal seguro por meio do qual a chave compartilhada possa ser distribuída:

1. A gera uma chave aleatória K para assinatura e a distribui usando canais seguros para um ou mais principais que autenticarão as mensagens recebidas de A . Os principais são *confiáveis* e não vão revelar a chave compartilhada.

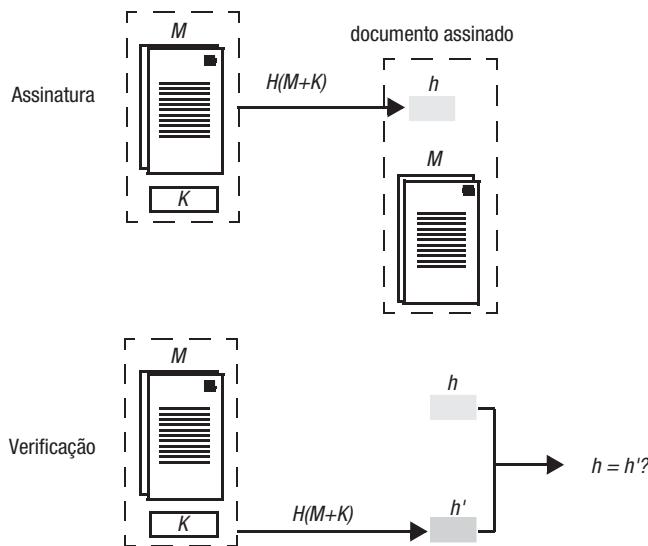


Figura 11.11 Assinaturas de baixo custo com uma chave secreta compartilhada.

2. Para qualquer documento M que A queira assinar, A concatena M com K , calcula o resumo do resultado $h = H(M + K)$ e envia o documento assinado $[M]_K = M, h$ para todos que quiserem verificar a assinatura. (O resumo h é um MAC.) K não será comprometida pela revelação de h , pois a função de resumo ocultou seu valor totalmente.
3. O receptor, B , concatena a chave secreta K com o documento M recebido e calcula o resumo $h' = H(M + K)$. A assinatura é verificada se $h = h'$.

Embora esse método sofra das desvantagens listadas anteriormente, ele tem uma vantagem de desempenho, pois não envolve nenhuma criptografia. (Normalmente, o resumo seguro é cerca de 3 a 10 vezes mais rápido do que a criptografia simétrica, veja a Seção 11.5.1.) O protocolo de canal seguro TLS, descrito na Seção 11.6.3, suporta o uso de uma ampla variedade de MACs, incluindo o esquema descrito aqui. O método também é usado no protocolo de caixa eletrônico Millicent, descrito no endereço www.cdk5.net/security, em que é importante manter baixo o custo do processamento para transações de baixo valor.

11.4.3 Funções de resumo seguras

Existem muitas maneiras de produzir um padrão de bits de comprimento fixo que caracterize uma mensagem ou um documento de comprimento arbitrário. Talvez a mais simples seja usar a operação XOR iterativamente, para combinar partes de comprimento fixo do documento de origem. Tal função é frequentemente usada em protocolos de comunicação para produzir um resumo de comprimento fixo (curto) para caracterizar uma mensagem com propósitos de detecção de erro, mas ela é inadequada como base para um esquema de assinatura digital. Uma função de resumo segura $h = H(M)$ deve ter as seguintes propriedades:

1. Dado M , é fácil calcular h
2. Dado h , é difícil calcular M .
3. Dado M , é difícil encontrar outra mensagem M' , tal que $H(M) = H(M')$.

Tais funções também são chamadas de *funções de resumo de mão única*. O motivo desse nome é evidente, com base nas duas primeiras propriedades. A propriedade 3 exige uma característica adicional: mesmo sabendo que o resultado de uma função de resumo não pode ser único (porque o resumo é uma transformação de redução da informação), precisamos garantir que um invasor, dada uma mensagem M que produza um resumo h , não possa descobrir outra mensagem M' que também produza h . Se um invasor pudesse fazer isso, então ele poderia falsificar um documento assinado M' sem conhecimento da chave de assinatura, copiando a assinatura do documento assinado M e anexando-a em M' .

Admitidamente, o conjunto de mensagens que produz um resumo de mesmo valor é restrito, e o invasor teria dificuldade para produzir uma falsificação significativa, mas com paciência, ela poderia ser feita; portanto, é preciso tomar cuidado com isso. A possibilidade de se fazer isso é considerável no caso do *ataque de data de nascimento (birthday attack)*:

1. Alice prepara duas versões, M e M' , de um contrato para Bob. M é favorável a Bob e M' , não.
2. Alice faz diversas versões sutilmente diferentes de M e de M' , que são visualmente indistinguíveis entre si, por meio de métodos como a adição de espaços nos finais das linhas. Ela compara os resumos de todos os M s com todos os M' s. Se encontrar dois que sejam iguais, ela pode passar para o próximo passo; caso contrário, ela continua produzindo versões visualmente indistinguíveis dos dois documentos, até obter uma correspondência.
3. Quando tiver dois documentos M e M' com o mesmo valor de resumo, ela envia o documento M favorável para Bob para que ele produza, usando sua chave privada, uma assinatura digital para esse documento. Quando ele o retorna, ela o substitui pela versão desfavorável M' correspondente, mantendo a assinatura de M .

Se nossos valores de resumo tiverem 64 bits, serão necessárias apenas, em média, de 2^{32} versões de M e M' . Isso é muito pouco para nos tranquilizar. Precisamos tornar nossos valores de resumo pelo menos 128 bits para nos protegermos desse ataque.

O ataque conta com um paradoxo estatístico conhecido como *paradoxo da data de nascimento* – a probabilidade de encontrar um par correspondente em determinado conjunto é bem maior do que a de encontrar uma correspondência para determinado indivíduo. Stallings [2005] fornece a derivação estatística da probabilidade de que existam duas pessoas com a mesma data de nascimento em um conjunto de n pessoas. Resumidamente, o resultado é que, para um conjunto de apenas 23 pessoas, as chances de se encontrar duas pessoas que fazem aniversário no mesmo dia são as mesmas para qualquer par de pessoas, enquanto é necessário um conjunto de 253 pessoas para se ter a mesma chance de se encontrar uma pessoa com uma data de nascimento em um determinado dia.

Para satisfazer as propriedades listadas anteriormente, uma função de resumo segura precisa ser cuidadosamente projetada. As operações em nível de bit usadas, e sua disposição em sequência, são semelhantes àquelas encontradas na criptografia simétrica, mas, neste caso, as operações não precisam preservar as informações, pois a função não é destinada a ser reversível. Portanto, uma função de resumo segura pode usar toda a gama de operações aritméticas e lógicas em nível de bit. O comprimento do texto de origem normalmente é incluído nos dados do resumo.

<i>Sujeito</i>	Nome Discriminado, Chave Pública
<i>Emitente</i>	Nome Discriminado, Assinatura
<i>Período de validade</i>	Não Antes de Data, Não Após Data
<i>Informações administrativas</i>	Versão, Número de Série
<i>Informações estendidas</i>	

Figura 11.12 Formato do certificado X.509.

Duas funções de resumo amplamente usadas para aplicações práticas são o algoritmo MD5 (assim chamado porque ele é o quinto em uma sequência de algoritmos de resumos de mensagem desenvolvidos por Ron Rivest) e o SHA-1 (Secure Hash Algorithm), adotado para padronização pelo National Institute for Standards and Technology (NIST) dos Estados Unidos. Ambas foram cuidadosamente testadas e analisadas e podem ser consideradas adequadamente seguras dentro de um futuro próximo, enquanto suas implementações são razoavelmente eficientes. Nós as descreveremos sucintamente aqui. Schneier [1996] e Mitchell *et al.* [1992] fazem um levantamento profundo das técnicas de assinatura digital e das funções de resumo de mensagem.

MD5 • O algoritmo MD5 [Rivest 1992a] usa quatro passagens, cada uma aplicando uma de quatro funções não lineares em cada um dos 16 segmentos de 32 bits de um bloco de texto de origem de 512 bits. O resultado é um resumo de 128 bits. O MD5 é um dos algoritmos mais eficientes atualmente disponíveis.

SHA-1 • O SHA-1 [NIST 2002] é um algoritmo que produz um resumo de 160 bits. Ele é baseado no algoritmo MD4 de Rivest (que é semelhante ao MD5), com algumas operações adicionais. Ele é significativamente mais lento do que o MD5, mas o resumo de 160 bits apresenta uma segurança maior contra ataques do estilo força bruta e data de nascimento. Algoritmos SHA que geram resumos maiores (224, 256 e 512 bits) também estão incluídos no padrão [NIST 2002]. É claro que seu comprimento adicional acarreta custos a mais para a geração, o armazenamento e a comunicação de assinaturas digitais e MACs, mas após a publicação de ataques nos predecessores do SHA-1, que sugeriram que o SHA-1 é vulnerável [Randall e Szydlo 2004], o NIST anunciou que ele vai ser substituído por versões de resumo SHA mais longas nos *softwares* do governo dos Estados Unidos [NIST 2004].

Uso de um algoritmo de criptografia para fazer um resumo • É possível usar um algoritmo de criptografia simétrico, como aqueles detalhados na Seção 11.3.1, para produzir um resumo seguro. Nesse caso, a chave deve ser publicada para que o algoritmo de resumo possa ser aplicado por qualquer um que queira verificar uma assinatura digital. O algoritmo de criptografia é usado no modo CBC, e o resumo é o resultado da combinação do penúltimo valor de CBC com o bloco cifrado final.

11.4.4 Padrões de certificado e autoridades certificadoras

O X.509 é o formato padrão mais usado para certificados digitais [CCITT 1988b]. Embora o formato de certificado X.509 faça parte do padrão X.500 para a construção de diretórios globais de nomes e atributos [CCITT 1988a], ele é comumente usado em trabalhos de criptografia como uma definição de formato para certificados. Descreveremos o padrão de atribuição de nomes X.500 no Capítulo 13.

A estrutura e o conteúdo de um certificado X.509 estão ilustrados na Figura 11.12. Conforme vemos, ele vincula uma chave pública a uma entidade nomeada, chamada de *sujeito* (*subject*). A vinculação se dá na assinatura, que é distribuída por outra entidade nomeada, chamada de *emitente* (*issuer*). O certificado tem um *período de validade*, que é definido por duas datas. As entradas *<Nome Discriminado>* (*Distinguished Name*) se destinam a ser o nome de uma pessoa, organização, ou outra entidade, junto a informações contextuais suficientes para torná-las exclusivas. Em uma implementação completa do X.500, essas informações contextuais são extraídas de uma hierarquia global de diretórios na qual a entidade nomeada aparece, mas na ausência de implementações de X.500 globais, elas podem ser apenas um *string* descritivo.

Esse formato é incluído no protocolo TLS para comércio eletrônico e, na prática, é amplamente usado para autenticar as chaves públicas de serviços e seus clientes. Certas empresas e organizações bem conhecidas se estabeleceram para atuar como *autoridades certificadoras* (por exemplo, Verisign [www.verisign.com], CREN [www.cren.net]) e outras empresas e indivíduos, enviando evidência satisfatória de suas identidades, podem obter com elas certificados de chave pública X.509. Isso leva a um procedimento de verificação de duas etapas para qualquer certificado X.509:

1. Obter, de uma fonte confiável, o certificado de chave pública do emitente (uma autoridade de certificação).
2. Validar a assinatura.

A estratégia SPKI • A estratégia X.509 é baseada na exclusividade global de nomes discriminados. Mostrou-se que esse é um objetivo impraticável que não reflete a realidade da prática jurídica e comercial corrente [Ellison 1996], na qual não se presume que a identidade dos indivíduos seja exclusiva, mas que se torna exclusiva pela referência a outras pessoas e organizações. Isso pode ser visto no uso de uma carteira de motorista, ou de uma carta de um banco, para autenticar o nome e o endereço de um indivíduo (é improvável que apenas um nome seja exclusivo dentre a população mundial). Isso leva a sequências de verificação mais longas, pois existem muitos emitentes possíveis de certificados de chave pública, e suas assinaturas devem ser validadas por meio de um encadeamento de verificação que leve de volta a alguém conhecido, e de confiança, do principal que está realizando a verificação. No entanto, provavelmente, a verificação resultante será mais convincente, e muitas das etapas desse encadeamento podem ser colocadas em cache para encurtar o processo em ocasiões futuras.

Os argumentos anteriores são a base das propostas SPKI (Simple Public-Key Infrastructure) recentemente desenvolvidas (veja RFC 2693 [Ellison *et al.* 1999]). Trata-se de um esquema para a criação e gerenciamento de conjuntos de certificados públicos. Ele permite que encadeamentos de certificados sejam processados usando inferência lógica para produzir certificados derivados. Por exemplo, “Bob acredita que a chave pública de Alice é K_{Apub} ” e “Carol confia em Bob a respeito das chaves de Alice”, implica que “Carol acredita que a chave pública de Alice é K_{Apub} ”.

11.5 Criptografia na prática

Na Seção 11.5.1, compararemos o desempenho dos algoritmos de criptografia e de resumo seguro descritos ou mencionados anteriormente. Consideraremos os algoritmos de criptografia lado a lado com as funções de resumo seguras, pois a criptografia também pode ser usada como um método de assinatura digital.

	<i>Tamanho da chave/tamanho do resumo (bits)</i>	<i>PRB otimizado Pentium 1 de 90 Mhz (Mbytes/s)</i>	<i>Crypto++ Pentium 4 de 2,1 GHz (Mbytes/s)</i>
TEA	128	–	23,801
DES	56	2,113	21,340
Triple-DES	112	0,775	9,848
IDEA	128	1,219	18,963
AES	128	–	61,010
AES	192	–	53,145
AES	256	–	48,229
MD5	128	17,025	216,674
SHA-1	160	–	67,977

Figura 11.13 Desempenho dos algoritmos de criptografia simétrica e de resumo seguros.

Na Seção 11.5.2, discutiremos alguns problemas não técnicos que cercam o uso da criptografia. Não temos espaço para tratar adequadamente do vasto volume de discussão política que tem ocorrido sobre esse assunto desde que os algoritmos de criptografia poderosos apareceram pela primeira vez em domínio público; tampouco os debates chegaram a muitas conclusões definitivas. Nosso objetivo é simplesmente dar ao leitor algum conhecimento desse debate.

11.5.1 Desempenho dos algoritmos de criptografia

A Figura 11.13 mostra a velocidade dos algoritmos de criptografia simétricos e as funções de resumo seguras que discutimos neste capítulo. Onde estão disponíveis, fornecemos duas medidas de velocidade. Na coluna rotulada como *PRB otimizado*, fornecemos valores baseados naqueles publicados por Preneel *et al.* [1998]. Os valores presentes na coluna rotulada como *Crypto++* foram obtidos muito mais recentemente pelos autores da biblioteca de criptografia Crypto++, de código-fonte aberto [www.cryptopp.com]. Os cabeçalhos de coluna indicam a velocidade do *hardware* usado para esses comparativos. As implementações de Preneel são programas em assembly e otimizados manualmente, enquanto as da Crypto++ são programadas em C++ gerados com um compilador com otimização de código.

Os comprimentos de chave dão uma indicação do custo computacional de um ataque de força bruta contra a chave; o poder real dos algoritmos de criptografia é muito mais difícil de avaliar e baseia-se no raciocínio sobre o êxito do algoritmo na ocultação do texto puro. Preneel *et al.* [1998] apresentam uma discussão interessante sobre o poder e o desempenho dos principais algoritmos simétricos.

O que esses valores de desempenho significam para aplicações reais de criptografia, como o seu uso no esquema TLS para interações seguras na Web (o protocolo *https*, descrito na Seção 11.6.3)? Raramente as páginas Web são maiores do que 100 quilobytes; portanto, o conteúdo de uma página pode ser cifrado em poucos milisegundos usando-se qualquer um dos algoritmos simétricos, mesmo com um processador considerado muito lento pelos padrões atuais. O RSA é usado principalmente para assinaturas digitais, e esse passo também pode ser dado em poucos milisegundos. Assim, o impacto do desempenho do algoritmo na velocidade sentida por uma aplicação que usa o protocolo *https* é mínimo.

Os algoritmos assimétricos, como o RSA, raramente são usados para criptografia de dados, mas seu desempenho para assinatura é interessante. As páginas da biblioteca Crypto++ indicam que, com o *hardware* mencionado na última coluna da Figura 11.13, demora cerca de 4,75 ms, usando RSA com uma chave de 1024 bits, para assinar um resumo seguro (presumivelmente usando SHA-1 de 160 bits) e cerca de 0,18 ms para verificar a assinatura.

11.5.2 Aplicações da criptografia e obstáculos políticos

Todos os algoritmos descritos anteriormente surgiram durante os anos 80 e 90, quando as redes de computadores estavam começando a ser usadas para propósitos comerciais e estava se tornando evidente que sua falta de segurança era um grande problema. Conforme mencionamos na introdução deste capítulo, a revelação do *software* de criptografia teve forte resistência por parte do governo norte-americano. A resistência tinha duas origens: a Agência de Segurança Nacional dos Estados Unidos (NSA, National Security Agency), que achava que deveria haver um plano para restringir o poder da criptografia disponível para outros países a um nível que a NSA pudesse decifrar qualquer comunicação secreta para propósitos da inteligência militar; e o FBI (Federal Bureau of Investigation), que pretendia garantir que seus agentes pudessem ter acesso privilegiado às chaves de criptografia usadas por todas as organizações privadas e indivíduos dos Estados Unidos, para propósitos de cumprimento da lei.

O *software* de criptografia era classificado como munição nos Estados Unidos e estava sujeito a rigorosas restrições de exportação. Outros países, especialmente aliados dos Estados Unidos, aplicaram restrições semelhantes ou, em alguns casos, ainda mais rigorosas. O problema era composto pela ignorância geral dentre os políticos, e o público em geral, com relação ao que era *software* de criptografia e suas aplicações não militares em potencial. As empresas de *software* norte-americanas protestaram, dizendo que as restrições inibiam a exportação de *software* como os navegadores, e as restrições de exportação foram finalmente reformuladas para permitir a exportação de código usando chaves de não mais do que 40 bits – dificilmente uma criptografia forte!

As restrições de exportação podem ter atrapalhado o crescimento do comércio eletrônico, mas não eram particularmente eficazes em impedir a divulgação da experiência em criptografia, nem em manter o *software* de criptografia fora das mãos de usuários de outros países, pois muitos programadores, dentro e fora dos Estados Unidos, estavam ansiosos e eram capazes de implementar e distribuir código de criptografia. A posição atual é que *softwares* que implementam a maioria dos principais algoritmos de criptografia estão disponíveis há vários anos para o mundo todo, impressos [Schneier 1996], *online*, em versões comerciais e *freeware* [www.rsasecurity.com, cryptography.org, privacy.nb.ca, www.openssl.org].

Um exemplo é o programa chamado PGP (Pretty Good Privacy) [Garfinkel 1994, Zimmermann 1995], originalmente desenvolvido por Philip Zimmermann e levado adiante por ele e outros. Isso é parte da campanha de técnicos e políticos para garantir que a disponibilidade de métodos de criptografia não seja controlada pelo governo norte-americano. O PGP foi desenvolvido e distribuído com o objetivo de permitir que todos os usuários de computador usufruíssem do nível de privacidade e integridade proporcionado pelo uso de criptografia de chave pública em suas comunicações. O PGP gera e gerencia chaves públicas e secretas para um usuário. Ele usa criptografia de chave pública RSA para autenticação e para transmitir chaves secretas a um interlocutor e usa os algoritmos de criptografia de chave secreta IDEA ou 3DES para cifrar mensagens de correio eletrônico ou outros documentos. (Quando o PGP foi desenvolvido, o uso do

algoritmo DES era controlado pelo governo norte-americano.) O PGP está amplamente disponível em versões gratuitas e comerciais. Ele é distribuído por intermédio de *sites* de distribuição separados para usuários norte-americanos [www.pgp.com] e para os de outras partes do mundo [International PGP], para burlar (de maneira perfeitamente legal) as normas de exportação dos Estados Unidos.

Finalmente, o governo norte-americano reconheceu a futilidade da posição da NSA e o dano que estava causando à indústria da computação dos Estados Unidos (que não podia comercializar versões seguras de navegadores Web, sistemas operacionais distribuídos e muitos outros produtos no mundo todo). Em janeiro de 2000, o governo norte-americano apresentou um novo plano [www.bxa.doc.gov], destinado a permitir que os fornecedores de *software* dos Estados Unidos exportassem *software* incorporando criptografia forte, mas foi mantida uma barreira legal para a entrega a certos países ou usuários finais – acesse www.rsa.com para mais informações. É claro que os Estados Unidos não detêm o monopólio sobre a produção ou a publicação de *software* de criptografia; estão disponíveis implementações de código-fonte aberto para todos os algoritmos conhecidos [www.cryptopp.com]. O efeito das normas é simplesmente atrapalhar a comercialização de alguns produtos de *software* comerciais produzidos nos Estados Unidos.

Outras iniciativas políticas tiveram como objetivo manter o controle sobre o uso da criptografia, apresentando uma legislação que insistia na inclusão de brechas ou pontos de entradas disponíveis apenas para os órgãos governamentais jurídicos e de segurança. Tais propostas procedem da percepção de que canais de comunicação secretos podem ser muito úteis para criminosos de todos os tipos. Antes do advento da criptografia digital, os governos sempre tinham um meio de interceptar e analisar as comunicações entre os membros do público. A criptografia digital forte altera radicalmente essa situação. No entanto, essas propostas de legislação para impedir o uso de criptografia forte têm grande resistência por parte dos cidadãos e dos organismos de liberdades civis, que estão preocupados com seu impacto sobre os direitos à privacidade das pessoas. Até agora, nenhuma dessas propostas legislativas foi adotada, mas os esforços políticos continuam, e a eventual introdução de uma estrutura jurídica para o uso de criptografia pode ser inevitável.

11.6 Estudos de caso: Needham–Schroeder, Kerberos, TLS, 802.11 WiFi

Os protocolos de autenticação originalmente publicados por Needham e Schroeder [1978] estão no centro de muitas técnicas de segurança. Nós os detalharemos na Seção 11.6.1. Uma das aplicações mais importantes do protocolo de autenticação de chave secreta deles é o sistema Kerberos [Neuman e Ts'o 1994], que será o assunto de nosso segundo estudo de caso (Seção 11.6.2). O Kerberos foi projetado para fornecer autenticação entre clientes e servidores em redes que formam um único domínio de gerenciamento (*intranets*).

Nosso terceiro estudo de caso (Seção 11.6.3) trata do protocolo TLS (Transport Layer Security). Ele foi projetado especificamente para atender à necessidade de transações eletrônicas seguras. Atualmente, ele é suportado pela maioria dos navegadores e servidores Web e é empregado em muitas transações comerciais que ocorrem na Web.

Nosso estudo de caso final (Seção 11.6.4) ilustra a dificuldade de construir sistemas seguros. O padrão IEEE 802.11 WiFi foi publicado em 1999 com uma especificação de segurança incluída. Contudo, análise e ataques subsequentes mostraram que a especificação era muito inadequada. Identificaremos as deficiências e as relacionaremos aos princípios de criptografia abordados neste capítulo.

11.6.1 O protocolo de autenticação Needham–Schroeder

Os protocolos descritos aqui foram desenvolvidos em resposta à necessidade de um meio seguro de gerenciar chaves (e senhas) em uma rede. Quando o trabalho foi publicado [Needham e Schroeder 1978], os servidores de arquivos estavam surgindo e havia a necessidade urgente de gerenciar de melhor maneira a segurança em redes locais.

Nas redes gerenciadas de forma integrada, essa necessidade pode ser satisfeita por um serviço de chaves seguro, que publica chaves de sessão na forma de desafios (veja a Seção 11.2.2). Esse é o objetivo do protocolo de chave secreta desenvolvido por Needham e Schroeder. No mesmo artigo, Needham e Schroeder também relatam um protocolo baseado no uso de chaves públicas para autenticação e distribuição de chaves que não depende da existência de servidores de chaves seguros e que, portanto, é mais conveniente para uso em redes com muitos domínios de gerenciamento independentes, como a Internet. Não vamos descrever a versão de chave pública aqui, mas o protocolo TLS descrito na Seção 11.6.3 é uma variação dela.

Needham e Schroeder propuseram uma solução para a autenticação e distribuição de chaves baseada em um *servidor de autenticação* que fornece chaves secretas para os clientes. A tarefa do servidor de autenticação é proporcionar uma forma segura para pares de processos obterem chaves compartilhadas. Para fazer isso, ele precisa se comunicar com seus clientes usando mensagens cifradas.

Needham e Schroeder com chaves secretas • Nesse modelo, um processo atuando em nome de um principal *A*, que queira iniciar uma comunicação segura com outro processo atuando em nome de um principal *B*, pode obter uma chave para tal. O protocolo é descrito para dois processos arbitrários *A* e *B*, mas, nos sistemas cliente-servidor, *A* pode ser um cliente iniciando uma sequência de requisições para um servidor *B*. Nesse protocolo, o processo *A* recebe duas informações para serem usadas na interação com o processo *B*, uma chave de sessão usada para cifrar mensagens para *B* e um *tíquete* que ele pode transmitir com segurança para *B*. (Este último é cifrado com uma chave conhecida por *B*, mas não por *A*, para que *B* possa decifrá-la sem que a chave de sessão seja comprometida durante a transmissão.)

O servidor de autenticação *S* mantém uma tabela contendo um nome e uma chave secreta para cada principal conhecido pelo sistema. A chave secreta é usada para autenticar processos clientes no servidor de autenticação e para transmitir mensagens com segurança entre processos clientes e o servidor de autenticação. Ela nunca é revelada para terceiros e é transmitida pela rede no máximo uma vez, ao ser gerada. (De preferência, uma chave sempre deve ser transmitida por algum outro meio, como no papel ou em uma mensagem verbal, evitando qualquer exposição na rede.) Uma chave secreta é equivalente à senha usada para autenticar usuários em sistemas centralizados. Para usuários humanos atuando como principais, o nome mantido pelo serviço de autenticação é o nome de usuário, e a chave secreta é a senha. Ambos são fornecidos pelo usuário para os processos clientes que atuam em seu nome.

O protocolo é baseado na geração e transmissão de tíquetes pelo servidor de autenticação. Um tíquete é uma mensagem cifrada contendo uma chave secreta (de sessão) para uso na comunicação entre *A* e *B*. Na Figura 11.14, tabulamos as mensagens do protocolo de chave secreta de Needham e Schroeder. O servidor de autenticação é *S*.

N_A e N_B são *números usados uma vez*. Na terminologia de protocolos baseados em desafio-resposta, esses números são denominados *nonce*. Um *nonce* é um valor inteiro incluído em uma mensagem para demonstrar que ela é nova. Os *nonces* são

Cabeçalho	Mensagem	Notas
1. A → S:	A, B, N_A	A pede a S para que forneça uma chave para comunicação com B .
2. S → A:	$\{N_A, B, K_{AB}, \{K_{AB}, A\}_{K_B}\}_{K_A}$	S retorna uma mensagem cifrada com a chave secreta de A , contendo uma chave recentemente gerada K_{AB} e um tiquete cifrado com chave secreta de B . O nonce N_A indica que a mensagem foi enviada em resposta à precedente. A acredita que S enviou a mensagem, pois somente S conhece a chave secreta de A .
3. A → B:	$\{K_{AB}, A\}_{K_B}$	A envia o tiquete para B .
4. B → A:	$\{N_B\}_{K_{AB}}$	B decifra o tiquete e usa a nova chave, K_{AB} , para cifrar o nonce N_B .
5. A → B:	$\{N_B - 1\}_{K_{AB}}$	A demonstra para B que foi o remetente da mensagem anterior, retornando uma transformação combinada de N_B .

Figura 11.14 O protocolo de chave secreta de autenticação Needham–Schroeder.

utilizados uma única vez e são obtidos sob demanda. Por exemplo, os *nonces* podem ser gerados como uma sequência de valores inteiros ou pela leitura do relógio da máquina remetente.

Se o protocolo for concluído com êxito, A e B podem ter certeza de que qualquer mensagem recebida cifrada com K_{AB} é proveniente um do outro e de que qualquer mensagem enviada cifrada com K_{AB} só pode ser entendida pelo outro ou pelo servidor S (e pressupõe-se que S é confiável). Isso funciona assim porque as únicas mensagens que foram enviadas contendo K_{AB} foram cifradas com a chave secreta de A ou com a chave secreta de B .

Existe uma deficiência nesse protocolo, pois B não tem motivos para acreditar que a mensagem do passo 3 é nova. Um intruso que consiga obter a chave K_{AB} e fazer uma cópia do tiquete e do autenticador $\{K_{AB}, A\}_{K_B}$ (ambos os quais podem ter sido deixados em um local de armazenamento exposto por um programa cliente descuidado, ou defeituoso, executando sob a autorização de A), pode usá-los para iniciar uma troca subsequente com B , personificando A . Para que esse ataque ocorra, um valor antigo de K_{AB} deve ser comprometido; usando a terminologia atual, Needham e Schroeder não incluíram essa possibilidade em sua lista de ameaças, e a opinião de consenso é que alguém deve fazer isso. A deficiência pode ser remediada pela inserção de um nonce ou de um carimbo de tempo na mensagem 3, para que ela se torne: $\{K_{AB}, A, t\}_{K_{Bpub}}$. B decifra essa mensagem e verifica se t é recente. Essa é a solução adotada no Kerberos.

11.6.2 Kerberos

O Kerberos foi desenvolvido no MIT, nos anos 80 [Steiner *et al.* 1988], com o objetivo de fornecer uma variedade de recursos de autenticação e segurança para uso na rede de computadores do campus daquela instituição e em outras intranets. Ele passou por várias revisões e aprimoramentos, à luz da experiência e dos informes de organizações de usuários. O Kerberos versão 5 [Neuman e Ts'o 1994], que descreveremos aqui, faz parte dos padrões de Internet (veja a RFC 4120 [Neuman *et al.* 2005]) e agora é usado por muitas

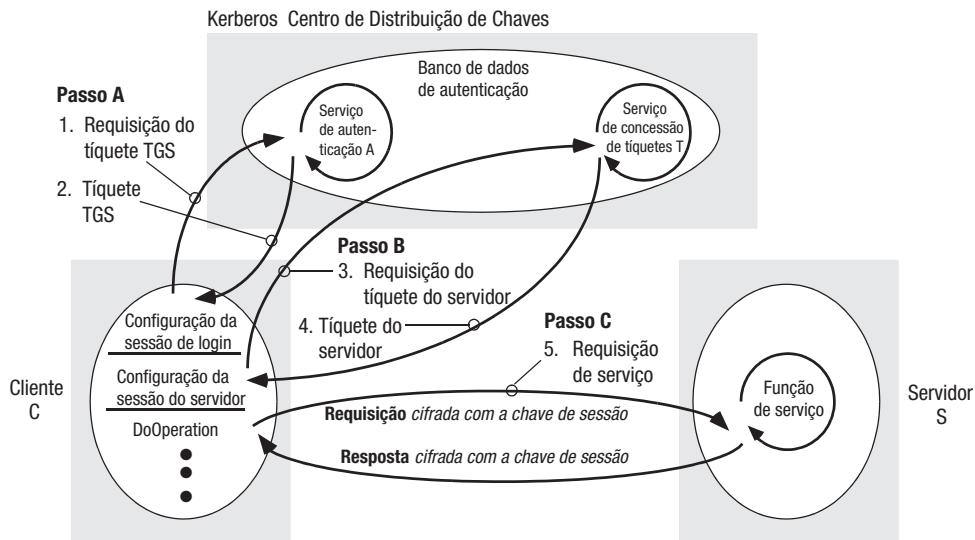


Figura 11.15 Arquitetura do sistema Kerberos.

empresas e universidades. O código-fonte para uma implementação do Kerberos está disponível no MIT [[web.mit.edu I](http://web.mit.edu/I)]; ele está incluído no DCE (Distributed Computing Environment) do OSF [OSF 1997] e no sistema operacional Microsoft Windows 2000 como serviço de autenticação padrão [www.microsoft.com II]. Foi proposta uma extensão para incorporar o uso de certificados de chave pública para a autenticação inicial de principais (Passo A, na Figura 11.15) [Neuman *et al.* 1999].

A Figura 11.15 mostra a arquitetura do processo. O Kerberos lida com três tipos de objeto de segurança:

Tíquete: uma ficha emitida pelo serviço de concessão de tíquetes do Kerberos para ser apresentada a um servidor específico para verificar se o remetente foi autenticado recentemente pelo Kerberos. Os tíquetes incluem um tempo de expiração e uma chave de sessão gerada para uso do cliente e do servidor.

Autenticador: dar uma ficha construída por um cliente e enviada a um servidor para provar a identidade do usuário e a validade de toda comunicação com esse servidor. Um autenticador só pode ser usado uma vez. Ele contém o nome do cliente e um carimbo de tempo e é cifrado na chave de sessão apropriada.

Chave de sessão: uma chave secreta gerada aleatoriamente pelo Kerberos e emitida para um cliente para ser usada na comunicação com um servidor em particular. A criptografia não é obrigatória para toda comunicação com servidores; a chave de sessão é usada para cifrar todos os autenticadores e a comunicação com os servidores que exigem isso (veja a seguir).

Os processos clientes devem possuir um tíquete e uma chave de sessão para cada servidor que usarem. Seria impraticável fornecer um novo tíquete e uma chave para cada interação cliente-servidor; portanto, a maioria dos tíquetes é concedida aos clientes com um tempo de validade de várias horas, para que eles possam ser usados na interação com um servidor em particular até que expirem.

Um servidor Kerberos é conhecido como KDC (Key Distribution Centre – Centro de Distribuição de Chaves). Cada KDC apresenta um Serviço de Autenticação (AS, Authentication Service) e um Serviço de Concessão de Tiquetes (TGS, Ticket-Granting Service). No *login*, os usuários são autenticados pelo AS, usando uma variação do método de senha segura para a rede, e o processo cliente que está atuando em nome do usuário recebe um *tíquete* e uma chave de sessão para comunicação com o TGS. Subsequentemente, o processo cliente original e seus descendentes podem usar o tíquete do TGS para obter tíquetes e chaves de sessão para servidores específicos.

O protocolo de Needham e Schroeder [1978] é seguido muito de perto no Kerberos, com valores de tempo (inteiros representando uma data e hora) utilizados como *nonces*. Isso tem dois objetivos:

- evitar a reprodução de mensagens antigas interceptadas na rede ou a reutilização de tíquetes antigos encontrados na memória de máquinas das quais o usuário autorizado se desconectou (os *nonces* foram utilizados para atingir esse objetivo em Needham e Schroeder);
- aplicar um tempo de validade aos tíquetes, permitindo ao sistema revogar direitos dos usuários quando, por exemplo, eles deixam de ser usuários autorizados do sistema.

A seguir, descreveremos os protocolos do Kerberos em detalhes, usando a notação definida abaixo. Primeiramente, descreveremos o protocolo por meio do qual o cliente obtém um tíquete e uma chave de sessão para acesso ao TGS.

Um tíquete do Kerberos tem um período de validade fixo, começando no tempo t_1 e terminando no tempo t_2 . Um tíquete para um cliente C acessar um servidor S tem a forma:

$\{C, S, t_1, t_2, K_{CS}\}_{K_S}$, que vamos denotar como $\{\text{ticket}(C, S)\}_{K_S}$

O nome do cliente é incluído no tíquete para evitar um possível uso por impostores, conforme veremos posteriormente. Os passos e os números de mensagem na Figura 11.15 correspondem àqueles da descrição da Tabela A. Note que a mensagem 1 não é cifrada e não inclui a senha de C . Ela contém um *nonce* que é utilizado para verificar a validade da resposta.

A. Obtém chave de sessão do Kerberos e tíquete TGS, uma vez por sessão de login		
Cabeçalho	Mensagem	Notas
1. $C \rightarrow A$: Requisição de tíquete TGS	C, T, n	O cliente C pede ao servidor de autenticação Kerberos A para que forneça um tíquete para comunicação com o serviço de concessão de tíquetes T .
2. $A \rightarrow C$: Chave de sessão e tíquete TGS	$\{K_{CT}, n\}_{K_C} \{ \text{ticket}(C, T) \}_{K_T}$ contendo C, T, t_1, t_2, K_{CT}	A retorna uma mensagem contendo um tíquete cifrado em sua chave secreta e uma chave de sessão para C , para usar com T . A inclusão do <i>nonce</i> n , cifrado com K_C , mostra que a mensagem vem do destinatário da mensagem 1, que deve conhecer K_C .

Notação:

- | | | | |
|-----|--|-------|---------------------------------------|
| A | Nome do serviço de autenticação do Kerberos. | n | Nonce. |
| T | Nome do serviço de concessão de tíquetes Kerberos. | t | Um carimbo de tempo. |
| C | Nome do cliente. | t_1 | Tempo inicial da validade do tíquete. |
| | | t_2 | Tempo final da validade do tíquete. |

A mensagem 2 às vezes é chamada de desafio (*challenge*), pois apresenta informações ao solicitante que são úteis apenas se ele conhecer a chave secreta de C , K_C . Um impostor que tente personificar C , enviando a mensagem 1, não poderá ir adiante, pois não conseguirá decifrar a mensagem 2. Para principais que são usuários, K_C é uma versão derivada da senha do usuário. O processo cliente pedirá para que o usuário digite sua senha e tentará decifrar a mensagem 2 com ela. Se o usuário fornecer a senha correta, o processo cliente obterá a chave de sessão K_{CT} e um tíquete válido para o serviço de concessão de tíquetes (TGS); caso contrário, ele obterá dados incoerentes. Os servidores têm suas próprias chaves secretas, conhecidas apenas pelo processo servidor relevante e pelo servidor de autenticação.

Quando um tíquete válido tiver sido obtido do serviço de autenticação, o cliente C poderá usá-lo, até que expire, quantas vezes quiser, para se comunicar com o serviço de concessão de tíquetes para obter tíquetes para outros servidores. Assim, para obter um tíquete para um servidor S , C constrói um autenticador cifrado K_{CT} , da forma:

$\{C, t\}_{K_{CT}}$, que vamos denotar como $\{\text{auth}(C)\}_{K_{CT}}$, e envia um pedido para T :

B. Obtém tíquete para um servidor S, uma vez por sessão cliente-servidor		
<i>Cabeçalho</i>	<i>Mensagem</i>	<i>Notas</i>
3. $C \rightarrow T$: Requisita tíquete para o serviço S	$\{\text{auth}(C)\}_{K_{CT}}, \{\text{ticket}(C, T)\}_{K_T}, S, n$	C pede ao servidor de concessão de tíquetes T para que forneça um tíquete para comunicação com outro servidor S .
4. $T \rightarrow C$: Tíquete de serviço	$\{K_{CS}, n\}_{K_{CT}}, \{\text{ticket}(C, S)\}_{K_S}$	T verifica o tíquete. Se ele for válido, T gerará uma nova chave de sessão aleatória, K_{CS} , e a retornará com um tíquete para S (cifrado com a chave secreta, K_S , do servidor).

Então, C está pronto para emitir mensagens de requisição para o servidor S :

C. Emite um pedido de servidor com um tíquete		
<i>Cabeçalho</i>	<i>Mensagem</i>	<i>Notas</i>
5. $C \rightarrow S$: Requisição de serviço	$\{\text{auth}(C)\}_{K_{CS}}, \{\text{ticket}(C, S)\}_{K_S}, \text{request}, n$	C envia o tíquete para S , com um autenticador recentemente gerado para C e uma requisição. A requisição pode ser cifrada com K_{CS} se for exigido segredo dos dados.

Para o cliente certificar-se da autenticidade do servidor, S deve retornar para C o *nonce* n . (Para reduzir o número de mensagens exigidas, isso poderia ser incluído nas mensagens que contêm a resposta do servidor à requisição):

D. Autentica o servidor (opcional)		
<i>Cabeçalho</i>	<i>Mensagem</i>	<i>Notas</i>
6. $S \rightarrow C$: Autenticação do servidor	$\{n\}_{K_{CS}}$	(Opcional): S envia o <i>nonce</i> para C , cifrado com K_{CS} .

Aplicação do Kerberos • O Kerberos foi desenvolvido para uso no Projeto Athena do MIT – um recurso de computação de rede para os alunos, abrangendo todo o campus, com muitas estações de trabalho e servidores fornecendo serviço para mais de 5.000 usuários. O ambiente é tal que nem a integridade de caráter dos clientes, nem a se-

gurança da rede e das máquinas que oferecem serviços, podem ser presunidas – por exemplo, as estações de trabalho não estão protegidas contra a instalação de *software* desenvolvido por usuários, e as máquinas servidoras (outras, que não o servidor Kerberos) não são necessariamente seguras contra a interferência física na configuração de seu *software*.

O Kerberos fornece praticamente toda a segurança do sistema Athena. Ele é usado para autenticar usuários e outros principais. A maioria dos servidores em execução na rede foi estendida para exigir um tiquete de cada cliente no início de cada interação cliente-servidor. Isso inclui sistemas de arquivos (NFS e Andrew File System), correio eletrônico, *login* remoto e impressão. As senhas dos usuários são conhecidas apenas por eles próprios e pelo serviço de autenticação do Kerberos. Os serviços têm chaves secretas conhecidas apenas pelo Kerberos e pelos servidores que fornecem o serviço.

Vamos descrever a maneira pela qual o Kerberos é aplicado na autenticação de usuários no momento do *login*. Seu uso para tornar o serviço de arquivos NFS seguro será descrito no Capítulo 12.

Login com Kerberos • Quando um usuário se conecta em uma estação de trabalho, o programa de *login* envia o nome do usuário para o serviço de autenticação do Kerberos. Se o usuário for conhecido do serviço de autenticação, este responderá com uma chave de sessão e um *nonce*, cifrados com base na senha do usuário, e um tiquete para o TGS. Então, o programa de *login* tenta decifrar a chave de sessão e o *nonce*, utilizando a senha fornecida pelo usuário em resposta ao *prompt* de senha. Se a senha estiver correta, o programa de *login* obterá a chave de sessão e o *nonce*. Ele verificará o *nonce* e armazenará a chave de sessão com o tiquete para uso subsequente, ao se comunicar com o TGS. Nesse ponto, o programa de *login* pode apagar de sua memória a senha do usuário, pois agora o tiquete serve para autênticá-lo. Uma sessão de *login* é, então, iniciada para o usuário na estação de trabalho. Note que a senha do usuário nunca é exposta à intromissão na rede – ela é mantida na estação de trabalho e é apagada da memória logo depois de ser digitada.

Acesso a servidores com o Kerberos • Quando um programa em execução em uma estação de trabalho precisa acessar um serviço, ele solicita um tiquete para o serviço de concessão de tiquetes (TGS). Por exemplo, quando um usuário UNIX deseja se conectar em um computador remoto, o programa de comando *rlogin* da estação de trabalho do usuário obtém um tiquete do serviço de concessão de tiquetes do Kerberos para acessar o serviço de rede *rlogind*. O programa do comando *rlogin* envia o tiquete, junto a um novo autenticador, em uma requisição para o processo *rlogind* no computador onde o usuário deseja se conectar. O programa *rlogind* decifra o tiquete com a chave secreta do serviço *rlogin* e verifica a validade do tiquete (isto é, se o tempo de vida do tiquete ainda não expirou). As máquinas servidoras devem tomar o cuidado de armazenar suas chaves secretas de modo inacessível para intrusos.

Então, o programa *rlogind* usa a chave de sessão incluída no tiquete para decifrar o autenticador e verificar se ele é novo (os autenticadores só podem ser usados uma vez). Quando o programa *rlogind* tiver certeza de que o tiquete e o autenticador são válidos, não haverá necessidade de verificar o nome e a senha do usuário, pois a identidade do usuário é conhecida do programa *rlogind*, e uma sessão de *login* é estabelecida para esse usuário na máquina remota.

Implementação do Kerberos • O Kerberos é implementado como um servidor executado em uma máquina segura. Um conjunto de bibliotecas é fornecido para ser empregado por aplicativos clientes e serviços. É usado o algoritmo de criptografia DES, mas ele é implementado como um módulo separado que pode ser facilmente substituído.

O serviço Kerberos é flexível – o mundo está dividido em domínios de autoridade de autenticação separados, chamados de *realms*, cada um com seu próprio servidor Kerberos. A maioria dos principais é registrada em apenas um *realm*, mas os servidores de concessão de tíquetes do Kerberos são registrados em todos eles. Os principais podem autenticar a si mesmos em servidores de outros *realms* por meio de seus servidores de concessão de tíquetes locais.

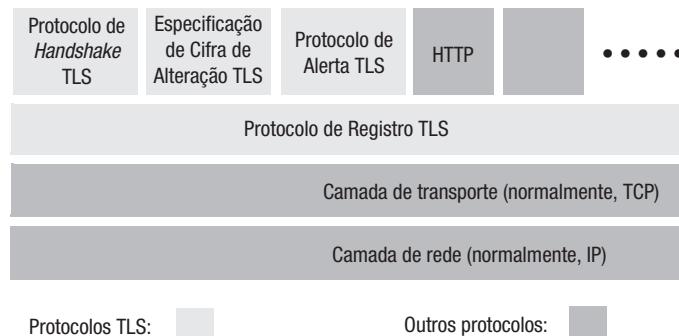
Dentro de um único *realm* podem existir vários servidores de autenticação, todos os quais têm cópias do mesmo banco de dados de autenticação. O banco de dados de autenticação é duplicado por uma técnica simples de mestre-escravo. As atualizações são aplicadas na cópia mestra por um único serviço KDBM (Kerberos Database Management), que é executado apenas na máquina mestra. O KDBM trata das solicitações dos usuários para mudar suas senhas e dos pedidos dos administradores de sistema, para adicionar ou excluir principais e alterar suas senhas.

Para tornar esse esquema transparente para os usuários, o tempo de validade dos tíquetes TGS deve ser igual à sessão de *login* mais longa possível, pois o uso de um tíquete expirado resultará na rejeição de requisições de serviço, e a única solução é o usuário autenticar a sessão de *login* novamente e depois solicitar novos tíquetes de servidor para todos os serviços em uso. Na prática, são usados tempos de validade na faixa de 12 horas.

Críticas ao Kerberos • O protocolo do Kerberos versão 5, descrito anteriormente, contém vários aprimoramentos projetados para resolver as críticas às versões anteriores [Bellovin e Merritt 1990, Burrows *et al.* 1990]. A crítica mais importante à versão 4 era que os *nonces*, utilizados nos autenticadores, eram implementados por carimbos de tempo, e a proteção contra a reprodução de autenticadores depende pelo menos de uma sincronização aproximada dos relógios dos clientes e dos servidores. Além disso, se um protocolo de sincronização for usado para acertar os relógios dos clientes e dos servidores, ele próprio deverá ser seguro contra ataques. Veja o Capítulo 14 para informações sobre os protocolos de sincronização de relógios.

A definição do protocolo da versão 5 permite que os *nonces* usados nos autenticadores sejam implementados como carimbos de tempo ou como números de sequência. Nos dois casos, isso exige que eles sejam exclusivos e, para verificar se eles não foram reproduzidos, que os servidores contenham uma lista dos *nonces* recentemente recebidos de cada cliente. Esse é um requisito de implementação inconveniente e, em caso de falhas, difícil de ser garantido pelos servidores. Kehne *et al.* [1992] publicaram um aprimoramento proposto para o protocolo Kerberos que não conta com relógios sincronizados.

A segurança do Kerberos depende dos tempos de duração das sessões de trabalho – o período de validade dos tíquetes TGS geralmente é limitado a poucas horas; o período deve ser escolhido de forma a ser longo o bastante para evitar interrupções de serviço inconvenientes, mas curto o suficiente para garantir que os usuários que deixaram de ser autenticados, ou que foram recadastrados, não continuem a usar os recursos por mais do que um curto período. Isso pode causar dificuldades em alguns ambientes comerciais, pois pode impor o requisito do usuário fornecer uma nova autenticação em um ponto arbitrário da interação.



(As Figuras 11.16 a 11.19 são baseadas nos diagramas presentes em Hirsch [1977] e foram usadas com a permissão de Frederick Hirsch.)

Figura 11.16 Pilha de protocolos TLS.

11.6.3 Transações eletrônicas seguras com soquetes

O protocolo SSL (Secure Sockets Layer) foi desenvolvido originalmente pela Netscape Corporation [www.mozilla.org] e, posteriormente, proposto como padrão para atender às necessidades descritas a seguir. Uma versão estendida do SSL foi adotada como padrão da Internet com o nome de protocolo TLS (Transport Layer Security), descrito no RFC 2246 [Dierks e Allen 1999]. O TLS é suportado pela maioria dos navegadores e é amplamente usado no comércio da Internet. Suas principais características são:

Criptografia negociável e algoritmos de autenticação • Em uma rede aberta, não devemos supor que todas as partes envolvidas utilizam o mesmo *software* cliente, ou que todo *software* cliente e servidor incluam um algoritmo de criptografia em particular. Na verdade, as leis de alguns países tentam restringir o uso de certos algoritmos de criptografia apenas para esses países. O TLS foi projetado de modo que os algoritmos usados para criptografia e autenticação sejam negociados entre os processos nas duas extremidades da conexão, durante um contato inicial (*handshake*). Pode ser que elas não tenham algoritmos suficientes em comum e, nesse caso, a tentativa de conexão falhará.

Comunicação de partida segura • Para atender à necessidade de comunicação segura, sem negociação prévia nem a ajuda de terceiros, o canal seguro é estabelecido com um protocolo semelhante ao esquema misto descrito anteriormente. É usada uma comunicação não cifrada para as trocas iniciais, depois é usada criptografia de chave pública e, finalmente, troca para criptografia de chave secreta, após uma chave secreta compartilhada ter sido estabelecida. Cada troca é opcional e precedida de uma negociação.

Assim, o canal seguro pode ser totalmente configurado, permitindo que a comunicação em cada direção seja cifrada e autenticada, mas não impondo isso, para que recursos de computação não precisem ser consumidos em execuções desnecessárias de criptografia.

Os detalhes do protocolo TLS estão publicados e padronizados, e várias bibliotecas de *software* e *toolkits* estão disponíveis para suportá-lo [Hirsch 1997, www.openssl.org], alguns deles no domínio público. Ele foi incorporado em uma ampla variedade de *software* aplicativo e sua segurança foi verificada por auditorias independentes.

O TLS consiste em duas camadas (Figura 11.16): uma camada Protocolo de Registro TLS (TLS Record Protocol), que implementa um canal seguro, cifrando e

autenticando as mensagens transmitidas por qualquer protocolo orientado a conexão; e uma camada de *handshake*, contendo o protocolo de *handshake* TLS e dois outros protocolos relacionados, que estabelecem e mantêm uma sessão TLS (isto é, um canal seguro) entre um cliente e um servidor. Ambas são normalmente implementadas por bibliotecas de *software* em nível aplicativo, no cliente e no servidor. O protocolo de registro TLS é uma camada de sessão; e ele pode ser usado para transportar dados da camada de aplicação de forma transparente entre dois processos, enquanto garante sua privacidade, integridade e autenticidade. Essas são exatamente as propriedades que especificamos para canais seguros em nosso modelo de segurança (Seção 2.4.3), mas no TLS existem opções para os participantes da comunicação escolherem se vão ou não implantar decifração e autenticação de mensagens em cada direção. Cada sessão segura recebe um identificador, e cada parceiro pode armazenar identificadores de sessão em uma cache para subsequente reutilização, evitando a sobrecarga do estabelecimento de uma nova sessão, quando for exigida outra sessão segura com o mesmo parceiro.

O protocolo TLS é amplamente usado para adicionar uma camada de comunicação segura abaixo dos protocolos de aplicação existentes. A maior utilização do TLS é em interações HTTP seguras como as de comércio eletrônico e em outras aplicações Web sensíveis a segurança. Ele é implementado por praticamente todos os navegadores e servidores Web – o uso do prefixo de protocolo *https*: nos URLs inicia o estabelecimento de um canal seguro TLS entre um navegador e um servidor Web. Ele também tem sido amplamente implantado para fornecer implementações seguras de Telnet, FTP e muitos outros protocolos de aplicação. O TLS é o padrão *de facto* para uso em aplicações que exigem canais seguros e há uma ampla escolha de implementações disponíveis, tanto comerciais como de domínio público, com APIs para CORBA e Java.

O protocolo de *handshake* TLS está ilustrado na Figura 11.17. O *handshake* é realizado por meio de uma conexão existente. Para estabelecer uma sessão TLS, ele inicia fazendo “em claro” uma troca de mensagens que contêm as opções e os parâmetros necessários para realizar a criptografia e a autenticação. A sequência de *handshake* varia, dependendo se é exigida a autenticação de cliente e/ou do servidor. O protocolo de *handshake* também pode ser ativado posteriormente para alterar a especificação de um canal seguro; por exemplo, a comunicação pode começar com autenticação de mensagem usando apenas códigos de autenticação e, na sequência, se for desejado, pode-se adicionar criptografia. Isso é obtido executando-se novamente o protocolo de *handshake* para negociar uma nova especificação de cifras usando o canal existente.

O *handshake* inicial do TLS é potencialmente vulnerável a ataques de homem no meio, conforme descrito na Seção 11.2.2, Cenário 3. Para proteger-se contra eles, a chave pública usada para verificar o primeiro certificado recebido pode ser distribuída por um canal separado – por exemplo, navegadores e outros *softwares* de Internet distribuídos em CD-ROM podem incluir um conjunto de chaves públicas de algumas autoridades certificadoras reconhecidas. Outra defesa para os clientes de serviços conhecidos é baseada na inclusão do nome de domínio do serviço em seus certificados de chave pública – os clientes só devem contactar o serviço no endereço IP correspondente a esse nome de domínio.

O TLS suporta uma variedade de opções para as funções de criptografia a serem usadas. Coletivamente, elas são conhecidas como *conjunto de cifras*. Um conjunto de cifras inclui uma escolha única para cada um dos recursos mostrados na Figura 11.18.

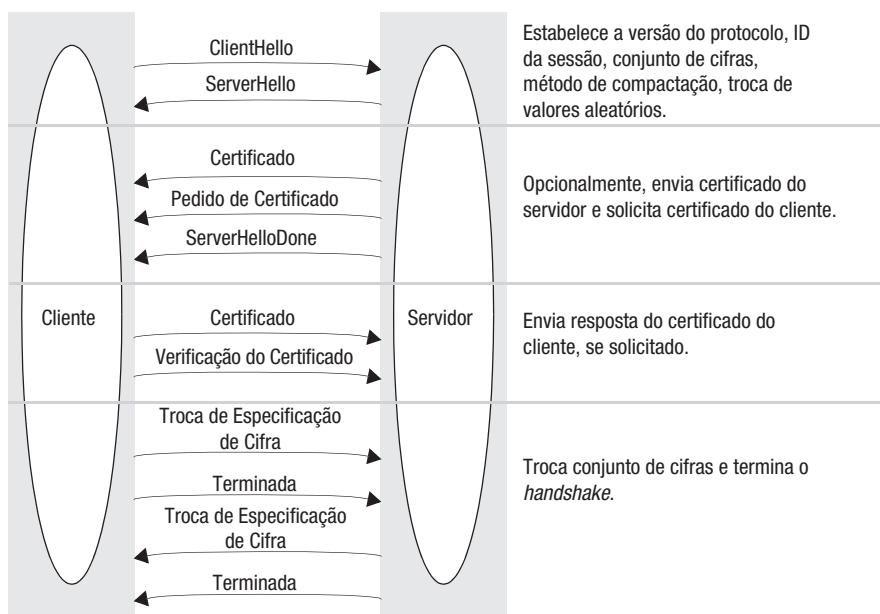


Figura 11.17 Protocolo de *handshake* do TLS.

Uma variedade de conjuntos de cifras populares são previamente armazenadas com identificadores padrão no cliente e no servidor. Durante o *handshake*, o servidor apresenta ao cliente uma lista dos identificadores de conjunto de cifras que tem disponível e o cliente responde selecionando um deles (ou fornecendo uma indicação de erro, caso não tenha nenhum que corresponda). Nesse estágio, eles também concordam com um método de compactação (opcional) e com um valor inicial aleatório para as funções de criptografia de bloco CBC (veja a Seção 11.3).

Em seguida, opcionalmente, os parceiros autenticam-se um ao outro, trocando certificados de chave pública assinados no formato X.509. Esses certificados podem ser obtidos com uma autoridade de chave pública ou podem simplesmente ser gerados temporariamente para isso. De qualquer modo, pelo menos uma chave pública deve estar disponível para uso no próximo estágio do *handshake*.

Um participante gera, então, um *segredo pré-mestre* e o envia para o outro participante, cifrado com a chave pública. Um *segredo pré-mestre* é um valor aleatório

Componente	Descrição	Example
Método de troca de chave	Método a ser usado para a troca de uma chave de sessão	RSA com certificados de chave pública
Cifra para transferência de dados	Bloco ou cifra de fluxo a ser usado para dados	IDEA
Função de resumo (<i>digest</i>) de mensagem	Para criar códigos de autenticação de mensagem (MACs)	SHA-1

Figura 11.18 Opções de configuração de *handshake* do protocolo TLS.

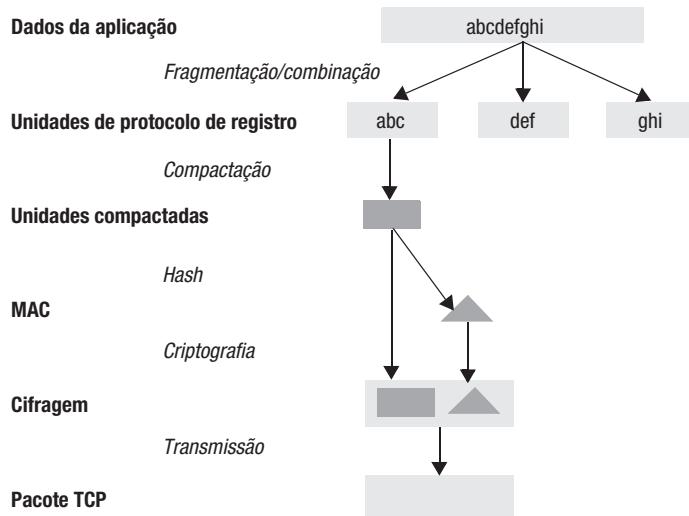


Figura 11.19 Protocolo de registro TLS.

(grande) usado pelos dois participantes para gerar duas chaves de sessão (chamadas de chaves de *escrita*), empregadas para cifrar os dados em cada direção, e os segredos de autenticação de mensagem a serem usados na autenticação da mensagem. Quando tudo isso estiver pronto, uma sessão segura começará. Isso é iniciado pelas mensagens de *Troca de Especificação de Cifra* entre os parceiros. Elas são seguidas de mensagens *Terminada*. Quando as mensagens *Terminada* tiverem sido trocadas, toda comunicação restante será cifrada e assinada de acordo com o conjunto de cifras escolhido, com as chaves combinadas.

A Figura 11.19 mostra o funcionamento do protocolo de registro. Uma mensagem a ser transmitida é primeiramente fragmentada em blocos de tamanho determinado e, em seguida, os blocos são opcionalmente compactados. A compactação não é rigorosamente uma característica da comunicação segura, mas ela é fornecida aqui porque um algoritmo de compactação pode auxiliar o trabalho de processamento do grande volume de dados feito pela criptografia e pelos algoritmos de assinatura digital. Em outras palavras, uma sequência de transformações pode ser feita nos dados para tornar mais eficiente as operações realizadas posteriormente.

A criptografia e as transformações de autenticação de mensagem (MAC) implementam os algoritmos especificados no conjunto de cifras negociado, exatamente como descrito nas Seções 11.3.1 e 11.4.2. Finalmente, o bloco assinado e cifrado é transmitido para o seu interlocutor, por meio da conexão TCP associada, em que as transformações são revertidas para produzir o bloco de dados original.

Resumo • O protocolo TLS fornece uma implementação prática de um esquema de criptografia misto com autenticação e troca de chave baseadas em chaves públicas. Como as cifras são negociadas no *handshake*, elas não dependem da disponibilidade de nenhum algoritmo em particular e também não dependem de quaisquer serviços seguros no momento do estabelecimento da sessão. O único requisito é que os certificados de chave pública sejam emitidos por uma autoridade reconhecida pelas duas partes.

Como a base SSL do TLS e sua implementação de referência foram publicadas [www.mozilla.org], elas foram assunto de muita análise e debate. Algumas emendas foram feitas nos primeiros projetos e ele foi aprovado como um padrão válido. Atualmente, o TLS está integrado na maioria dos navegadores e servidores Web e é usado em outras aplicações, como Telnet e FTP seguros. Implementações comerciais e de domínio público [www.rsasecurity.com I, Hirsch 1997, www.openssl.org] estão disponíveis, na forma de bibliotecas e *plugins* de navegador.

11.6.4 Deficiências no projeto de segurança do IEEE 802.11 WiFi

O padrão IEEE 802.11 para redes locais sem fio, descrito na Seção 3.5.2, foi lançado pela primeira vez em 1999 [IEEE 1999]. Ele foi implementado em pontos de acesso (base *wireless*), *laptops* e equipamentos portáteis e é largamente usado em comunicação móvel. Infelizmente, descobriu-se que o projeto de segurança do padrão era inadequado sob vários aspectos. Vamos esboçar esse projeto inicial e as suas deficiências como um estudo de caso das dificuldades do projeto de segurança já referido na Seção 11.1.3.

Reconheceu-se que as redes sem fio são, por sua natureza, mais expostas a ataques do que as redes cabeadas, pois a rede e os dados transmitidos são sensíveis a intromissão e ao mascaramento por parte de qualquer dispositivo equipado com um transmissor/receptor dentro de um raio de cobertura. Portanto, o projeto 802.11 inicial tinha como objetivo fornecer controle de acesso para redes WiFi, privacidade e integridade dos dados nas transmitidos, por meio de uma especificação de segurança intitulada WEP (Wired Equivalent Privacy), que incorporava as seguintes medidas, todas configuradas opcionalmente por um administrador de rede:

Controle de acesso por um protocolo de desafio-resposta (cf. Kerberos, Seção 11.6.2), no qual um nó que esteja se juntando à rede é questionado pelo ponto de acesso para demonstrar se tem a chave compartilhada correta. Uma única chave *K*, designada por um administrador de rede, é compartilhada entre o ponto de acesso e todos os dispositivos autorizados.

Privacidade e integridade usando um mecanismo de criptografia opcional baseado na cifra de fluxo RC4. A mesma chave *K*, empregada no controle de acesso, também é usada na criptografia. Existem opções de comprimento de chave de 40, 64 ou 128 bits. Uma soma de verificação cifrada é incluída em cada pacote para proteger sua integridade.

As seguintes deficiências e fraquezas de projeto foram descobertas logo depois que o padrão foi distribuído:

O compartilhamento de uma única chave por todos os usuários de uma rede torna o projeto fraco na prática, pois:

- A chave provavelmente será transmitida para novos usuários em canais desprotegidos.
- Um único usuário descuidado, ou mal-intencionado (como um antigo funcionário descontente), que tenha obtido acesso à chave, pode comprometer a segurança da rede inteira, e isso pode passar despercebido.

Solução: um protocolo baseado em chave pública para negociar chaves individuais, como é feito no TLS/SSL (veja a Seção 11.6.3).

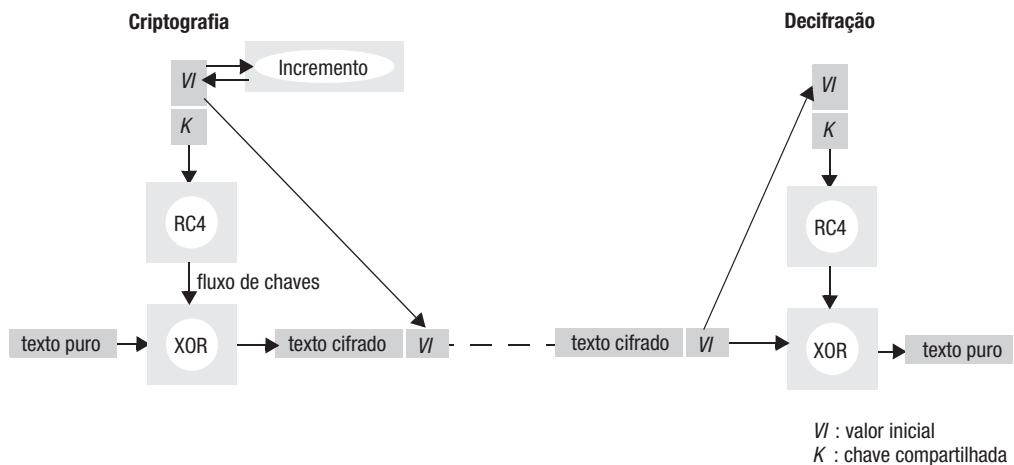


Figura 11.20 Uso de cifra de fluxo RC4 no IEEE 802.11 WEP.

Os pontos de acesso nunca são autenticados; portanto, um invasor que conheça a chave compartilhada corrente poderia introduzir um ponto de acesso de *spoofing* e invadir, inserir ou falsificar qualquer tráfego.

Solução: os pontos de acesso devem fornecer um certificado que possa ser autenticado pelo uso de uma chave pública obtida de terceiros.

Uso inadequado de uma cifra de fluxo, em vez de uma cifra de bloco (veja descrições das cifras de bloco e de fluxo na Seção 11.3). A Figura 11.20 mostra o processo de criptografia e decifração na segurança do 802.11 WEP. Cada pacote é cifrado pela operação XOR de seu conteúdo com um fluxo de chave produzido pelo algoritmo RC4. A estação receptora usa RC4 para gerar o mesmo fluxo de chave e decifrar o pacote com outra operação XOR. Para evitar erros de sincronização quando pacotes são perdidos ou corrompidos, o RC4 é reiniciado com um valor de partida composto por um *valor inicial* de 24 bits concatenado com a chave compartilhada globalmente. O valor inicial é atualizado e incluído (abertamente) em cada pacote transmitido. A chave compartilhada não pode ser alterada facilmente no uso normal; portanto, o valor de partida tem apenas $s = 2^{24}$ (ou cerca de 10^7) estados diferentes, resultando na repetição do valor inicial após o envio de 10^7 pacotes. Na prática, isso pode ocorrer dentro de poucas horas, e até ciclos de repetição mais curtos podem surgir, caso pacotes sejam perdidos. Um invasor que receba os pacotes cifrados sempre pode detectar repetições, pois o valor inicial é enviado abertamente.

A especificação RC4 alerta explicitamente contra a repetição de fluxo de chaves. Isso porque um invasor que receba um pacote cifrado C_i e conheça o texto puro P_i (por exemplo, supondo que se trata de uma pergunta padrão para um servidor), pode calcular o fluxo de chaves K_i usado para cifrar o pacote. No entanto, o mesmo valor de K_i tornará a acontecer depois que s pacotes forem transmitidos, e o invasor poderá usá-lo para decifrar o pacote recentemente transmitido. Dessa maneira, o invasor poderá eventualmente ter êxito na decifração de uma grande proporção dos pacotes, adivinhando os pacotes de texto puro corretamente. Essa deficiência foi apontada pela primeira vez por Borisov *et al.* [2001] e levou a uma reavaliação importante da segurança do WEP e a sua substituição nas versões posteriores do padrão 802.11.

Solução: negociar uma nova chave após um tempo menor do que o pior caso de repetição. Seria necessário um código de finalização explícito, como acontece no TLS.

Comprimentos de chave de 40 e 64 bits foram incluídos no padrão para permitir que os produtos fossem distribuídos no exterior por fornecedores norte-americanos, em uma época em que as normas do governo dos Estados Unidos, referidas na Seção 11.5.2, restringiam a 40 bits (e, subsequentemente, 64 bits) os comprimentos de chave para equipamentos exportados. Contudo, as chaves de 40 bits são tão facilmente violadas por força bruta que oferecem muito pouca segurança, e mesmo chaves de 64 bits são potencialmente violáveis com um ataque determinado.

Solução: usar somente chaves de 128 bits. Isso foi adotado em muitos produtos WiFi recentes.

A *cifra de fluxo RC4* revelou, após a publicação do padrão 802.11, ter deficiências que permitiam o descobrimento da chave após observação de um volume substancial do tráfego, mesmo sem repetição do fluxo de chaves [Fluhrer *et al.* 2001]. Essa deficiência foi demonstrada na prática. Ela tornou o esquema WEP inseguro, mesmo com chaves de 128 bits, e levou algumas empresas a proibir o uso de redes WiFi por seus funcionários.

Solução: preparativos para a negociação de especificações de cifra, como foi feito no TLS, fornecendo uma escolha de algoritmos de criptografia. O RC4 é incorporado no padrão WEP, sem nenhum preparativo para a negociação de algoritmos de criptografia.

Frequentemente, os usuários não implantavam a proteção oferecida pelo esquema WEP, provavelmente porque não percebiam claramente como seus dados estavam expostos. Essa não era uma deficiência no projeto do padrão, mas na comercialização dos produtos nele baseados. A maioria era projetada para começar com a segurança desativada e, frequentemente, a documentação sobre os riscos para a segurança era deficiente.

Solução: melhores configurações padrão e documentação. Porém, os usuários estavam preocupados em obter desempenho ótimo e, com o *hardware* disponível na época, a comunicação era perceptivelmente mais lenta com a criptografia ativada. Tentativas de evitar o uso de criptografia WEP levaram ao acréscimo de recursos nos pontos de acesso para a supressão dos pacotes de identificação normalmente divulgados por elas e à rejeição de pacotes não enviados a partir de um endereço MAC autorizado (veja a Seção 3.5.1). Nenhum deles oferecia muita segurança, pois uma rede pode ser descoberta pela interceptação de pacotes (*sniffing*) na transmissão, e os endereços MAC estão sujeitos a *spoofing* por meio de modificações no sistema operacional.

O IEEE montou uma força-tarefa para criar uma nova solução de segurança, e seu trabalho resultou em um protocolo de segurança completamente novo, conhecido como Wi-Fi Protected Access (WPA). Ele foi especificado no IEEE 802.11i [IEEE 2004b, Edney e Arbaugh, 2003] e começou a aparecer em produtos em 2003. O IEEE 802.11i (também conhecido como WPA2) foi ratificado em junho de 2004 e utiliza critografia AES, em vez de RC4, que foi usada em WEP. Os aprimoramentos subsequentes de IEEE 802.11 também incorporaram WPA2.

11.7 Resumo

As ameaças à segurança dos sistemas distribuídos são generalizadas. É fundamental proteger os canais de comunicação e as interfaces de qualquer sistema que manipule informações que possam ser objeto de ataques. Correspondência pessoal, comércio eletrônico e outras transações financeiras, todos são exemplos de tais informações. Os protocolos de segurança são cuidadosamente projetados para evitar brechas. O projeto de sistemas seguros começa a partir de uma lista de ameaças e de um conjunto de suposições de “pior caso”.

Os mecanismos de segurança são baseados em criptografia de chave pública e de chave secreta. Os algoritmos de criptografia misturam as mensagens de maneira que não possam ser revertidas sem o conhecimento da chave de decifração. A criptografia de chave secreta é simétrica – a mesma chave serve tanto para cifrar como para decifrar. Se duas pessoas compartilham uma chave secreta, elas podem trocar informações cifradas sem risco de intromissão, ou falsificação, e com garantia de autenticidade.

A criptografia de chave pública é assimétrica – chaves distintas são usadas para cifrar e decifrar, e o conhecimento de uma não revela a outra. Uma chave é publicada, permitindo que qualquer um envie mensagens seguras para o portador da chave privada correspondente. Também permite que o portador da chave privada assine mensagens e certificados. Os certificados podem atuar como credenciais para a utilização de recursos protegidos.

Os recursos são protegidos por mecanismos de controle de acesso. Os esquemas de controle de acesso atribuem direitos aos principais (isto é, aos portadores de credenciais) para efetuarem operações nos objetos distribuídos e em coleções de objetos. Os direitos podem ser mantidos em listas de controle de acesso (ACLs) associadas a conjuntos de objetos ou podem ser mantidos pelos principais, na forma de capacidades – chaves impossíveis de falsificar, para acessar conjuntos de recursos. As capacidades são convenientes para a delegação de direitos de acesso, mas são difíceis de revogar. As alterações nas ACLs surtem efeito imediatamente, revogando os direitos de acesso anteriores, mas elas são mais complexas e dispendiosas de gerenciar do que as capacidades.

Até recentemente, o algoritmo de criptografia DES era o esquema de criptografia simétrica mais difundido, mas suas chaves de 56 bits não são mais seguras contra ataques de força bruta. A versão *triple DES* fornece o poder de chaves de 112 bits, que são seguras, mas outros algoritmos modernos, como IDEA e AES, são muito mais rápidos e poderosos.

O RSA é o esquema de criptografia assimétrica mais usado. Para obter segurança contra ataques de decomposição em fatores, ele deve ser usado com chaves de 768 bits ou mais. A execução dos algoritmos de chave pública (assimétricos) é superada pela dos algoritmos de chave secreta (simétricos) em várias ordens de grandeza; portanto, eles são geralmente usados apenas em protocolos mistos, como o TLS, para o estabelecimento de canais seguros que utilizam chaves compartilhadas para trocas subsequentes.

O protocolo de autenticação Needham–Schroeder foi o primeiro protocolo de segurança prático de propósito geral e ainda serve de base para muitos sistemas. O Kerberos é um esquema bem projetado para a autenticação de usuários e proteção de serviços dentro de uma única organização. O Kerberos é baseado no protocolo Needham–Schroeder e em criptografia simétrica. O TLS é o protocolo de segurança projetado para comércio eletrônico e é amplamente usado para tal. Trata-se de um protocolo flexível para o estabelecimento e uso de canais seguros, baseado em criptografia simétrica e assimétrica. As deficiências da segurança do IEEE 802.11 WiFi proporcionam uma demonstração prática das dificuldades do projeto de segurança.

Exercícios

- 11.1 Descreva algumas das políticas de segurança física de sua organização. Expressse-as na forma como poderiam ser implementadas em um sistema de travamento de porta computadorizado. *página 464*
- 11.2 Descreva algumas das maneiras pelas quais o *e-mail* convencional é vulnerável a intromissão, mascaramento, falsificação, reprodução e negação de serviço. Sugira métodos pelos quais o *e-mail* poderia ser protegido contra cada uma dessas formas de ataque. *página 466*
- 11.3 As trocas iniciais de chaves públicas são vulneráveis ao ataque do homem no meio. Descreva quantas defesas contra ele você puder imaginar. *páginas 473, 511*
- 11.4 O PGP é frequentemente usado para comunicação segura por *e-mail*. Descreva os passos que dois usuários usando PGP devem efetuar, antes que possam trocar mensagens por *e-mail* com garantias de privacidade e autenticidade. Qual escopo existe para tornar a negociação de chaves preliminar invisível para os usuários? (A negociação PGP é um caso do esquema misto.) *páginas 493, 502*
- 11.5 Como o *e-mail* poderia ser enviado para uma lista grande de destinatários, usando PGP ou um esquema similar? Sugira um esquema que seja mais simples e mais rápido, quando a lista for usada frequentemente. *página 502, Seção 4.4*
- 11.6 A implementação do algoritmo de criptografia simétrica TEA, apresentado nas Figuras 11.7 a 11.9, não pode ser portada entre todas as arquiteturas de máquina. Explique por quê. Como uma mensagem cifrada usando a implementação TEA poderia ser transmitida de forma a ser decifrada corretamente em todas as outras arquiteturas? *página 488*
- 11.7 Modifique o programa aplicativo TEA da Figura 11.9 para usar encadeamento de blocos de cifra (CBC). *páginas 485, 488*
- 11.8 Construa um aplicativo de cifra de fluxo baseado no programa da Figura 11.9. *páginas 486, 488*
- 11.9 Faça uma estimativa do tempo exigido para violar uma chave DES de 56 bits por um ataque de força bruta, usando um computador de 2.000 MIPS (milhões de instruções por segundo), supondo que o laço interno de um programa de ataque de força bruta envolva cerca de 10 instruções por valor de chave, mais o tempo para cifrar um texto puro de 8 bytes (veja a Figura 11.13). Efetue o mesmo cálculo para uma chave IDEA de 128 bits. Extrapole seus cálculos para obter o tempo de violação para um processador paralelo de 200.000 MIPS (ou um consórcio na Internet com poder de processamento similar). *página 489*
- 11.10 No protocolo de autenticação de Needham e Shroeder com chaves secretas, explique por que a seguinte versão da mensagem 5 não é segura:
- $$A \rightarrow B: \quad \{N_B\}_{K_{AB}}$$
- página 504*
- 11.11 Examine as soluções propostas na discussão sobre o projeto do protocolo 802.11 *Wireless Equivalent Privacy*, esboçando maneiras pelas quais cada solução poderia ser implementada e mencionando as desvantagens ou os inconvenientes. (5 respostas) *página 515*

12

Sistemas de Arquivos Distribuídos

- 12.1 Introdução
- 12.2 Arquitetura do serviço de arquivos
- 12.3 Estudo de caso: Sun Network File System
- 12.4 Estudo de caso: Andrew File System
- 12.5 Aprimoramentos e outros desenvolvimentos
- 12.6 Resumo

Um sistema de arquivos distribuído permite aos programas armazenarem e acessarem arquivos remotos exatamente como se fossem locais, possibilitando que os usuários acessem arquivos a partir de qualquer computador em uma rede. O desempenho e a segurança no acesso aos arquivos armazenados em um servidor devem ser comparáveis aos arquivos armazenados em discos locais.

Neste capítulo, definiremos uma arquitetura simples para sistemas de arquivos e descreveremos duas implementações básicas de serviços de arquivos distribuídos com projetos contrastantes que estão em pleno uso há duas décadas:

- o Network File System, NFS, da Sun;
- o Andrew File System, AFS.

Cada um deles simula a interface de sistema de arquivos do UNIX com diferentes graus de escalabilidade, tolerância a falhas e variações da restrita semântica de cópia única (*one-copy*) do UNIX.

Também serão examinados outros sistemas de arquivos que exploram novos modos de organização de dados no disco, ou entre vários servidores, para obter sistemas de arquivos de alto desempenho, tolerantes a falhas e com escalabilidade. Diferentes tipos de sistemas de armazenamento serão descritos em outras partes do livro. Isso inclui os sistemas de armazenamento *peer-to-peer* (Capítulo 10), os sistemas de arquivos replicados (Capítulo 18), os servidores de dados multimídia (Capítulo 20) e o estilo de serviço de armazenamento específico exigido para suportar pesquisas de busca na Internet e outras aplicações de grande escala com uso intensivo de dados (Capítulo 21).

12.1 Introdução

Nos Capítulos 1 e 2, identificamos o compartilhamento de recursos como um objetivo chave dos sistemas distribuídos. O compartilhamento de informações armazenadas talvez seja o aspecto mais importante dos recursos distribuídos. Os mecanismos para compartilhamento de dados assumem muitas formas e serão descritos em várias partes deste livro. Os servidores Web fornecem uma forma restrita de compartilhamento de dados, na qual os arquivos armazenados de forma local no servidor estão disponíveis para clientes em toda a Internet, mas os dados acessados por meio de servidores Web são gerenciados e atualizados em sistemas de arquivos no servidor, ou distribuídos em uma rede local. O projeto de sistemas de armazenamento de arquivos remotos para leitura e escrita em larga escala apresenta problemas de balanceamento de carga, confiabilidade, disponibilidade e segurança, cuja solução é o objetivo dos sistemas de armazenamento de arquivos *peer-to-peer*, descritos no Capítulo 10. O Capítulo 18 aborda os sistemas de armazenamento replicados, que são convenientes para aplicações que exigem acesso confiável a dados armazenados em sistemas nos quais a disponibilidade de computadores individuais não pode ser garantida. No Capítulo 20, descreveremos um servidor de mídia projetado para servir fluxos de dados de vídeo em tempo real para grandes números de usuários. O Capítulo 21 também descreve um sistema de arquivos para suportar aplicações de grande escala com uso intensivo de dados, como a pesquisa de busca na Internet.

Os requisitos de compartilhamento dentro de redes locais e intranets levam à necessidade de um tipo de serviço diferente – que suporte o armazenamento persistente dos dados e programas de todos os tipos e a distribuição consistente de dados atualizados. O objetivo deste capítulo é descrever a arquitetura e a implementação desses sistemas de arquivos distribuídos *básicos*. Usamos a palavra “básico” para denotar os sistemas de arquivos distribuídos cujo principal objetivo é simular a funcionalidade de um sistema de arquivos não distribuído para programas clientes executados em computadores remotos. Eles não mantêm várias réplicas persistentes dos arquivos, nem suportam as garantias de largura de banda e temporização exigidas para fluxos de dados multimídia – esses requisitos serão tratados em capítulos posteriores. Os sistemas de arquivos distribuídos básicos fornecem um apoio fundamental para computação organizacional baseada em intranets.

Os sistemas de arquivos foram originalmente desenvolvidos para sistemas de computadores centralizados e computadores *desktop* como um recurso do sistema operacional que fornece uma interface de programação conveniente para armazenamento em disco. Subsequentemente, eles adquiriram características como controle de acesso e mecanismos de proteção de arquivos, que os tornaram úteis para o compartilhamento de dados e programas. Os sistemas de arquivos distribuídos suportam o compartilhamento de informações por meio de arquivos e recursos de *hardware* com armazenamento persistente em toda uma intranet. Um serviço de arquivo bem projetado dá acesso a arquivos armazenados em um servidor com desempenho e confiabilidade semelhantes a (e, em alguns casos, melhores que) arquivos armazenados em discos locais. Seu projeto é adaptado para as características de desempenho e confiabilidade das redes locais e, portanto, eles são eficazes no fornecimento de armazenamento persistente compartilhado para uso em intranets. Os primeiros servidores de arquivos foram desenvolvidos por pesquisadores nos anos 70 [Birrell e Neudham 1980, Mitchell e Dion 1982, Leach *et al.* 1983], e o Sun Network File System foi disponibilizado no início dos anos 80 [Sandberg *et al.* 1985, Callaghan 1999].

Um serviço de arquivos permite que os programas armazenem e acessem arquivos remotos exatamente como se fossem locais, possibilitando que os usuários acessem seus arquivos a partir de qualquer computador em uma intranet. A concentração do armazena-

	Compartilhamento	Persistência	Cache/réplicas distribuídas	Manutenção da consistência	Exemplo
Memória principal	✗	✗	✗	1	RAM
Sistema de arquivos	✗	✓	✗	1	Sistema de arquivos UNIX
Sistema de arquivos distribuído	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	✗	Servidor web
Memória compartilhada distribuída	✓	✗	✓	✓	Ivy (DSM, Cap. 6)
Objetos remotos (RMI/ORB)	✓	✗	✗	1	CORBA
Armazenamento de objetos persistentes	✓	✓	✗	1	Persistent State Service do CORBA
Sistema de armazenamento peer-to-peer	✓	✓	✓	2	OceanStore (Cap. 10)

Tipos de consistência:

1 : Uma cópia restrita. ✓ : Garantias ligeiramente mais fracas. 2 : Garantias consideravelmente mais fracas.

Figura 12.1 Sistemas de armazenamento e suas propriedades.

mento persistente em alguns poucos servidores reduz a necessidade de armazenamento em disco local e (o mais importante) permite que sejam feitas economias no gerenciamento e no arquivamento (*backups*) dos dados persistentes pertencentes a uma organização. Outros serviços, como o de nomes, o de autenticação de usuário e o de impressão, podem ser mais facilmente implementados quando são capazes de usar o serviço de arquivos para atender a suas necessidades de armazenamento persistente. Os servidores Web confiam no sistema de arquivos para o armazenamento das páginas que servem. Nas organizações que operam servidores Web para acesso externo e interno por meio de uma intranet, esses servidores frequentemente armazenam e acessam o material a partir de um sistema de arquivos distribuído local.

Com o advento da programação orientada a objetos distribuída, surgiu a necessidade do armazenamento persistente e da distribuição de objetos compartilhados. Uma maneira de obter isso é dispor de objetos serializados (da maneira descrita na Seção 4.3.2) e armazenar e recuperar esses objetos usando arquivos. Contudo, esse método de obtenção de persistência e distribuição se torna impraticável para objetos que mudam rapidamente e, portanto, várias estratégias mais diretas foram desenvolvidas. A invocação a objeto remoto da linguagem Java e os ORBs do CORBA fornecem acesso a objetos remotos compartilhados, mas nenhum deles garante a persistência dos objetos, nem a replicação dos objetos distribuídos.

A Figura 12.1 apresenta um panorama dos tipos de sistema de armazenamento. Além daqueles já mencionados, a tabela inclui sistemas de memória compartilhada distribuída (DSM, Distributed Shared Memory) e armazenamento de objetos persistentes. A DSM foi descrita em detalhes no Capítulo 6. Ela fornece uma simulação de uma memória compartilhada por meio da replicação de páginas ou de segmentos de memória em cada computador (nó). Ela não fornece necessariamente persistência automática. O armazenamento de objetos persistentes foi apresentado no Capítulo 5 e seu objetivo é fornecer persistência para objetos compartilhados distribuídos. Exemplos incluem o Persistent State Service CORBA (veja o Capítulo 8) e extensões persistentes para Java [Jordan 1996, java.sun.com VIII]. Alguns projetos de pesquisa foram desenvolvidos em plataformas que suportam a replicação automática e o armazenamento persistente de objetos (por exemplo, PerDiS [Ferreira

Módulo de diretório:	relaciona nomes de arquivo com IDs de arquivo
Módulo de arquivo:	relaciona IDs de arquivo com arquivos em particular
Módulo de controle de acesso:	verifica a permissão para a operação solicitada
Módulo de acesso a arquivo:	lê ou escreve dados ou atributos
Módulo de bloco:	acessa e aloca blocos de disco
Módulo de dispositivo:	E/S de disco e uso de <i>buffers</i>

Figura 12.2 Módulos do sistema de arquivos.

et al. 2000] e Khazana [Carter *et al.* 1998]). Os sistemas de armazenamento *peer-to-peer* oferecem escalabilidade para suportar cargas de cliente muito maiores do que os sistemas descritos neste capítulo, mas acarretam altos custos de desempenho para fornecer controle de acesso seguro e consistência entre as réplicas que podem ser atualizadas.

A coluna *consistência* indica se existem mecanismos para a manutenção da consistência entre várias cópias dos dados quando ocorrem atualizações. Praticamente todos os sistemas de armazenamento contam com o uso de cache para otimizar o desempenho dos programas. O uso de cache foi aplicado pela primeira vez na memória principal e em sistemas de arquivos não distribuídos, e para eles, a consistência é estrita (denotada por 1, significando consistência de cópia única, na Figura 12.1) – após uma atualização, os programas não conseguem observar quaisquer discrepâncias entre as cópias na cache e os dados armazenados. Quando são usadas réplicas distribuídas, é mais difícil obter a consistência estrita. Os sistemas de arquivos distribuídos, como o Sun NFS e o Andrew File System, colocam em cache, nos computadores clientes, cópias parciais de arquivos e adotam mecanismos de consistência específicos para manter uma aproximação da consistência estrita. Isso está indicado por um tique (✓) na coluna da consistência na Figura 12.1 – discutiremos esses mecanismos, e até que ponto eles se desviam da consistência estrita, nas Seções 12.3 e 12.4.

A Web usa cache extensivamente, tanto em computadores clientes como em servidores *proxies* mantidos por organizações de usuários. A consistência entre o original e as cópias armazenadas em servidores *proxies* Web e em caches de clientes é mantida apenas por ações explícitas do usuário. Os clientes não são notificados quando uma página original, armazenada em um servidor, é atualizada; eles precisam fazer verificações explícitas para manter suas cópias locais atualizadas. Isso tem como objetivo a navegação adequada na Web, mas não suporta o desenvolvimento de aplicativos cooperativos, como o do quadro branco compartilhado distribuído. Os mecanismos de consistência usados nos sistemas DSM são discutidos em detalhes no *site* que acompanha o livro [www.cdk5.net] (em inglês). Os sistemas de objetos persistentes variam consideravelmente em sua estratégia para uso de cache e consistência. Os esquemas CORBA e Persistent Java mantêm cópias únicas de objetos persistentes e é necessário usar invocação remota para acessá-los; portanto, o único problema de consistência se dá entre a cópia persistente de um objeto no disco e a cópia ativa na memória, que não é visível para clientes remotos. Os projetos PerDiS e Khazana, mencionados anteriormente, mantêm réplicas dos objetos em cache e empregam mecanismos de consistência bastante elaborados para reproduzir àquelas encontradas nos sistemas DSM.

Apresentamos aqui alguns problemas mais amplos relacionados ao armazenamento e à distribuição de dados persistentes e não persistentes, agora vamos voltar ao assunto principal deste capítulo – o projeto de sistemas de arquivos distribuídos básicos. Descreveremos algumas características relevantes dos sistemas de arquivos (não distribuídos), na Seção 12.1.1, e os requisitos dos sistemas de arquivos distribuídos, na Seção 12.1.2.

Tamanho do arquivo
Horário de criação
Horário de acesso (leitura)
Horário de modificação (escrita)
Horário de alteração de atributo
Contagem de referência
Proprietário
Tipo de arquivo
Lista de controle de acesso

Figura 12.3 Estrutura básica de um registro de atributo de arquivo.

A Seção 12.1.3 apresentará os estudos de caso que serão utilizados por todo o capítulo. Na Seção 12.2, definiremos um modelo abstrato de um sistema de arquivos distribuído básico, incluindo um conjunto de interfaces de programação. O sistema Sun NFS será descrito na Seção 12.3; ele compartilha muitas das características do modelo abstrato. Na Seção 12.4, descreveremos o Andrew File System – um sistema amplamente usado, que emprega mecanismos de cache e de consistência substancialmente diferentes. A Seção 12.5 examina alguns desenvolvimentos recentes no projeto de serviços de arquivo.

Os sistemas descritos neste capítulo não cobrem todo o espectro dos sistemas de gerenciamento de arquivos e dados distribuídos. Vários sistemas, com características mais avançadas, serão descritos posteriormente neste livro. O Capítulo 18 contém uma descrição do Coda, um sistema de arquivos distribuído que mantém réplicas persistentes dos arquivos para obter confiabilidade, disponibilidade e trabalho em modo não conectado. O Bayou, um sistema de gerenciamento de dados distribuído que fornece uma forma de replicação com consistência fraca para obter alta disponibilidade, também será abordado no Capítulo 18. O Capítulo 20 abordará o servidor de arquivos de vídeo Tiger, projetado para fornecer a distribuição de fluxos de dados para grandes números de clientes. O Capítulo 21 descreve o GFS (Google File System), um sistema de arquivos projetado especificamente para suportar aplicações de grande escala com uso intensivo de dados, incluindo a busca na Internet.

12.1.1 Características dos sistemas de arquivos

Os sistemas de arquivos são responsáveis pela organização, armazenamento, recuperação, atribuição de nomes, compartilhamento e proteção de arquivos. Eles fornecem uma interface de programação que caracteriza a abstração de arquivo, liberando os programadores da preocupação com os detalhes da alocação e do leiaute do armazenamento físico no disco. Os arquivos são armazenados em discos ou em outra mídia de armazenamento não volátil.

Os arquivos contêm *dados* e *atributos*. Os dados consistem em uma sequência de elementos (normalmente, bytes – 8 bits), acessíveis pelas operações de leitura e escrita de qualquer parte dessa sequência. Os atributos são mantidos como um único registro, contendo informações como o tamanho do arquivo, carimbos de tempo, tipo de arquivo, identidade do proprietário e listas de controle de acesso. Uma estrutura de registro de atributo típica está ilustrada na Figura 12.3. Os atributos escondidos (*shadow attributes*) são gerenciados pelo sistema de arquivos e, normalmente, não podem ser atualizados por programas de usuário.

<code>filedes = open(nome, modo)</code>	Abre um arquivo existente com o <i>nome</i> fornecido.
<code>filedes = creat(nome, modo)</code>	Cria um novo arquivo com o <i>nome</i> fornecido.
	As duas operações produzem um descritor de arquivo referenciando o arquivo aberto. O <i>modo</i> é <i>read</i> , <i>write</i> ou ambos.
<code>status = close(filedes)</code>	Fechá o arquivo aberto referenciado por <i>filedes</i> .
<code>count = read(filedes, buffer, n)</code>	Transfere para <i>buffer</i> <i>n</i> bytes do arquivo referenciado por <i>filedes</i> .
<code>count = write(filedes, buffer, n)</code>	Transfere <i>n</i> bytes de <i>buffer</i> para o arquivo referenciado por <i>filedes</i> . As duas operações produzem o número de bytes realmente transferidos e avançam o ponteiro de leitura e escrita.
<code>pos = lseek(filedes, offset, whence)</code>	Desloca o ponteiro de leitura e escrita para <i>offset</i> (relativo ou absoluto, dependendo de <i>whence</i>).
<code>status = unlink(nome)</code>	Remove o arquivo <i>nome</i> da estrutura de diretório. Se o arquivo não tiver outras referências, ele será excluído.
<code>status = link(nome1, nome2)</code>	Cria uma nova referência, ou nome (<i>nome2</i>), para um arquivo (<i>nome1</i>).
<code>status = stat(nome, buffer)</code>	Escreve os atributos do arquivo <i>nome</i> em <i>buffer</i> .

Figura 12.4 Operações do sistema de arquivos UNIX.

Os sistemas de arquivos são projetados para armazenar e gerenciar um grande número de arquivos, com recursos para criação, atribuição de nomes e exclusão de arquivos. A atribuição de nomes de arquivos é suportada pelo uso de diretórios. Um *diretório* é um arquivo, frequentemente de um tipo especial, que fornece um mapeamento dos nomes textuais para identificadores internos de arquivo. Os diretórios podem incluir nomes de outros diretórios, levando ao conhecido esquema de atribuição hierárquica de nomes e aos *nomes de caminho* (*pathname*), usados no sistema de arquivo UNIX e em outros sistemas operacionais. Os sistemas de arquivos também assumem a responsabilidade pelo controle do acesso aos arquivos, restringindo o acesso de acordo com as autorizações dos usuários e com o tipo de acesso solicitado (leitura, atualização, execução, etc.).

O termo *metadados* é usado frequentemente para se referir a todas as informações extras armazenadas por um sistema de arquivos que são necessárias para o gerenciamento dos arquivos. Isso inclui atributos de arquivo, diretórios e todas as outras informações persistentes utilizadas pelo sistema de arquivos.

A Figura 12.2 mostra uma estrutura modular em camadas, típica, para a implementação de um sistema de arquivos não distribuído em um sistema operacional convencional. Cada camada depende apenas das camadas que estão abaixo dela. A implementação de um serviço de arquivo distribuído precisa de todos os componentes mostrados, com componentes adicionais para tratar da comunicação cliente-servidor e da atribuição de nomes e da localização de arquivos distribuídos.

Operações do sistema de arquivos • A Figura 12.4 resume as principais operações sobre arquivos que estão disponíveis nos sistemas UNIX para aplicativos. Essas são as chamadas de sistema implementadas pelo núcleo; normalmente, os programadores de aplicativos as acessam por meio de funções de biblioteca, como a biblioteca de entrada e saída padrão da linguagem C, ou as classes de arquivo Java. Fornecemos as primitivas como uma indicação das operações que os serviços de arquivo devem suportar e para comparação com as interfaces do serviço de arquivos que vamos apresentar a seguir.

As operações UNIX são baseadas em um modelo de programação no qual algumas informações sobre o estado do arquivo são armazenadas pelo sistema de arquivos para cada programa em execução.

Elas consistem, entre outros, em uma lista de arquivos correntemente abertos e um ponteiro de leitura e escrita que fornece a posição dentro do arquivo na qual será aplicada a próxima operação de leitura ou de escrita.

O sistema de arquivos é responsável por realizar o controle de acesso para os arquivos. Em sistemas de arquivos locais, como o UNIX, ele faz isso quando cada arquivo é aberto, verificando os direitos permitidos à identidade do usuário na lista de controle de acesso em relação ao *modo* de acesso solicitado na chamada de sistema *open*. Se os direitos corresponderem ao modo, o arquivo será aberto, e o *modo* é armazenado nas informações de estado de arquivo aberto.

12.1.2 Requisitos do sistema de arquivos distribuído

Muitos dos requisitos e das armadilhas em potencial no projeto de serviços distribuídos foram detectados pela primeira vez no desenvolvimento inicial dos sistemas de arquivos distribuídos. Inicialmente, eles ofereciam transparência de acesso e de localização; surgiram requisitos de desempenho, escalabilidade, controle de concorrência, tolerância a falhas e segurança, e eles foram atendidos em fases subsequentes de desenvolvimento. Vamos discutir esses e outros requisitos nas subseções a seguir.

Transparência • O serviço de arquivo normalmente é o mais usado em uma intranet; portanto, sua funcionalidade e seu desempenho são críticos. O projeto do serviço de arquivos deve suportar muitos dos requisitos de transparência dos sistemas distribuídos, identificados na Seção 1.5.7. O projeto deve contrabalançar a flexibilidade e a escalabilidade derivadas da transparência, com a complexidade e o desempenho do *software*. As seguintes formas de transparência são parcialmente, ou totalmente, tratadas pelos serviços de arquivos atuais:

Transparência do acesso: os programas clientes não devem conhecer a distribuição de arquivos. Um único conjunto de operações é fornecido para acesso a arquivos locais e remotos. Os programas escritos para operar sobre arquivos locais são capazes de acessar arquivos remotos sem modificação.

Transparência de localização: os programas clientes devem ver um espaço de nomes de arquivos uniforme. Os arquivos, ou grupos de arquivos, podem ser deslocados de um servidor a outro sem alteração de seus nomes de caminho, e os programas de usuário devem ver o mesmo espaço de nomes onde quer que sejam executados.

Transparência de mobilidade: nem os programas clientes, nem as tabelas de administração de sistema nos computadores clientes precisam ser alterados quando os arquivos são movidos. Isso permite a mobilidade do arquivo – arquivos ou, mais comumente, conjuntos ou volumes de arquivos podem ser movidos, ou pelos administradores de sistema ou automaticamente.

Transparência de desempenho: os programas clientes devem continuar a funcionar satisfatoriamente, enquanto a carga sobre o serviço varia dentro de um intervalo especificado.

Transparência de mudança de escala: o serviço pode ser expandido de forma paulatina, para lidar com uma ampla variedade de cargas e tamanhos de rede.

Atualizações concorrentes de arquivos • As alterações feitas em um arquivo por um único cliente não devem interferir na operação de outros clientes que estejam acessando, ou

alterando, o mesmo arquivo simultaneamente. Esse é o conhecido problema do controle de concorrência, discutido em detalhes no Capítulo 16. Em muitos aplicativos, a necessidade de controle de concorrência para acesso a dados compartilhados é amplamente aceita, e são conhecidas técnicas para sua implementação, mas elas são dispendiosas. A maior parte dos serviços de arquivo atuais segue os padrões UNIX modernos, fornecendo travamento (*locking*) em nível de arquivo ou em nível de registro.

Replicação de arquivos • Em um serviço de arquivos que suporta replicação, um arquivo pode ser representado por várias cópias de seu conteúdo em diferentes locais. Isso tem duas vantagens – permite que vários servidores compartilhem a carga do fornecimento de um serviço para clientes que acessam o mesmo conjunto de arquivos, melhorando a escalabilidade do serviço, e melhora a tolerância a falhas, permitindo que, em caso de falhas, os clientes localizem outro servidor que contenha uma cópia do arquivo. Poucos serviços de arquivo suportam replicação completa, mas a maioria suporta o armazenamento de arquivos, ou de porções de arquivos, em caches locais, que é uma forma limitada de replicação. A replicação de dados será discutida no Capítulo 18, que inclui uma descrição do serviço de arquivos replicado Coda.

Heterogeneidade do hardware e do sistema operacional • As interfaces de serviço devem ser definidas de modo que o *software* cliente e servidor possa ser implementado para diferentes sistemas operacionais e computadores. Esse requisito é um aspecto importante de sistemas abertos.

Tolerância a falhas • Por ser parte essencial nos sistemas distribuídos, é essencial que o serviço de arquivo distribuído continue a funcionar diante de falhas de clientes e servidores. Felizmente, um projeto moderadamente tolerante a falhas é fácil para servidores simples. Para suportar falhas de comunicação transientes, o projeto pode ser baseado na semântica de invocação *no máximo uma vez* (veja a Seção 5.3.1). Ou, então, ele pode usar uma semântica mais simples, como *pelo menos uma vez*, com um protocolo de servidor projetado em termos de operações *idempotentes*, garantindo que pedidos duplicados não resultem em atualizações inválidas nos arquivos. Os servidores podem ser *sem estado* (*stateless*), para que após uma falha o serviço possa ser reiniciado e restaurado sem necessidade de recuperar o estado anterior. A tolerância a falha de desconexão, ou de servidor, exige replicação de arquivo, a qual é mais difícil de obter (e será discutida no Capítulo 18).

Consistência • Os sistemas de arquivos convencionais, como o fornecido no UNIX, oferecem *semântica de atualização de cópia única* (*one-copy*). Isso se refere a um modelo de acesso concorrente a arquivos, no qual o conteúdo do arquivo visto por todos os processos que estão acessando, ou atualizando determinado arquivo, é aquele que eles veriam se existisse apenas uma cópia do conteúdo do arquivo. Quando os arquivos são replicados, ou armazenados em cache, em diferentes *sites*, há um atraso inevitável na propagação das modificações feitas em um *site* para os outros *sites* que contêm cópias, e isso pode resultar em certo desvio da semântica de cópia única.

Segurança • Praticamente todos os sistemas de arquivos fornecem mecanismos de controle de acesso baseados no uso de listas de controle de acesso. Nos sistemas de arquivos distribuídos, há necessidade de autenticar as requisições dos clientes para que o controle de acesso no servidor seja baseado nas identidades corretas de usuário e para proteger o conteúdo das mensagens de requisição-resposta com assinaturas digitais e (opcionalmente) criptografia de dados secretos. Vamos discutir o impacto desses requisitos em nossas descrições de estudo de caso.

Eficiência • Um serviço de arquivo distribuído deve oferecer recursos que tenham pelo menos o mesmo poder e generalidade daqueles encontrados nos sistemas de arquivos

convencionais, e deve obter um nível de desempenho comparável. Birrell e Needham [1980] expressaram seus objetivos de projeto para o CFS (Cambridge File Server) nos seguintes termos:

Desejáramos ter um servidor de arquivos simples, de baixo nível, para compartilhar um recurso dispendioso, a saber, o disco, que nos deixasse livres para projetar um sistema de arquivos mais apropriado para um cliente em particular, ao mesmo tempo em que disponibilizasse um sistema de alto nível compartilhado entre os clientes.

A queda no custo do armazenamento em disco reduziu a importância do primeiro objetivo, mas a percepção da necessidade de uma gama de serviços tratando dos requisitos dos clientes com diferentes objetivos permanece e pode ser melhor resolvida por meio de uma arquitetura modular, do tipo esboçado anteriormente.

As técnicas usadas para a implementação de serviços de arquivos representam uma parte importante do projeto de sistemas distribuídos. Um sistema de arquivos distribuído deve fornecer um serviço que seja comparável aos sistemas de arquivos locais (ou melhor que eles), em termos de desempenho e confiabilidade. Ele deve ser conveniente para administrar, com operações e ferramentas que permitam aos administradores de sistema instalá-lo e operá-lo adequadamente.

12.1.3 Estudos de caso

Construiremos um modelo abstrato para um serviço de arquivos, para atuar como um exemplo introdutório, separando as preocupações com a implementação e fornecendo um modelo simplificado. Descreveremos o Sun Network File System com alguns detalhes, recorrendo ao nosso modelo abstrato mais simples para esclarecer sua arquitetura. O Andrew File System será descrito em seguida, fornecendo uma visão de um sistema de arquivos distribuído que adota uma estratégia diferente para escalabilidade e manutenção da consistência.

Arquitetura do serviço de arquivos • Trata-se de um modelo abstrato de arquitetura que serve tanto para o NFS como para o AFS. Ele é baseado em uma divisão de responsabilidades entre três módulos – um módulo cliente, que simula uma interface de sistema de arquivos convencional para programas aplicativos, e dois módulos servidores que efetuam operações em diretórios e em arquivos. A arquitetura é projetada para permitir uma implementação *sem estado* (*stateless*) dos módulos servidores.

Sun NFS • O NFS (Network File System) da Sun Microsystem foi amplamente adotado na indústria e nos ambientes acadêmicos, desde sua introdução, em 1985. O projeto e desenvolvimento do NFS foram feitos pelo pessoal da Sun Microsystems, em 1984 [Sandberg *et al.* 1985; Sandberg 1987, Callaghan 1999]. Embora vários serviços de arquivo distribuídos já tivessem sido desenvolvidos, e usados em universidades e laboratórios de pesquisa, o NFS foi o primeiro serviço de arquivo projetado como um produto. O projeto e a implementação do NFS obtiveram sucesso tanto técnico como comercial.

Para estimular sua adoção como padrão, as definições das principais interfaces foram colocadas em domínio público [Sun 1989], permitindo que outros fornecedores produzissem implementações, e o código-fonte de uma implementação de referência foi disponibilizado, sob licença, para outros fabricantes de computadores. Atualmente, ele é suportado por muitos fornecedores, e o protocolo NFS (versão 3) é um padrão da Internet, definido no RFC 1813 [Callaghan *et al.* 1995]. O livro de Callaghan sobre o NFS [Callaghan 1999] é uma excelente fonte sobre o projeto e desenvolvimento do NFS e a respeito de tópicos relacionados.

O NFS fornece acesso transparente a arquivos remotos para programas clientes executando em UNIX e outros sistemas operacionais. O relacionamento cliente-servidor é simétrico: cada computador em uma rede NFS pode atuar como cliente e como servi-

dor, e os arquivos de cada máquina podem se tornar disponíveis para acesso remoto por outras máquinas. Qualquer computador pode ser um servidor, exportando alguns de seus arquivos, e cliente, acessando arquivos de outras máquinas. No entanto, é uma prática comum, em instalações maiores, configurar algumas máquinas como servidores dedicados e outras como estações de trabalho.

Um objetivo importante do NFS é obter um alto nível de suporte para heterogeneidade de *hardware* e sistema operacional. O projeto é independente do sistema operacional: existem implementações de cliente e servidor para quase todos os sistemas operacionais e plataformas, incluindo todas as versões do Windows, Mac OS, Linux e qualquer outra versão de UNIX. Implementações do NFS em multiprocessadores de alto desempenho foram desenvolvidas por diversos fornecedores e são usadas para satisfazer os requisitos de armazenamento em intranets com muitos usuários concomitantes.

Andrew File System • O Andrew é um ambiente de computação distribuída, desenvolvido na Carnegie Mellon University (CMU) para uso como sistema de computação e informação do campus [Morris *et al.* 1986]. O projeto do Andrew File System (daqui em diante abreviado como AFS) reflete a intenção de suportar o compartilhamento de informações em larga escala, minimizando a comunicação cliente-servidor. Isso foi conseguido pela transferência de arquivos inteiros entre computadores servidores e clientes, armazenando-os em cache nos clientes até que o servidor receba uma versão mais atualizada. Vamos descrever o AFS-2, a primeira implementação de “produção”, segundo as descrições de Satyanarayanan [1989a; 1989b]. Descrições mais recentes podem ser encontradas em Campbell [1997] e [[Linux AFS](#)].

Inicialmente, o AFS foi implementado na CMU em uma rede de estações de trabalho e servidores executando UNIX BSD e o sistema operacional Mach e, posteriormente, tornou-se disponível em versões comerciais e de domínio público. Uma implementação de domínio público do AFS está disponível no sistema operacional Linux [[Linux AFS](#)]. O AFS foi adotado como a base do sistema de arquivos DCE/DFS no DCE (Distributed Computing Environment) da Open Software Foundation [[www.opengroup.org](#)]. O projeto do DCE/DFS foi além do AFS sob vários aspectos importantes, os quais destacaremos na Seção 12.5.

12.2 Arquitetura do serviço de arquivos

Uma arquitetura que apresenta uma separação clara das principais preocupações no fornecimento de acesso aos arquivos é obtida por meio da estruturação do serviço de arquivo em três componentes – um *serviço de arquivos plano*, um *serviço de diretório* e um *módulo cliente*. Os módulos relevantes e seus relacionamentos aparecem na Figura 12.5. O serviço de arquivos plano e o serviço de diretório exportam uma interface para uso dos programas clientes, e suas interfaces RPC, consideradas em grupo, fornecem um amplo conjunto de operações para acesso aos arquivos. O módulo cliente fornece uma única interface de programação, com operações sobre arquivos semelhantes àquelas encontradas nos sistemas de arquivos convencionais. O projeto é *aberto*, no sentido de que vários módulos clientes podem ser usados para implementar distintas interfaces de programação, simulando as operações de arquivo de vários sistemas operacionais e otimizando o desempenho para diferentes configurações de *hardware* de cliente e servidor.

A divisão de responsabilidades entre os módulos pode ser definida como segue:

Serviço de arquivos plano (flat file service) • O serviço de arquivos plano se preocupa com a implementação de operações sobre o conteúdo dos arquivos. São usados *identificadores exclusivos de arquivo (UFID, Unique File Identifiers)* para fazer referência aos

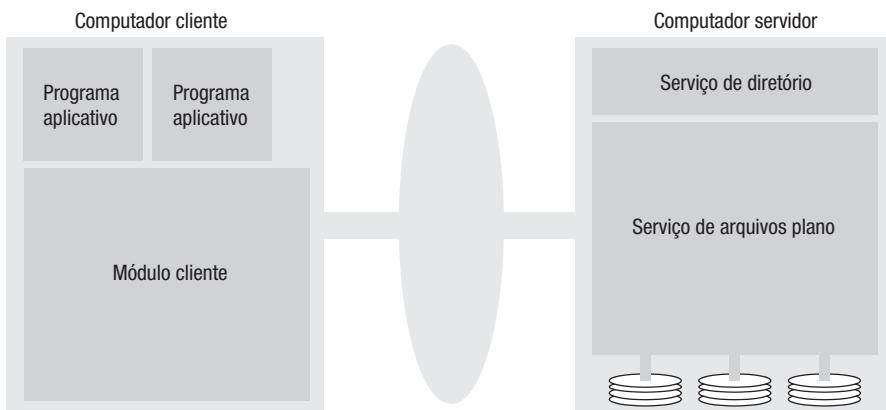


Figura 12.5 Arquitetura do serviço de arquivos.

arquivos em todas as requisições de operações ao serviço de arquivos plano. A divisão de responsabilidades entre o serviço de arquivos e o serviço de diretório é baseada no uso de UFIDs. As UFIDs são longas sequências de bits, escolhidas de modo que cada arquivo tenha um UFID único dentre todos os arquivos de um sistema distribuído. Quando o serviço de arquivos plano recebe uma requisição para criar um arquivo, ele gera um novo UFID e retorna esse UFID para o solicitante.

Serviço de diretório • O serviço de diretório fornece um mapeamento entre *nomes textuais* de arquivos e seus UFIDs. Os clientes podem obter o UFID de um arquivo fornecendo seu nome textual para o serviço de diretório. O serviço de diretório possui as funções necessárias para gerar diretórios, adicionar novos nomes de arquivo a eles e para obter suas UFIDs. Ele é um cliente do serviço de arquivos plano; os arquivos de seu diretório são armazenados em arquivos do serviço de arquivos plano. Quando um esquema de atribuição de nomes hierárquico é adotado, como no UNIX, os diretórios contêm referências para outros diretórios.

Módulo cliente • Um módulo cliente é executado em cada computador cliente, integrando e estendendo as operações do serviço de arquivos plano e do serviço de diretório sob uma interface de programação de aplicativo única, disponível para programas em nível de usuário nos computadores clientes. Por exemplo, em máquinas UNIX, seria fornecido um módulo cliente que simularia o conjunto completo de operações de arquivos UNIX, interpretando nomes de arquivos por meio de requisições iterativas ao serviço de diretório. O módulo cliente também contém informações sobre os locais de rede dos processos servidor de arquivos plano e servidor de diretório. Finalmente, o módulo cliente pode assumir um papel importante na obtenção de um desempenho satisfatório, por meio da implementação de uma cache de blocos de arquivo recentemente usados no cliente.

Interface do serviço de arquivos plano • A Figura 12.6 contém uma definição da interface para um serviço de arquivos planos. Essa é a interface RPC usada pelos módulos clientes. Normalmente, ela não é usada diretamente por programas em nível de usuário. Um argumento *FileId* é inválido se o arquivo a que se refere não está presente no servidor que processa a requisição, ou se suas permissões de acesso são inadequadas para a operação solicitada. Todas as funções da interface, exceto *Create*, geram exceções se o argumento *FileId* contém um UFID inválido ou se o usuário não tem direitos de acesso suficientes. Essas exceções foram omitidas da definição por motivos de clareza.

<i>Read(FileId, i, n) → Data</i>	Se $1 \leq i \leq \text{Length}(\text{File})$: lê uma sequência de até n elementos de um arquivo, começando no elemento i , e a retorna em <i>Data</i> . Gera uma exceção se o valor i é inválido.
<i>Write(FileId, i, Data)</i> — gera <i>BadPosition</i>	Se $1 \leq i \leq \text{Length}(\text{File})+1$: grava uma sequência de <i>Data</i> em um arquivo, começando no elemento i , ampliando o arquivo, se necessário. Gera uma exceção se o valor i é inválido.
<i>create() → FileId</i>	Cria um novo arquivo de tamanho zero e gera um UFID para ele.
<i>Delete(FileId)</i>	Remove o arquivo.
<i>GetAttributes(FileId) → Attr</i>	Retorna os atributos do arquivo.
<i>SetAttributes(FileId, Attr)</i>	Configura os atributos do arquivo (somente os atributos que não estão sombreados na Figura 12.3).

Figura 12.6 Operações do serviço de arquivos plano.

As operações mais importantes são as de leitura e escrita. Tanto a operação *Read* (leitura) como a operação *Write* (escrita) necessitam de um parâmetro i para especificar uma posição no arquivo. A operação *Read* copia para *Data* a sequência de n elementos de dados a partir do elemento i do arquivo especificado. A operação *Write* escreve, a partir do elemento i , a sequência de elementos de dados em *Data* para o arquivo especificado, substituindo o conteúdo anterior do arquivo na posição correspondente e, se necessário, aumentando o arquivo.

Create cria um novo arquivo vazio e retorna o UFID gerado. *Delete* remove o arquivo especificado.

GetAttributes e *SetAttributes* permitem que os clientes acessem o registro de atributos. Normalmente, *GetAttributes* está disponível para qualquer cliente que tenha permissão para ler o arquivo. Normalmente, o acesso à operação *SetAttributes* é restrito ao serviço de diretório que dá acesso ao arquivo. Os valores das partes relativas ao tamanho e ao carimbo de tempo do registro de atributos não são afetados por *SetAttributes*; eles são mantidos separadamente pelo próprio serviço de arquivos plano.

Comparaçāo com o UNIX: nossa interface e as primitivas do sistema de arquivos UNIX são funcionalmente equivalentes. Basta construir um módulo cliente que simule as chamadas de sistema UNIX em termos de nosso serviço de arquivos plano e das operações do serviço de diretório descritas na próxima seção.

Em comparação com a interface UNIX, nosso serviço de arquivos plano não tem as operações *open* e *close* – os arquivos podem ser acessados imediatamente citando-se o UFID apropriado. Em nossa interface, as funções *Read* e *Write* incluem um parâmetro especificando um ponto de partida dentro do arquivo para cada transferência, enquanto as operações equivalentes do UNIX não incluem. No UNIX, cada operação *read* ou *write* começa na posição corrente do ponteiro de leitura e escrita, e esse ponteiro avança de acordo com o número de bytes transferidos após cada operação *read* ou *write*. É fornecida uma operação *seek* para permitir que o ponteiro de leitura e escrita seja posicionado explicitamente.

A interface para nosso serviço de arquivos plano difere da interface de sistema de arquivos do UNIX principalmente por motivos de tolerância a falhas:

Operações que podem ser repetidas: com exceção de *Create*, as operações são *idempotentes*, permitindo o uso da semântica RPC *pelo menos uma vez* – os clientes podem repetir chamadas para as quais não receberam resposta. A execução repetida de *create* produz um novo arquivo diferente para cada chamada.

Servidores sem estado (stateless): a interface é conveniente para a implementação por servidores sem estado. Um servidor sem estado pode ser reiniciado após uma

fallha e retomar a operação sem necessidade dos clientes, ou do servidor, para restaurar qualquer estado.

As operações de arquivo do UNIX não são idempotentes, nem consistentes, com o requisito de uma implementação sem estado. Quando um arquivo é aberto, um ponteiro de leitura e escrita é gerado pelo sistema de arquivos do UNIX, e ele é mantido, junto aos resultados das verificações de controle de acesso até que o arquivo seja fechado. As operações *read* e *write* do UNIX não são idempotentes; se uma operação for repetida accidentalmente, o avanço automático do ponteiro de leitura e escrita resultará no acesso a uma parte diferente do arquivo na operação repetida. O ponteiro de leitura e escrita é uma variável (oculta) de estado relacionado ao cliente. Para imitá-la em um servidor de arquivos, seriam necessárias as operações *open* e *close*, e o valor do ponteiro de leitura e escrita teria de ser mantido pelo servidor enquanto o arquivo relevante estivesse aberto. Eliminando o ponteiro de leitura e escrita, eliminamos a parte da necessidade do servidor de arquivos de manter informações de estado em nome de clientes específicos.

Controle de acesso • No sistema de arquivos do UNIX, os direitos de acesso do usuário são verificados com relação ao *modo* de acesso (leitura ou escrita) solicitado na chamada *open* (a Figura 12.4 mostra a API do sistema de arquivos do UNIX), e o arquivo é aberto somente se o usuário tiver os direitos necessários. A identidade de usuário (UID) utilizada na verificação dos direitos de acesso é o resultado do *login* anteriormente autenticado do usuário e não pode ser falsificada em implementações não distribuídas. Os direitos de acesso resultantes são mantidos até que o arquivo seja fechado, e mais nenhuma verificação é exigida quando são solicitadas operações subsequentes no mesmo arquivo.

Nas implementações distribuídas, as verificações de direitos de acesso não precisam ser realizadas no servidor, pois, sob outros aspectos, a interface RPC do servidor é um ponto desprotegido de acesso aos arquivos. Uma identidade de usuário precisa ser passada com as requisições, e o servidor é vulnerável a identidades falsificadas. Além disso, se os resultados de uma verificação de direitos de acesso fossem mantidos no servidor e usados para acessos futuros, o servidor não seria mais sem estado. Podem ser adotadas duas estratégias alternativas para este último problema:

- É feita uma verificação de acesso quando um nome de arquivo é convertido em um UFID, e os resultados são codificados na forma de uma capacidade (veja a Seção 11.2.4), a qual é retornada para o cliente, para envio com as requisições subsequentes.
- Uma identidade de usuário é enviada com cada requisição de cliente e as verificações de acesso são realizadas pelo servidor para cada operação de arquivo.

Os dois métodos permitem implementação de servidor sem estado e ambos têm sido usados em sistemas de arquivos distribuídos. O segundo é mais comum; ele é usado no NFS e no AFS. Nenhuma dessas estratégias resolve o problema de segurança relativo às identidades de usuário falsificadas. Vimos, no Capítulo 11, que isso pode ser tratado com o uso de assinaturas digitais. O Kerberos é um esquema de autenticação eficaz que tem sido aplicado no NFS e no AFS.

Em nosso modelo abstrato, não fazemos nenhuma suposição sobre o método pelo qual o controle de acesso é implementado. A identidade do usuário é passada como um parâmetro implícito e pode ser usada quando for necessário.

Interface do serviço de diretório • A Figura 12.7 contém uma definição da interface RPC para um serviço de diretório. O principal objetivo do serviço de diretório é fornecer a

<i>Lookup(Dir, Name) → FileId</i> — gera <i>NotFound</i>	Localiza o nome textual no diretório e retorna o UFID relevante. Se <i>Name</i> não estiver no diretório, gera uma exceção.
<i>AddName(Dir, Name, FileId)</i> — gera <i>NameDuplicate</i>	Se <i>Name</i> não estiver no diretório, adiciona <i>(Name, File)</i> no diretório e atualiza o registro de atributos do arquivo. Se <i>Name</i> já estiver no diretório, gera uma exceção.
<i>UnName(Dir, Name)</i> — gera <i>NotFound</i>	Se <i>Name</i> estiver no diretório, a entrada contendo <i>Name</i> é removida do diretório. Se <i>Name</i> não estiver no diretório, gera uma exceção.
<i>GetNames(Dir, Pattern) → NameSeq</i>	Retorna todos os nomes textuais presentes no diretório que correspondam à expressão regular <i>Pattern</i> .

Figura 12.7 Operações do serviço de diretório.

transformação de nomes textuais em UFIDs. Para isso, ele mantém arquivos de diretório contendo os mapeamentos entre nomes textuais de arquivos e UFIDs. Cada diretório é armazenado como um arquivo convencional com um UFID, de modo que o serviço de diretório é um cliente do serviço de arquivos.

Definimos apenas operações sobre diretórios individuais. Para cada operação, é exigido o UFID do arquivo que contém o diretório (no parâmetro *Dir*). A operação *Lookup* no serviço de diretório realiza a transformação *Nome → UFID*. Ela é um bloco básico para uso em outros serviços, ou no módulo cliente, para realizar transformações mais complexas, como a interpretação de nome hierárquica encontrada no UNIX. Como antes, as exceções causadas por direitos de acesso inadequados foram omitidas das definições.

Existem duas operações para alterar diretórios: *AddName* e *UnName*. *AddName* adiciona uma entrada em um diretório e incrementa o campo de contagem de referência no registro de atributos do arquivo.

UnName remove uma entrada de um diretório e decremente a contagem de referência. Se a contagem de referência chegar a zero, o arquivo é removido. *GetNames* é fornecida para permitir que os clientes examinem o conteúdo dos diretórios e para implementar operações de correspondência de padrão em nomes de arquivo, como aquelas encontradas no *shell* do UNIX. Ela retorna todos os nomes (ou um subconjunto deles) armazenados em determinado diretório. Os nomes são selecionados pela correspondência de padrão contra uma expressão regular fornecida pelo cliente.

A existência da correspondência de padrão na operação *GetNames* permite que os usuários determinem os nomes de um ou mais arquivos fornecendo uma especificação incompleta dos caracteres presentes nos nomes. Uma expressão regular é uma especificação para uma classe de *strings*, na forma de uma expressão contendo uma combinação de *substrings* literais e símbolos denotando caracteres variáveis ou ocorrências repetidas de caracteres ou *substrings*.

Sistema de arquivos hierárquico • Um sistema de arquivos hierárquico, como aquele fornecido pelo UNIX, consiste em vários diretórios organizados em uma estrutura em árvore. Cada diretório contém os nomes dos arquivos e de outros diretórios que podem ser acessados a partir dele. Qualquer arquivo ou diretório pode ser referenciado usando-se um *nome de caminho (pathname)* – um nome composto de várias partes que representa um caminho através da árvore. A raiz tem um nome distinto e cada arquivo, ou (sub)dire-

tório, possui um nome definido pelo usuário e armazenado em um diretório. O esquema de atribuição de nomes de arquivo do UNIX não é rigorosamente hierárquico – os arquivos podem ter vários nomes (*alias*) que podem estar no mesmo diretório ou em diretórios diferentes. Isso é implementado por meio de uma operação *link*, a qual adiciona um novo nome para um arquivo em um diretório especificado.

Um sistema de atribuição de nomes de arquivos como o do UNIX pode ser implementado pelo módulo cliente por meio dos serviços de arquivo plano e de diretório já definidos. Uma hierarquia de diretórios estruturada em árvore é construída com arquivos nas folhas e diretórios nos outros nós da árvore. A raiz da árvore é um diretório com um UFID conhecido. É possível atribuir vários nomes a um arquivo por meio da operação *AddName* e com o campo de contagem de referência no registro de atributos.

No módulo cliente, pode ser fornecida uma função que obtenha o UFID de um arquivo a partir de seu nome de caminho. A função interpreta o nome de caminho a partir da raiz, usando *Lookup* para obter o UFID de cada parte presente no caminho.

Em um serviço de diretório hierárquico, os atributos associados aos arquivos devem incluir um campo de tipo que faça a distinção entre arquivos normais e de diretórios. Isso é usado ao se seguir um caminho para garantir que cada parte do nome, exceto a última, refira-se a um diretório.

Grupos de arquivos • Um *grupo de arquivos* é um conjunto de arquivos localizado em determinado servidor. Um servidor pode conter vários grupos de arquivos e os grupos podem ser movidos entre os servidores, mas um arquivo não pode mudar do grupo a que pertence. Uma construção semelhante (*filesystem*) é usada no UNIX e na maioria dos outros sistemas operacionais. Os grupos de arquivos foram introduzidos originalmente para suportar recursos para mover conjuntos de arquivos armazenados em mídia removível entre computadores. Em um serviço de arquivos distribuído, os grupos de arquivos suportam a alocação de arquivos em unidades lógicas maiores e permitem que o serviço seja implementado com arquivos armazenados em vários servidores. Em um sistema de arquivos distribuído que suporta grupos de arquivos, a representação de UFIDs inclui um componente identificador de grupo de arquivos, permitindo que o módulo cliente assuma a responsabilidade por enviar as requisições para o servidor que contém o grupo de arquivos em questão.

Os identificadores de grupo de arquivos devem ser únicos em todo um sistema distribuído. Como os grupos de arquivos podem ser movidos, os sistemas de arquivos inicialmente separados podem ser combinados para formar um só sistema de arquivo, e a única maneira de garantir que os identificadores de grupo de arquivos sejam sempre diferentes em determinado sistema de arquivo é gerá-los com um algoritmo que garanta a exclusividade global. Por exemplo, quando um novo grupo de arquivos é criado, um identificador exclusivo pode ser gerado pela concatenação do endereço IP de 32 bits do computador que está criando o novo grupo, com um inteiro de 16 bits derivado da data, produzindo um inteiro único de 48 bits:



Note que o endereço IP não pode ser usado para localizar o grupo de arquivos, pois ele pode ser movido para outro servidor. Em vez disso, o serviço de arquivo deve manter um mapeamento entre os identificadores de grupo e os servidores.

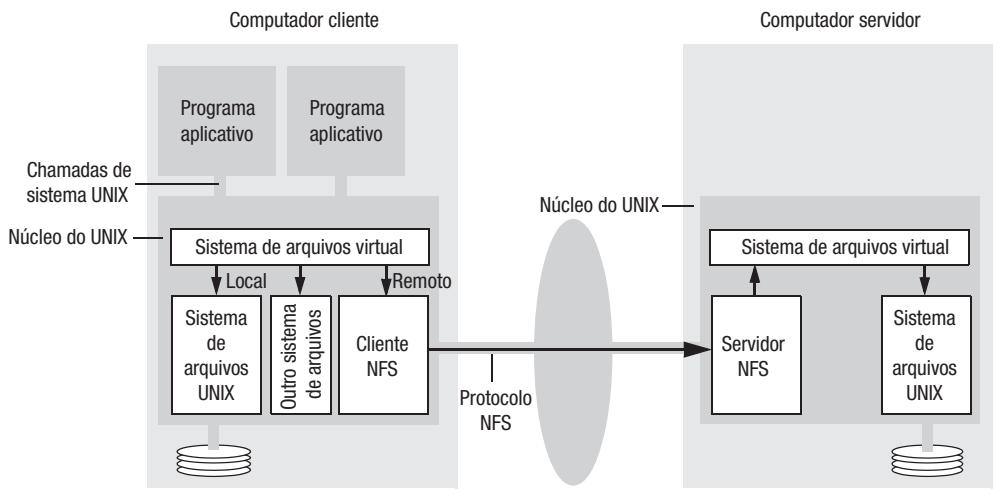


Figura 12.8 Arquitetura do NFS.

12.3 Estudo de caso: Sun Network File System

A Figura 12.8 mostra a arquitetura do NFS. Ela segue o modelo abstrato definido na seção anterior. Todas as implementações de NFS suportam o protocolo NFS – um conjunto de chamadas de procedimentos remotos que fornece o meio para os clientes efetuarem operações em um meio de armazenamento de arquivos remoto. O protocolo NFS é independente do sistema operacional, mas foi originalmente desenvolvido para uso em redes UNIX, e vamos descrever a implementação UNIX do protocolo NFS (versão 3).

O módulo *servidor NFS* reside no núcleo de cada computador que atua como servidor NFS. As requisições que se referem a arquivos de um sistema de arquivos remoto são transformadas em operações do protocolo NFS pelo módulo cliente e depois passadas para o módulo servidor NFS no computador que contém o sistema de arquivos em questão.

Os módulos cliente e servidor NFS se comunicam usando chamadas de procedimentos remotos. O sistema RPC da Sun, descrito na Seção 5.3.3, foi desenvolvido para ser usado no NFS. Ele pode ser configurado para usar UDP ou TCP, e o protocolo NFS é compatível com ambos. É incluído um serviço mapeador de porta (*portmapper*) para permitir que os clientes se associem, por meio de um nome, aos serviços de determinado computador. A interface RPC do servidor NFS é aberta: qualquer processo pode enviar requisições para um servidor NFS; caso as requisições sejam válidas e incluam credenciais de usuários autorizados, elas serão atendidas. Como um recurso de segurança opcional, pode-se exigir o envio de credenciais de usuário assinadas, assim como a criptografia dos dados para se obter privacidade e integridade.

Sistema de arquivos virtual • A Figura 12.8 torna claro que o NFS fornece transparência de acesso: os programas de usuário podem executar operações de arquivo, em arquivos locais ou remotos, sem distinção. Podem estar presentes outros sistemas de arquivos distribuídos que suportem chamadas de sistema UNIX e, se assim for, eles seriam integrados da mesma maneira.

A integração é obtida por meio de um módulo de sistema de arquivos virtual (VFS, Virtual File System), o qual foi adicionado no núcleo do UNIX para fazer a distinção entre arquivos locais e remotos, e para transformar os identificadores de arquivo usados

pelo NFS em identificadores internos normalmente usados no UNIX e em outros sistemas de arquivos. Além disso, o VFS controla os sistemas de arquivos que estão correntemente disponíveis tanto de forma local como remota e passa cada requisição para o módulo apropriado (o sistema de arquivos UNIX, o módulo de cliente NFS ou o módulo de serviço de outro sistema de arquivos).

Os identificadores de arquivo usados no NFS são chamados de *manipuladores de arquivo*. Um manipulador de arquivo não é visível para os clientes e contém as informações de que o servidor precisa para distinguir um arquivo individual. Nas implementações UNIX do NFS, o manipulador de arquivo é derivado do *i-node* do arquivo, adicionando-se dois campos extras, como dado a seguir (o *i-node* de um arquivo UNIX é um número que serve para identificar e localizar o arquivo dentro do sistema de arquivos no qual ele está armazenado):

<i>Manipulador de arquivo:</i>	<i>Identificador do sistema de arquivos</i>	<i>i-node</i> do arquivo	<i>número de geração do i-node</i>
--------------------------------	---	--------------------------	------------------------------------

O NFS adota a montagem de sistemas de arquivos* (*filesystem*) do UNIX como unidade de agrupamento de arquivos, definida na seção anterior. O campo *identificador de filesystem* é um número exclusivo alocado para cada sistema de arquivos (*filesystem*) quando ele é criado (e, na implementação UNIX, é armazenado no superbloco do sistema de arquivos). O número de geração do *i-node* é necessário porque no sistema de arquivos UNIX convencional os *i-nodes* são reutilizados após um arquivo ser removido. Nas extensões VFS do sistema de arquivos UNIX, um número de geração é armazenado com cada arquivo e é incrementado sempre que um *i-node* é reutilizado (por exemplo, em uma chamada de sistema *creat*). O cliente obtém o primeiro manipulador de arquivo para um sistema de arquivo remoto ao montá-lo. Os manipuladores de arquivo são passados do servidor para o cliente nos resultados das operações *lookup*, *create* e *mkdir* (veja a Figura 12.9) e do cliente para o servidor como um dos argumentos que as requisições contêm.

A camada de sistema de arquivos virtual tem uma estrutura VFS para cada sistema de arquivos (*filesystem*) montado e um *v-node* por arquivo aberto. Uma estrutura VFS relaciona um sistema de arquivo remoto com o diretório local em que ele é montado. O *v-node* contém uma indicação se um arquivo é local ou remoto. Se o arquivo é local, o *v-node* contém uma referência para o índice do arquivo local (um *i-node* em uma implementação UNIX). Se o arquivo é remoto, ele contém o manipulador de arquivo do arquivo remoto.

Integração com o cliente • O módulo cliente NFS desempenha o papel descrito para o módulo cliente em nosso modelo básico, fornecendo uma interface conveniente para os programas aplicativos. Porém, ao contrário do nosso modelo de módulo cliente, ele simula precisamente a semântica das primitivas padrão do sistema de arquivos UNIX e é integrado com o núcleo UNIX. As principais razões para que ele seja integrado, e não fornecido como uma biblioteca a ser ligada nos processos clientes, são para permitir que:

- os programas de usuário possam acessar arquivos por meio de chamadas de sistema UNIX sem serem ligados e compilados novamente;
- um único módulo cliente atenda a todos os processos em nível de usuário, com uma cache compartilhada dos blocos usados recentemente (descrito a seguir);

* N. de R.T.: Em inglês, há uma distinção entre os termos *filesystem* (uma única palavra) e *file system* (duas palavras). Por *filesystem*, entende-se o conjunto de arquivos e diretórios mantidos em uma partição de um dispositivo de armazenamento; *file system* refere-se ao módulo de *software* do sistema operacional que provê acesso aos arquivos. Corriqueiramente, em português, o termo *sistema de arquivos* engloba as duas funcionalidades e é empregado para referenciar ambos. Quando essa distinção for necessária, usaremos os termos em inglês entre parênteses.

<i>lookup(dirfh, name) → fh, attr</i>	Retorna o manipulador de arquivo e os atributos do arquivo <i>name</i> no diretório <i>dirfh</i> .
<i>create(dirfh, name, attr) → newfh, attr</i>	Cria um arquivo <i>name</i> no diretório <i>dirfh</i> com atributos <i>attr</i> e retorna o novo manipulador de arquivo e os seus atributos.
<i>remove(dirfh, name) → status</i>	Remove o arquivo <i>name</i> do diretório <i>dirfh</i> .
<i>getattr(fh) → attr</i>	Retorna os atributos do arquivo <i>fh</i> . (Semelhante à chamada de sistema <i>stat</i> do UNIX.)
<i>setattr(fh, attr) → attr</i>	Configura os atributos (modo, ID de usuário, ID de grupo, tamanho, horário de acesso e de modificação de um arquivo). Configurar o tamanho como zero trunca o arquivo.
<i>read(fh, offset, count) → attr, data</i>	Retorna até <i>count</i> bytes de dados de um arquivo, a partir de <i>offset</i> . Além disso, retorna os últimos atributos do arquivo.
<i>write(fh, offset, count, data) → attr</i>	Grava <i>count</i> bytes de dados em um arquivo, a partir de <i>offset</i> . Retorna os atributos do arquivo após a gravação terminar.
<i>rename(dirfh, name, todirfh, toname) → status</i>	Muda o nome do arquivo <i>name</i> no diretório <i>dirfh</i> para <i>toname</i> no diretório para <i>dirfh</i> .
<i>link(newdirfh, newname, fh) → status</i>	Cria uma entrada <i>newname</i> no diretório <i>newdirfh</i> referindo-se ao arquivo ou diretório <i>fh</i> .
<i>symlink(newdirfh, newname, string) → status</i>	Cria uma entrada <i>newname</i> no diretório <i>newdirfh</i> de tipo <i>vínculo simbólico</i> (<i>link</i> simbólico) com o valor <i>string</i> . O servidor não interpreta o <i>string</i> , mas produz um arquivo de vínculo simbólico para contê-la.
<i>readlink(fh) → string</i>	Retorna o <i>string</i> associado ao arquivo de vínculo simbólico identificado por <i>fh</i> .
<i>mkdir(dirfh, name, attr) → newfh, attr</i>	Cria um novo diretório <i>name</i> com atributos <i>attr</i> e retorna seu manipulador de arquivo e atributos.
<i>rmdir(dirfh, name) → status</i>	Remove o diretório vazio <i>name</i> do diretório pai <i>dirfh</i> . Falha se o diretório não estiver vazio.
<i>readdir(dirfh, cookie, count) → entries</i>	Retorna até <i>count</i> bytes de entradas do diretório <i>dirfh</i> . Cada entrada contém um nome de arquivo, um manipulador de arquivo e um ponteiro para a próxima entrada de diretório, chamado de <i>cookie</i> . O <i>cookie</i> é usado nas chamadas de <i>readdir</i> subsequentes para começar a ler a partir da entrada seguinte. Se o valor de <i>cookie</i> for zero, lê a partir da primeira entrada no diretório.
<i>statfs(fh) → fsstats</i>	Retorna informações do sistema de arquivos (como o tamanho do bloco, número de blocos livres, etc.) que contêm um arquivo <i>fh</i> .

Figura 12.9 Operações do servidor NFS (protocolo NFS versão 3, simplificado).

- a chave de criptografia usada para autenticar as IDs de usuário passadas para o servidor (veja a seguir) possa ser mantida no núcleo, impedindo a personificação por parte de clientes em nível de usuário.

O módulo cliente NFS coopera com o sistema de arquivos virtual em cada máquina cliente. Ele funciona de maneira semelhante ao sistema de arquivos convencional do UNIX, transferindo blocos de arquivos para o servidor (e dele) e, quando possível, colocando os blocos em cache na memória local. Ele compartilha a mesma cache usada pelo sistema de entrada e saída local. Porém, como vários clientes, em diferentes máquinas, podem acessar simultaneamente o mesmo arquivo remoto, surge um problema de consistência de cache.

Controle de acesso e autenticação • Ao contrário do sistema de arquivos UNIX convencional, o servidor NFS é sem estado e não mantém arquivos abertos no nome de seus clientes.

Portanto, a cada requisição, o servidor deve verificar novamente a identidade do usuário nos atributos de permissão de acesso do arquivo, para ver se o usuário pode acessar o arquivo da maneira solicitada. O protocolo RPC Sun exige que os clientes enviem informações de autenticação de usuário (por exemplo, a ID de usuário e a ID de convencionais – em 16 bits – do UNIX) com cada requisição, e isso é verificado em relação à permissão de acesso presente nos atributos do arquivo. Esses parâmetros adicionais não aparecem em nossa visão geral do protocolo NFS da Figura 12.9; eles são fornecidos automaticamente pelo sistema RPC.

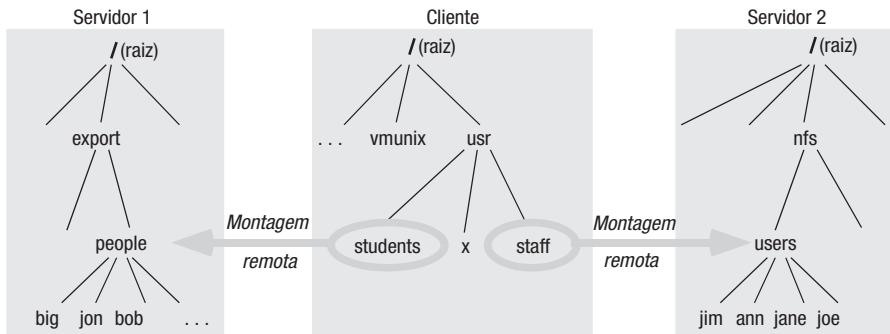
Em sua forma mais simples, existe uma brecha de segurança nesse mecanismo de controle de acesso. Um servidor NFS fornece uma interface RPC convencional em uma porta bem conhecida em cada computador, e qualquer processo pode se comportar como cliente, enviando requisições para o servidor para acessar ou atualizar um arquivo. O cliente pode modificar as chamadas de RPC para incluir a ID de qualquer usuário, personificando-o sem seu conhecimento ou permissão. Essa brecha de segurança foi fechada pelo uso de uma opção do protocolo RPC para a criptografia DES das informações de autenticação do usuário. Mais recentemente, o Kerberos foi integrado com o Sun NFS para proporcionar uma solução mais forte e abrangente para os problemas da autenticação de usuário e da segurança, e descreveremos isso a seguir.

Interface do servidor NFS • Uma representação simplificada da interface RPC fornecida pelos servidores NFS versão 3 (definidos no RFC 1813 [Callaghan *et al.* 1995]) aparece na Figura 12.9. As operações de acesso a arquivo do NFS, *read*, *write*, *getattr* e *setattr* são quase idênticas às operações *Read*, *Write*, *GetAttributes* e *SetAttributes* definidas para nosso modelo de serviço de arquivo simples (Figura 12.6). A operação *lookup* e a maior parte das outras operações de diretório, definidas na Figura 12.9, são semelhantes àquelas de nosso modelo de serviço de diretório (Figura 12.7).

As operações de arquivo e diretório são integradas em um único serviço; a criação e a inserção de nomes de arquivo em diretórios são realizadas por uma única operação *create*, que recebe como argumentos o nome textual do novo arquivo e o manipulador do arquivo do diretório de destino. As outras operações NFS sobre diretórios são *create*, *remove*, *rename*, *link*, *symlink*, *readlink*, *mkdir*, *rmdir*, *readdir* e *statfs*. Elas são semelhantes às suas correlatas do UNIX, com exceção de *readdir*, que fornece um método independente de representação para ler o conteúdo de diretórios, e *statfs*, que fornece as informações de status em sistemas de arquivos remotos.

Serviço de montagem • A montagem de subárvores de sistemas de arquivos (*filesystems*) remotos feita por clientes é suportada por um processo separado, o *serviço de montagem*, executado em nível de usuário em cada computador servidor NFS. Em cada servidor existe um arquivo com um nome conhecido (*/etc/exports*), contendo os nomes dos sistemas de arquivos locais (*local filesystems*) que estão disponíveis para montagem remota. Uma lista de acesso é associada a cada nome de sistema de arquivos (*filesystem*), indicando quais máquinas podem montá-los.

Os clientes usam uma versão modificada do comando *mount* do UNIX para solicitar a montagem de um sistema de arquivos remoto, especificando o nome de computador remoto, o nome de caminho de um diretório no sistema de arquivos remoto e o nome local com o qual ele será montado. O diretório remoto pode ser qualquer subárvore do sistema de arquivo remoto solicitado, permitindo aos clientes montarem qualquer parte sua. O comando *mount* modificado se comunica com o processo do serviço de montagem no computador remoto, usando um *protocolo de montagem*. Trata-se de um protocolo RPC que inclui uma operação que recebe um nome de caminho de diretório e retorna o manipulador do arquivo do diretório especificado, caso o cliente tenha permissão de acesso para o sistema de arqui-



Nota: O sistema de arquivos montado em /usr/students no cliente é, na verdade, a subárvore localizada em /export/people no Servidor 1; o sistema de arquivos montado em /usr/staff no cliente é, na verdade, a subárvore localizada em /nfs/users no Servidor 2.

Figura 12.10 Sistemas de arquivos local e remoto acessíveis em um cliente NFS.

vos relevante. A localização (endereço IP e número de porta) do servidor e do manipulador de arquivo do diretório remoto são passados na camada VFS e no cliente NFS.

A Figura 12.10 ilustra um *Cliente* com dois sistemas de arquivos montados de forma remota. Os nós *people* e *users* nos sistemas de arquivos de *Servidor 1* e *Servidor 2* são montados sobre os nós *students* e *staff* no sistema de arquivos local do *Cliente*. O significado disso é que os programas executados no *Cliente* podem acessar arquivos em *Servidor 1* e em *Servidor 2*, usando nomes de caminho como */usr/students/jon* e */usr/staff/ann*.

Os sistemas de arquivos remotos podem ser *montados incondicionalmente* ou *condicionalmente* em um computador cliente. Quando um processo em nível de usuário acessa um arquivo em um sistema de arquivos montado incondicionalmente, o processo é suspenso até que a requisição possa ser concluída e, se o computador remoto estiver indisponível por qualquer razão, o módulo cliente NFS continuará a fazer novas tentativas até ser atendido. Assim, no caso de uma falha do servidor, os processos em nível de usuário são suspensos até que o servidor seja reiniciado e, depois, continuam normalmente. No entanto, se o sistema de arquivos relevante for montado condicionalmente, o módulo cliente NFS retornará uma indicação de erro para os processos em nível de usuário, após um pequeno número de novas tentativas. Então, os programas construídos corretamente detectarão a falha e executarão ações apropriadas de recuperação ou de geração de relatórios. Entretanto, muitos utilitários e aplicativos UNIX não testam a existência de falhas em operações de acesso a arquivo e se comportam de maneiras imprevisíveis no caso de falha de um sistema de arquivos montado condicionalmente. Por isso, muitas instalações utilizam exclusivamente a montagem incondicional, com a consequência de que os programas são incapazes de se recuperar normalmente quando um servidor NFS fica indisponível por um período de tempo significativo.

Tradução de nomes de caminho • Os sistemas de arquivos UNIX traduzem *nomes de caminho* de arquivos composto por muitas partes em referências a *i-nodes*, em um processo passo a passo, quando são usadas as chamadas de sistema *open*, *creat* ou *stat*. No NFS, os nomes de caminho não podem ser traduzidos em um servidor, pois um nome pode cruzar um ponto de montagem no cliente – diretórios contendo várias partes em nome podem residir em sistemas de arquivos de diferentes servidores. Portanto, os nomes de caminho são analisados e sua tradução é realizada de maneira iterativa pelo cliente. Cada parte de um nome que se refere a um diretório montado de forma remota é transformada em um manipulador de arquivo usando uma requisição de *lookup* separada para o servidor remoto.

A operação *lookup* procura uma única parte de um nome de caminho em determinado diretório e retorna o manipulador de arquivo e os atributos correspondentes. O manipulador de arquivo retornado em um passo anterior é usado como parâmetro no próximo passo da operação *lookup*. Como os manipuladores de arquivo são opacos (não interpretados) para o código do cliente NFS, o sistema de arquivos virtual é responsável por traduzir o manipulador de arquivo em um diretório local, ou remoto, e por realizar o procedimento indireto necessário quando fizer referência a um ponto de montagem local. O armazenamento em cache dos resultados de cada passo diminui a aparente ineficiência desse processo, tirando proveito do caráter local da referência a arquivos e diretórios; isto é, normalmente, os usuários e os programas acessam arquivos em apenas um diretório ou em um pequeno número de diretórios.

Montagem automática (automounter) • A montagem automática (*automounter*) foi adicionada na implementação UNIX do NFS para montar dinamicamente um diretório remoto quando um ponto de montagem “vazio” fosse referenciado por um cliente. A implementação original da montagem automática era executada como um processo UNIX em nível de usuário em cada computador cliente. Versões posteriores (chamadas de *autofs*) foram implementadas no núcleo do Solaris e do Linux. Descreveremos aqui a versão original.

O *automounter* mantém uma tabela de pontos de montagem (nomes de caminho), com uma referência, para cada um deles, de um ou mais servidores NFS. Ele se comporta como um servidor NFS local na máquina cliente. Quando o módulo cliente NFS tenta solucionar um nome de caminho que inclui um desses pontos de montagem, ele faz uma requisição de *lookup()* para o *automounter* local, que localiza em sua tabela o sistema de arquivos exigido e envia uma requisição de *sondagem (probe)* para cada servidor listado. O sistema de arquivos do primeiro servidor a responder é, então, montado no cliente, usando o serviço de montagem normal. O sistema de arquivos montado é associado ao ponto de montagem usando um vínculo simbólico (*symbolic link*), de modo que os acessos a ele não resultarão em mais requisições para o *automounter*. Então, o acesso ao arquivo prossegue normalmente, sem mais referências ao *automounter*, a não ser que não existam referências a esse vínculo simbólico por vários minutos. Neste caso, o *automounter* desmonta o sistema de arquivos remoto.

As implementações posteriores, no núcleo, substituíram os vínculos simbólicos por montagens reais, evitando alguns problemas que surgiam em aplicativos que armazenavam em cache os nomes de caminho temporários usados pelo *automounter* em nível de usuário [Callaghan 1999].

Uma forma simples de replicação, somente em leitura, pode ser obtida por um conjunto de vários servidores contendo cópias idênticas de um sistema de arquivos, ou de uma subárvore de arquivos, com uma única entrada na tabela do *automounter*. Isso é útil para sistemas de arquivos muito utilizados que mudam pouco, como os binários de sistema do UNIX. Por exemplo, cópias do diretório */usr/lib* e de sua subárvore poderiam ser mantidas em mais de um servidor. Na primeira ocasião em que um arquivo de */usr/lib* fosse aberto em um cliente, todos os servidores receberiam as mensagens de *probe* e o primeiro a responder seria montado no cliente. Isso proporciona um grau limitado de tolerância a falhas e balanceamento de carga, pois o primeiro servidor a responder será um que não falhou e é provavelmente um que não esteja muito ocupado com o atendimento de outras requisições.

Uso de cache no servidor • O uso de cache, tanto no cliente como no servidor, é um recurso indispensável para obter um desempenho adequado nas implementações do NFS.

Nos sistemas UNIX convencionais, blocos de arquivo, diretórios e atributos de arquivo que foram lidos do disco são mantidos em *cache* na memória principal até que o

espaço da cache seja exigido para outros blocos. Se um processo faz uma requisição de leitura ou escrita de um bloco que já está na cache, ele pode ser atendido sem outro acesso ao disco. A *leitura antecipada* (*read-ahead*) adianta os acessos de leitura e busca os blocos seguintes àqueles lidos mais recentemente, e a *escrita postergada* (*delayed-write*) otimiza as operações de escrita: quando um bloco tiver sido alterado (por uma requisição de escrita), seu novo conteúdo será gravado no disco somente quando o espaço ocupado por esse bloco na cache for exigido por outro. Para evitar a perda de dados no caso de falha do sistema, a operação *sync* do UNIX passa para o disco, a cada 30 segundos, os blocos alterados. Essas técnicas de uso de cache funcionam em um ambiente UNIX convencional porque todas as requisições de leitura e escritas feitas pelos processos em nível de usuário passam por uma única cache, que é implementada no espaço do núcleo do UNIX. A cache é sempre mantida atualizada e os acessos aos arquivos devem sempre passar por ela.

Os servidores NFS usam cache exatamente como ela é empregada para outros acessos a arquivos. O uso da cache do servidor para conter blocos de disco lidos recentemente não acarreta nenhum problema de consistência, mas quando um servidor realiza operações de escritas, medidas extras são necessárias para garantir que os clientes possam ter confiança de que os resultados das operações de escrita sejam persistentes, mesmo quando ocorre uma falha do servidor. Na versão 3 do protocolo NFS, a operação *write* oferece duas opções (não mostradas na Figura 12.9):

1. Nas operações *write*, os dados recebidos dos clientes são armazenados na cache no servidor e gravados no disco, antes que uma resposta seja enviada para o cliente. Isso é chamado de uso de cache com *escrita direta* (*write-through*). O cliente pode ter certeza de que os dados estão armazenados de modo persistente, assim que a resposta for recebida.
2. Nas operações *write*, os dados são armazenados apenas na cache de memória. Eles serão gravados no disco quando uma operação de confirmação (*commit*) for recebida para o arquivo relevante. O cliente tem certeza de que os dados estão armazenados de modo persistente somente quando a resposta da operação *commit* for recebida. Os clientes NFS padrão usam esse modo de operação, executando uma operação *commit* quando um arquivo aberto para gravação for fechado.

Commit é uma operação adicional fornecida na versão 3 do protocolo NFS; ela foi incluída para superar o gargalo de desempenho causado pelo modo de operação *write-through* nos servidores que recebem grandes quantidades de operações *write*.

O requisito de *write-through* em sistemas de arquivos distribuídos é um caso de modos de falha independentes, discutidos no Capítulo 1 – os clientes continuam a operar quando um servidor falha, e os programas aplicativos podem executar ações supondo que os resultados das escritas anteriores foram enviados para armazenamento em disco. É improvável que isso ocorra no caso de atualizações de arquivos locais, pois é quase certo que a falha de um sistema de arquivos local resulte na falha de todos os processos aplicativos em execução no mesmo computador.

Uso de cache no cliente • O módulo cliente NFS armazena em cache os resultados das operações *read*, *write*, *getattr*, *lookup* e *readdir*, para reduzir o número de requisições feitas aos servidores. O uso de cache no cliente implica a possibilidade de existirem diversas versões de arquivos, ou porções deles, em diferentes clientes, pois as escritas feitas por um cliente não resultam na atualização imediata em outros clientes das cópias do mesmo arquivo armazenados em cache. Em vez disso, os clientes são responsáveis por fazerem uma consulta sequencial no servidor para verificar se os dados armazenados em cache que eles contêm são atuais.

Um método baseado em carimbos de tempo é usado para validar blocos armazenados em cache antes de serem usados. Cada elemento de dados, ou de metadados, presente na cache é rotulado com dois carimbos de tempo (*timestamps*):

T_c é a hora em que a entrada da cache foi validada pela última vez.

T_m é a hora em que o bloco foi modificado pela última vez no servidor.

Uma entrada da cache é válida no tempo T , se $T - T_c$ for menor que um intervalo de atualização t , ou se o valor de T_m gravado no cliente corresponder ao valor T_m presente no servidor (isto é, os dados não foram modificados no servidor desde que a entrada na cache foi feita). Formalmente, a condição de validade é:

$$(T - T_c < t) \vee (T_{m_{cliente}} = T_{m_{servidor}})$$

A escolha de um valor para t é um compromisso entre consistência e eficiência. Um intervalo de atualização muito curto resultará em uma forte aproximação da consistência de cópia única, ao custo de uma carga relativamente pesada de chamadas para o servidor para verificar o valor de $T_{m_{servidor}}$. Nos clientes Sun Solaris, t é configurado de forma adaptativa para arquivos individuais, com um valor no intervalo de 3 a 30 segundos, dependendo da frequência das atualizações no arquivo. Para diretórios, o intervalo é de 30 a 60 segundos, refletindo o menor risco de atualizações concorrentes.

Existe um valor de $T_{m_{servidor}}$ para todos os blocos de dados de um arquivo e outro para os atributos de arquivo. Como os clientes NFS não podem determinar se um arquivo está sendo compartilhado ou não, o procedimento de validação deve ser usado para todos os acessos ao arquivo. Uma verificação de validade é realizada quando uma entrada da cache é usada. A primeira metade da condição de validade pode ser avaliada sem nenhum acesso ao servidor. Se ela for verdadeira, então a segunda metade não precisará ser avaliada; se for falsa, o valor corrente de $T_{m_{servidor}}$ será obtido (por meio de uma chamada de *getattr* no servidor) e comparado com o valor local $T_{m_{cliente}}$. Se eles forem iguais, então a entrada da cache será considerada válida, e o valor de T_c dessa entrada da cache será atualizado com o tempo correto. Se eles diferirem, então os dados armazenados em cache foram atualizados no servidor e a entrada da cache será invalidada, resultando em uma requisição para o servidor fornecer os dados relevantes.

Várias medidas são usadas para reduzir o tráfego das chamadas de *getattr* para o servidor:

- Quando um novo valor de $T_{m_{servidor}}$ é recebido em um cliente, ele é aplicado em todas as entradas de cache derivadas do arquivo relevante.
- Os valores de atributo correntes são enviados “a tiracolo” (*piggybacking*) com os resultados de cada operação em um arquivo e, se o valor de $T_{m_{servidor}}$ tiver mudado, o cliente o utilizará para atualizar as entradas de cache relativas ao arquivo.
- O algoritmo adaptativo para configurar o intervalo de atualização t , mencionado anteriormente, reduz consideravelmente o tráfego para a maioria dos arquivos.

O procedimento de validação não garante o mesmo nível de consistência de arquivos que é fornecido nos sistemas UNIX convencionais, pois as atualizações recentes nem sempre são visíveis aos clientes que estão compartilhando um arquivo. Existem duas fontes de atraso temporais: o atraso após uma escrita e antes que os dados atualizados sejam removidos da cache do cliente, e a janela de 3 segundos para a validação da cache. Felizmente, a maioria dos aplicativos UNIX não depende da sincronização das atualizações de arquivo, e poucas dificuldades foram relatadas por causa dessa fonte.

As escritas são manipuladas de forma diferente. Quando um bloco armazenado em cache é modificado, ele é marcado como alterado (*dirty*) e programado para ser transferido para

o servidor de forma assíncrona. Os blocos modificados são transferidos quando o arquivo é fechado, quando ocorre uma operação *sync* no cliente, ou ainda, mais frequentemente, se processos *bio-daemon* estiverem em uso (veja a seguir). Isso não proporciona a mesma garantia de persistência que a cache do servidor, mas simula o comportamento das escritas locais.

Para implementar leitura antecipada (*read-ahead*) e escrita postergada (*write-delayed*), o cliente NFS precisa realizar algumas leituras e escritas de forma assíncrona. Isso é obtido nas implementações UNIX do NFS por meio da inclusão de um ou mais processos *bio-daemon* em cada cliente. (*Bio* significa entrada e saída de bloco [*block input-output*]; o termo *daemon* é frequentemente usado para referenciar processos em nível de usuário que executam tarefas de sistema.) A função dos processos *bio-daemon* é realizar operações de leitura antecipada e escrita postergada. Após cada requisição de leitura, o *bio-deamon* é notificado e, então, ele inicia a transferência, do servidor para a cache cliente, do bloco de arquivo seguinte ao que foi lido. No caso de escrita, o *bio-daemon* envia um bloco para o servidor sempre que o bloco tiver sido completamente escrito pelo cliente. Os blocos de diretório são enviados quando ocorre uma modificação.

Os processos *bio-daemon* melhoraram o desempenho do sistema garantindo que o módulo cliente não fique bloqueado esperando pelo retorno de operações *read*, nem que operações *write* sejam efetivadas no servidor. Eles não são um requisito lógico obrigatório, pois na ausência de um mecanismo de leitura antecipada, uma operação *read* feita por um processo usuário, provocará uma requisição síncrona para o servidor; e os resultados das operações *write* serão transferidos para o servidor quando o arquivo for fechado ou quando o sistema de arquivos virtual no cliente executar uma operação *sync*.

Outras otimizações • O sistema de arquivos da Sun é baseado no *Fast File System* do BSD UNIX, que usa blocos de disco de 8 Kbytes, resultando em menos chamadas de sistema para acessar arquivos sequenciais do que os sistemas UNIX anteriores. Os datagramas UDP usados para a implementação da RPC Sun são ampliados para 9 Kbytes, permitindo que uma chamada de RPC, contendo um bloco inteiro como argumento, seja transferida em um único datagrama, minimizando o efeito da latência da rede na leitura sequencial de arquivos. No NFS versão 3, não há limite para o tamanho máximo de blocos de arquivo que podem ser manipulados em operações *read* e *write*; clientes e servidores podem negociar tamanhos maiores do que 8 Kbytes, se ambos puderem manipulá-los.

Conforme mencionado anteriormente, as informações de status do arquivo colocadas na cache dos clientes devem ser atualizadas pelo menos a cada três segundos para arquivos ativos. Para reduzir a consequente carga no servidor, resultante das requisições *getattr*, todas as operações que se referem a arquivos, ou diretórios, são consideradas requisições *getattr* implícitas, e os valores de atributo correntes são levados “a tiracolo”, junto aos outros resultados da operação.

Tornando o NFS seguro com o Kerberos • Na Seção 11.6.2, descrevemos o sistema de autenticação Kerberos, desenvolvido no MIT, que se transformou em um padrão para tornar servidores de intranet seguros contra acessos não autorizados e ataques de impostores. A segurança das implementações do NFS foi melhorada com o uso do esquema Kerberos para autenticar clientes. Nesta subseção, descreveremos a “kerberização” do NFS, conforme especificado pelos projetistas do Kerberos.

Na implementação padrão original do NFS, a identidade do usuário é incluída em cada requisição, na forma de um identificador numérico “em claro”. (Nas versões posteriores do NFS, o identificador podia ser cifrado.) O NFS não faz nada para verificar a autenticidade do identificador fornecido. Isso implica, por parte do NFS, um alto grau de confiança na integridade do computador cliente e em seu *software*; enquanto o objetivo

do Kerberos, e de outros sistemas de segurança baseados em autenticação, é reduzir a um mínimo a gama de componentes nos quais a confiança é pressuposta. Basicamente, quando o NFS é usado em um ambiente “kerberizado”, ele só deve aceitar requisições de clientes que possam mostrar que sua identidade foi autenticada pelo Kerberos.

Uma solução óbvia, considerada pelos desenvolvedores do Kerberos, foi alterar a natureza das credenciais exigidas pelo NFS para que fossem um tíquete e um autenticador completos do Kerberos. Contudo, como NFS é implementado como um servidor sem estado, cada requisição de acesso a arquivo é tratada isoladamente e, por consequência, os dados de autenticação devem ser incluídos em cada uma delas. Isso foi considerado inaceitavelmente dispendioso quanto ao tempo exigido para realizar os procedimentos de cifragem necessários e, além disso, acarretaria a adição da biblioteca de cliente Kerberos no núcleo de cada uma das estações de trabalho.

Em vez disso, foi adotada uma estratégia mista, na qual cada servidor NFS, ao montar os sistemas de arquivos de usuários e raiz, recebe os dados de autenticação completos do Kerberos. Os resultados dessa autenticação, incluindo o identificador numérico do usuário (*uid*) e o endereço do computador cliente, são mantidos pelo servidor com as informações de montagem para cada sistema de arquivos. (Embora o servidor NFS não mantenha o estado relacionado aos processos clientes individuais, ele mantém as montagens correntes em cada computador cliente).

Em cada requisição de acesso a arquivo, o servidor NFS verifica o identificador do usuário e o endereço do remetente, e só garante o acesso se eles corresponderem àqueles armazenados no servidor para o cliente relevante, no momento da montagem. Essa estratégia mista envolve apenas um custo adicional mínimo e é segura contra a maioria das formas de ataque, desde que apenas um usuário por vez possa se conectar em cada computador cliente. No MIT, o sistema é configurado de modo que esse seja o caso. As implementações de NFS recentes incluem autenticação do Kerberos como uma de várias opções de autenticação, e sites que também executam servidores Kerberos são aconselhados a utilizar essa opção.

Desempenho • Os primeiros resultados relatados por Sandberg [1987] mostraram que o uso de NFS, normalmente, não impunha uma penalidade sobre o desempenho em comparação com o acesso aos arquivos armazenados em discos locais. Ele identificou duas áreas problemáticas remanescentes:

- uso frequente da chamada de *getattr* para buscar carimbos de tempo nos servidores para validação da cache;
- desempenho relativamente deficiente da operação *write*, porque era usada escrita direta (*write-through*) no servidor.

Ele observou que as escritas são relativamente raras na carga de trabalho normal do UNIX (cerca de 5% de todas as chamadas para o servidor) e, portanto, o custo da escrita direta é tolerável, exceto quando arquivos grandes são gravados no servidor. A versão do NFS testada não incluía o mecanismo *commit* mencionado anteriormente, e isso resultou em uma melhoria substancial no desempenho de escrita nas versões atuais. Seus resultados também mostram que a operação *lookup* é responsável por quase 50% das chamadas ao servidor. Isso é uma consequência do método de tradução de nomes de caminho, passo a passo, necessário à semântica de atribuição de nomes de arquivos do UNIX.

Regularmente, medidas são realizadas pela Sun, e por outros implementadores de NFS, usando uma versão atualizada de um conjunto exaustivo de programas de *benchmark*, conhecido como LADDIS [Keith e Wittle 1993]. Resultados atuais e antigos estão disponíveis no endereço [www.spec.org]. Lá, está resumido o desempenho para implementações do servidor NFS de muitos fornecedores e de diferentes configurações de *hardware*. Implementações

com uma só CPU, baseadas em *hardware* de PC, mas com sistemas operacionais dedicados, atingem desempenho de mais de 12.000 operações por segundo no servidor, e configurações grandes, com multiprocessadores, vários discos e controladoras, atingirem cerca de até 300.000 operações por segundo no servidor. Esses valores indicam que o NFS oferece uma solução muito eficiente para as necessidades de armazenamento distribuído em intranets em diferentes tipos de uso, variando, por exemplo, desde uma carga de atividades em desenvolvimento, em um UNIX tradicional, executada por várias centenas de engenheiros de *software*, até um conjunto de servidores Web recuperando páginas de um único servidor NFS.

Resumo do NFS • O NFS segue de perto nosso modelo abstrato. O projeto resultante proporciona boa transparência de localização e de acesso, caso o serviço de montagem NFS seja usado corretamente para produzir espaços de nomes semelhantes em todos os clientes. O NFS suporta *hardware* e sistemas operacionais heterogêneos. A implementação do servidor NFS é sem estado (*stateless*), permitindo que clientes e servidores retomem a execução após uma falha, sem a necessidade de quaisquer procedimentos de recuperação. A migração de arquivos, ou de sistemas de arquivos, não é suportada, a não ser que seja feita por intervenção manual para reconfigurar diretivas de montagem, após a mudança de um sistema de arquivos para um novo local.

O desempenho do NFS melhora com o uso de cache para blocos de arquivo em cada computador cliente. Isso é importante para a obtenção de um desempenho satisfatório, mas resulta em certo desvio da semântica estrita de atualização de arquivo com cópia única do UNIX.

Os outros objetivos de projeto do NFS, e até que ponto eles foram atingidos, serão discutidos a seguir.

Transparência de acesso: o módulo cliente NFS fornece uma interface de programação de aplicativos para processos locais idêntica à interface do sistema operacional local. Assim, em um cliente UNIX, os acessos a arquivos remotos são realizados usando-se as chamadas de sistema UNIX normais. Nenhuma modificação é exigida nos programas existentes, para permitir que eles funcionem corretamente com arquivos remotos.

Transparência de localização: cada cliente estabelece um espaço de nomes de arquivo, adicionando diretórios montados em sistemas de arquivos remotos em seu espaço de nomes local. Os sistemas de arquivo precisam ser *exportados* pelo computador que os contém e *montados de forma remota* por um cliente, antes que possam ser acessados pelos processos em execução no cliente (veja a Figura 12.10). O ponto na hierarquia de nomes de um cliente em que um sistema de arquivos montado aparece é determinado pelo cliente; assim, o NFS não impõe um único espaço de nomes de arquivo em nível de rede – cada cliente vê um conjunto de sistemas de arquivos remotos de acordo com o ponto de montagem usado. Como consequência, sistemas de arquivos remotos podem ter diferentes nomes de caminho em diferentes clientes, mas um espaço de nomes uniforme pode ser estabelecido com tabelas de configuração apropriadas em cada cliente, atingindo o objetivo da transparência de localização.

Transparência de mobilidade: os sistemas de arquivos (no sentido do UNIX, isto é, subárvores de arquivos) podem ser movidos entre servidores, mas as tabelas de montagem em cada cliente devem ser atualizadas separadamente para permitir que os clientes acessem o sistema de arquivos em sua nova localização; portanto, a transparência da migração não é totalmente obtida pelo NFS.

Escalabilidade: os valores de desempenho publicados mostram que servidores NFS podem ser construídos para manipular cargas reais muito grandes de maneira eficiente e econômica. O desempenho de um único servidor pode ser aumentado

com a adição de processadores, discos e controladoras. Quando os limites desse processo são atingidos, servidores adicionais devem ser instalados, e os sistemas de arquivos devem ser realocados entre eles. A eficácia dessa estratégia é limitada pela existência de arquivos “concorridos” – aqueles que são acessados com tanta frequência que o servidor atinge um limite de desempenho. Quando a carga ultrapassa o desempenho máximo disponível com essa estratégia, uma solução melhor pode ser oferecida por um sistema de arquivos distribuído que suporte a replicação de arquivos (como o Coda, descrito no Capítulo 18), ou por um sistema de arquivos como o AFS, que reduz o tráfego do protocolo colocando arquivos inteiros na cache. Vamos discutir outras estratégias para atingir escalabilidade na Seção 12.5.

Replicação de arquivos: Meios de armazenamentos de arquivos somente para leitura (*read-only*) podem ser replicados em vários servidores NFS, mas o NFS não suporta replicação de arquivos com atualizações. O NIS (Network Information Service), da Sun, é um serviço separado, disponível para uso com o NFS, que suporta a replicação de bancos de dados simples organizados como pares chave-valor (por exemplo, os arquivos de sistema do UNIX */etc/passwd* e */etc/hosts*). Ele gerencia a distribuição de atualizações e acessos a arquivos replicados com base em um modelo de replicação mestre–escravo simples (também conhecido como modelo de *cópia primária*, melhor discutido no Capítulo 18), com possibilidade de replicação parcial ou total do banco de dados em cada *site*. O NIS fornece um repositório compartilhado de informações de sistema que muda com pouca frequência e não exige que as atualizações ocorram simultaneamente em todos os *sites*.

Heterogeneidade de hardware e de sistema operacional: o NFS foi implementado por quase todos os sistemas operacionais e plataformas de *hardware* disponíveis e é suportado por uma variedade de sistemas de arquivos.

Tolerância a falhas: as características sem estado e idempotente do protocolo de acesso NFS garantem que os modos de falha observados pelos clientes ao acessar arquivos remotos sejam semelhantes àqueles encontrados no acesso de arquivos locais. Quando um servidor falha, o serviço que ele fornece é suspenso até que o servidor seja reiniciado, mas uma vez que ele tenha sido reiniciado, os processos clientes em nível de usuário prosseguem a partir do ponto em que o serviço foi interrompido, sem conhecimento da falha (exceto no caso de acesso a sistemas de arquivos remotos *montados condicionalmente*). Na prática, a montagem incondicional é usada na maioria dos casos, e isso tende a impedir que os programas aplicativos tratem das falhas do servidor.

A falha de um computador cliente, ou de um processo usuário, não tem nenhum efeito sobre qualquer servidor que ele possa usar, pois os servidores não contêm nenhum estado para seus clientes.

Consistência: descrevemos o comportamento da atualização com alguns detalhes. Ele apresenta uma forte aproximação com a semântica de cópia única e atende às necessidades da maioria dos aplicativos; porém, o uso de compartilhamento de arquivos por meio do NFS para comunicação ou forte coordenação entre processos em diferentes computadores não é recomendado.

Segurança: a necessidade de segurança no NFS surgiu com a conexão de muitas intranets com a Internet. A integração do Kerberos com o NFS foi um passo importante nesse sentido. Outros desenvolvimentos recentes incluem a opção de usar uma implementação de RPC segura (RPC SEC_GSS, documentada no RFC 2203 [Eisler *et al.* 1997]) para autenticação, privacidade e segurança dos dados trans-

mitidos com operações de leitura e escrita. Existem muitas instalações que não implantam esses mecanismos e, portanto, são inseguras.

Eficiência: o desempenho medido de várias implementações de NFS, e sua adoção para uso em situações que geram cargas de trabalho muito pesadas, são indicações claras da eficiência com que o protocolo NFS pode ser implementado.

12.4 Estudo de caso: Andrew File System

Assim como o NFS, o AFS (Andrew File System) fornece acesso transparente a arquivos remotos compartilhados para programas UNIX executando em estações de trabalho. O acesso a arquivos AFS se dá por intermédio das primitivas de arquivo UNIX normais, permitindo que os programas UNIX existentes acessem arquivos AFS sem modificação, nem recompilação. O AFS é compatível com o NFS. Os servidores AFS contêm arquivos UNIX “locais”, mas o sistema de arquivos nos servidores é baseado no NFS; portanto, os arquivos são referenciados, em vez de por *i-nodes*, por manipuladores de arquivo do estilo NFS permitindo serem acessados de forma remota pelo NFS.

O AFS difere do NFS em seu projeto e implementação. As diferenças são atribuídas principalmente à identificação da escalabilidade como o objetivo de projeto mais importante. O AFS é feito para funcionar bem com números maiores de usuários ativos do que outros sistemas de arquivos distribuídos. A principal estratégia para se obter a escalabilidade é o uso de cache capazes de armazenar arquivos inteiros nos clientes. O AFS tem duas características de projeto incomuns:

Servir arquivos inteiros: o conteúdo inteiro de diretórios e de arquivos é transmitido para os computadores clientes pelos servidores AFS (no AFS-3, arquivos maiores do que 64 Kbytes são transferidos em porções de 64 Kbytes).

Cache de arquivos inteiros: quando uma cópia de um arquivo, ou uma porção, tiver sido transferida para um computador cliente, ela é armazenada em uma cache no disco local. A cache contém centenas dos arquivos usados mais recentemente nesse computador. Por ser em disco, a cache é permanente, sobrevivendo às reinitializações do computador cliente. Quando possível, são usadas cópias locais dos arquivos para atender às requisições *open* dos clientes, em detrimento das cópias remotas.

Cenário • Aqui está um cenário simples ilustrando o funcionamento do AFS:

1. Quando um processo de usuário em um computador cliente executa uma chamada de sistema *open* para um arquivo no espaço de arquivo compartilhado, e não existe uma cópia corrente do arquivo na cache local, o servidor que contém o arquivo é localizado e é enviada uma requisição de cópia do arquivo.
2. A cópia é armazenada no sistema de arquivos local do UNIX, no computador cliente; então, a cópia é aberta (com *open*) e o descritor de arquivo UNIX resultante é retornado para o cliente.
3. As operações *read*, *write* e outras, subsequentes no arquivo, executadas por processos no computador cliente, são realizadas na cópia local.
4. Quando o processo no cliente executa uma chamada de sistema *close*, se a cópia local tiver sido atualizada, seu conteúdo é enviado de volta para o servidor. O servidor atualiza o conteúdo do arquivo e os carimbos de tempo associados. A cópia que está no disco local do cliente é mantida, para o caso de ser novamente necessária, para um processo em nível de usuário na mesma estação de trabalho.

A seguir, vamos discutir o desempenho observado do AFS, mas podemos fazer aqui algumas observações e previsões gerais com base nas características de projeto descritas anteriormente:

- Para arquivos compartilhados que são raramente modificados (como os que contêm o código de comandos e bibliotecas do UNIX) e para arquivos que normalmente são acessados por apenas um usuário (como a maioria dos arquivos do diretório de base de um usuário e sua subárvore), é provável que as cópias armazenadas localmente na cache permaneçam válidas por longos períodos de tempo – no primeiro caso, porque eles não são atualizados, e no segundo, porque, se eles forem atualizados, a cópia atualizada estará na cache da estação de trabalho do proprietário. Essas classes de arquivo são responsáveis pela maioria dos acessos a arquivos.
- A cache local pode usar uma proporção substancial do espaço em disco de cada estação de trabalho, digamos, 100 megabytes. Normalmente, isso é suficiente para o estabelecimento de um conjunto de trabalho dos arquivos acessados por um único usuário. A provisão de armazenamento em cache suficiente para o estabelecimento de um conjunto de trabalho garante que os arquivos regularmente usados em uma determinada estação de trabalho sejam normalmente mantidos na cache até que sejam novamente necessários.
- A estratégia de projeto é baseada em algumas suposições sobre o tamanho de arquivo médio e máximo e sobre o caráter local da referência a arquivos nos sistemas UNIX. Essas suposições são derivadas de observações de cargas de trabalho típicas do UNIX em ambientes acadêmicos e outros [Satyanarayanan 1981; Ousterhout *et al.* 1985; Floyd 1986]. As observações mais importantes são:
 - Os arquivos são pequenos; a maioria tem menos de 10 Kbytes de tamanho.
 - As operações de leitura nos arquivos são muito mais comuns do que as de escrita (cerca de seis vezes mais comuns).
 - Acesso sequencial é comum, acesso aleatório é raro.
 - A maioria dos arquivos é lida e escrita por apenas um usuário. Quando um arquivo é compartilhado, normalmente é apenas um usuário que o modifica.
 - Os arquivos são referenciados em momentos específicos. Se um arquivo tiver sido referenciado recentemente, existe uma grande probabilidade de que ele seja referenciado novamente em um futuro próximo.

Essas observações foram usadas para delinear o projeto e a otimização do AFS e *não* para restringir a funcionalidade vista pelos usuários.

- O AFS funciona melhor com as classes de arquivo identificadas no primeiro ponto anterior. Há um tipo de arquivo importante que não se encaixa em nenhuma dessas classes – os bancos de dados são normalmente compartilhados por muitos usuários e, muitas vezes, são atualizados com bastante frequência. Os projetistas do AFS excluíram explicitamente de seus objetivos de projeto o suporte para bancos de dados, dizendo que as restrições impostas por diferentes estruturas de atribuição de nomes (isto é, acesso baseado no conteúdo), a necessidade de granularidade fina para acesso aos dados, o controle de concorrência e a atomicidade das atualizações dificultam o projeto de um sistema de banco de dados distribuído que também seja um sistema de arquivos distribuído. Eles argumentam que os requisitos necessários para bancos de dados distribuídos devem ser tratados separadamente [Satyanarayanan 1989a].

12.4.1 Implementação

O cenário anterior ilustra o funcionamento do AFS, mas deixa sem resposta muitas perguntas sobre sua implementação. Dentre as mais importantes estão:

- Como o AFS obtém o controle quando uma chamada de sistema *open* (ou *close*) se referindo a um arquivo no espaço de arquivos compartilhado é feita por um cliente?
- Como é localizado o servidor que contém o arquivo solicitado?
- Que espaço em cache é alocado para arquivos nas estações de trabalho?
- Como o AFS garante que as cópias dos arquivos colocadas na cache estão atualizadas quando os arquivos podem ser modificados por vários clientes?

Responderemos a essas perguntas a seguir.

O AFS é implementado com base em dois componentes de *software* que existem como processos UNIX, em nível de usuário, chamados *Vice* e *Venus*. A Figura 12.11 mostra a distribuição dos processos Vice e Venus. Vice é o nome dado ao *software* servidor executado em cada computador servidor, e Venus é um processo executado em cada computador cliente e corresponde ao módulo cliente em nosso modelo abstrato.

Os arquivos disponíveis para processos de usuário em execução nas estações de trabalho são *locais* ou *compartilhados*. Os arquivos locais são tratados como arquivos normais do UNIX. Eles são armazenados em um disco da estação de trabalho e estão disponíveis somente para processos de usuário locais. Os arquivos compartilhados são armazenados em servidores, e cópias deles são armazenadas na cache nos discos locais das estações de trabalho. O espaço de nomes visto pelos processos de usuário está ilustrado na Figura 12.12. Trata-se de uma hierarquia de diretórios UNIX convencional, com uma subárvore específica (chamada *cmu*), contendo todos os arquivos compartilhados. Essa divisão do espaço de nomes de arquivo em arquivos locais e compartilhados leva a certa perda de transparência de localização, mas isso dificilmente é perceptível para usuários que não sejam os administradores de sistema. Os arquivos locais são usados apenas por arquivos temporários (*/tmp*) e por processos que são fundamentais para a inicialização da estação de trabalho. Outros arquivos padrão do UNIX (como aqueles normalmente encontrados em */bin*, */lib*, etc.) são

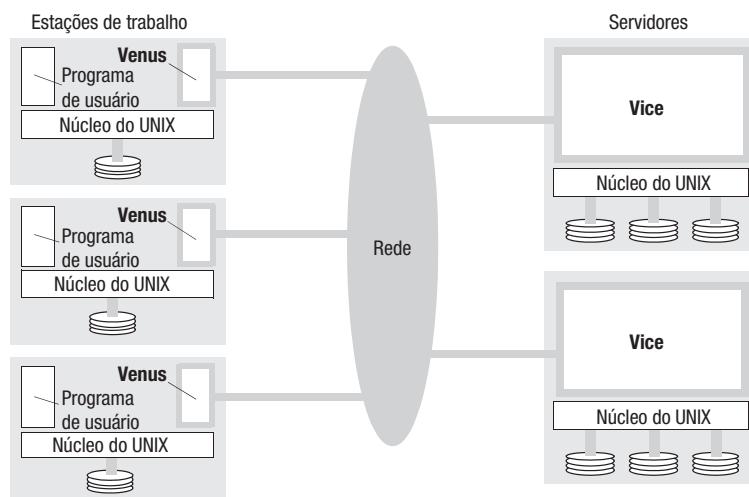


Figura 12.11 Distribuição de processos no Andrew File System.

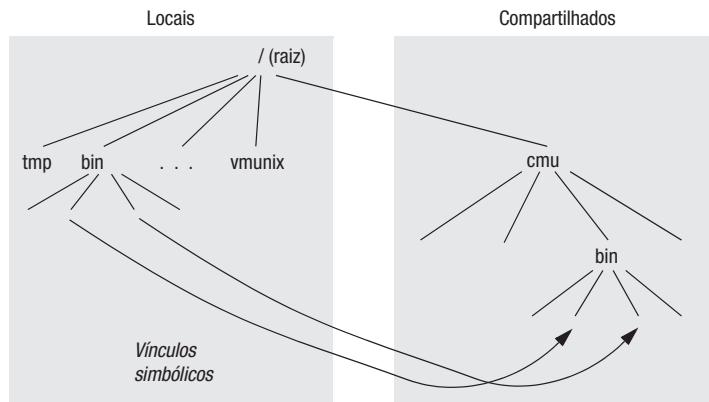


Figura 12.12 Espaço de nomes de arquivo visto pelos clientes do AFS.

implementados como vínculos simbólicos de diretórios locais para arquivos mantidos no espaço compartilhado. Os diretórios dos usuários ficam no espaço compartilhado, permitindo que os usuários acessem seus arquivos a partir de qualquer estação de trabalho.

O núcleo UNIX, em cada estação de trabalho e servidor, é uma versão modificada do UNIX BSD. As modificações são feitas para interceptar chamadas de sistema de arquivos *open*, *close* e algumas outras, quando elas se referem a arquivos do espaço de nomes compartilhado e as passam para o processo Venus no computador cliente (ilustrado na Figura 12.13). Outra modificação do núcleo é incluída por motivos de desempenho e será descrita posteriormente.

Uma partição do disco local de cada estação de trabalho é usada como a cache e conterá as cópias dos arquivos do espaço compartilhado. Venus gerencia a cache. Se, ao obter um novo arquivo a partir do servidor, a partição de cache estiver cheia, Venus remove os arquivos usados menos recentemente para abrir espaço. Normalmente, a cache da estação de trabalho é grande o suficiente para acomodar várias centenas de arquivos de tamanho médio. Isso torna a estação de trabalho amplamente independente dos servidores Vice, uma vez que um conjunto dos arquivos de trabalho do usuário corrente e dos arquivos de sistema usados frequentemente foram colocados na cache.

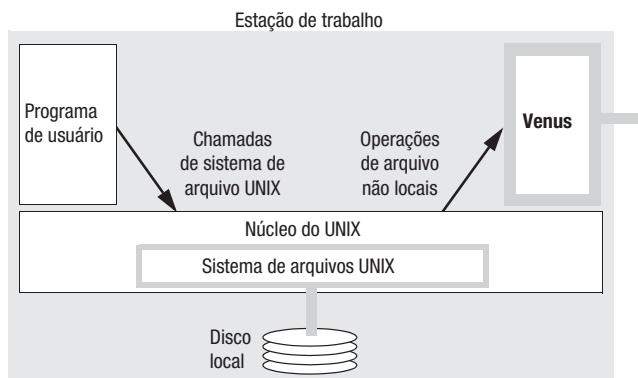


Figura 12.13 Interceptação de chamada de sistema no AFS.

O AFS é semelhante ao modelo de serviço de arquivos abstrato, descrito na Seção 12.2, sob os seguintes aspectos:

- Um serviço de arquivos simples é implementado pelos servidores Vice, e a estrutura de diretório hierárquica exigida pelos programas de usuário do UNIX é implementada pelo conjunto de processos Venus nas estações de trabalho.
- Cada arquivo e diretório no espaço de arquivo compartilhado é identificado por um identificador de arquivo (*fid – file identifier*) único de 96 bits, semelhante a um UFID. Os processos Venus traduzem os nomes de caminho, fornecidos pelos clientes, em *fids*.

Os arquivos são agrupados em *volumes* para facilitar a localização e a movimentação. Geralmente, os volumes são menores do que os sistemas de arquivos UNIX, que são a unidade de agrupamento de arquivos do NFS. Por exemplo, os arquivos pessoais de cada usuário geralmente estão localizados em um volume separado. Outros volumes são alocações para binários do sistema, documentação e códigos de bibliotecas.

A representação dos *fids* inclui o número do volume que contém o arquivo (compare com o *identificador de grupo de arquivos* nos UFIDs), um manipulador de arquivo NFS identificando o arquivo dentro do volume (conforme o *número de arquivo* nos UFIDs) e um *elemento de exclusividade (uniquifier)* para garantir que os identificadores de arquivo não sejam reutilizados:

32 bits	32 bits	32 bits
Número do volume	Manipulador de arquivo	Elemento de exclusividade

Os programas de usuário utilizam nomes de caminho convencionais do UNIX para referenciar arquivos, mas o AFS usa *fids* na comunicação entre os processos Venus e Vice. Os servidores Vice aceitam requisições apenas em termos de *fids*. O Venus traduz os nomes de caminho fornecidos pelos clientes em *fids*, usando uma pesquisa passo a passo para obter as informações dos diretórios de arquivo mantidos nos servidores Vice.

A Figura 12.14 descreve as ações executadas pelo Vice, pelo Venus e pelo núcleo do UNIX, quando um processo de usuário executa cada uma das chamadas de sistema mencionadas em nosso cenário delineado anteriormente. A *promessa de callback*, mencionada na figura, é um mecanismo para garantir que as cópias dos arquivos armazenadas em cache sejam atualizadas quando outro cliente fechar o mesmo arquivo após modificá-lo. Esse mecanismo será discutido na próxima seção.

12.4.2 Consistência da cache

Quando o Vice fornece uma cópia de um arquivo para um processo Venus, ele também fornece uma *promessa de callback* – um tipo de tiquete (*token*) emitido pelo servidor Vice que é o depositório do arquivo, garantindo que notificará o processo Venus quando qualquer outro cliente modificar o arquivo. As promessas de *callback* são armazenadas junto aos arquivos na cache em disco da estação de trabalho e têm dois estados: *válida* ou *cancelada*. Quando um servidor realiza uma requisição de atualização de um arquivo, ele notifica todos os processos Venus para os quais tem promessas de *callback* emitidas, enviando um *callback* para cada um – um *callback* é uma chamada de procedimento remoto feita por um servidor para um processo Venus. Quando o processo Venus recebe um *callback*, ele configura o *token de promessa de callback* do arquivo em questão como *cancelado*.

Quando Venus trata uma operação *open* em nome de um cliente, ele verifica a cache. Se o arquivo solicitado for encontrado na cache, seu *token* é verificado. Se o valor for *cancelado*, uma cópia nova do arquivo deve ser buscada no servidor Vice, mas se o

<i>Processo de usuário</i>	<i>Núcleo do UNIX</i>	<i>Venus</i>	<i>Rede</i>	<i>Vice</i>
<i>open(File Name, mode)</i>	Se <i>File Name</i> se refere a um arquivo no espaço de arquivos compartilhado, passa a requisição para Venus. Abre o arquivo local e retorna o descritor de arquivo para o aplicativo.	Verifica a lista de arquivos na cache local. Se não estiver presente ou não houver uma promessa de <i>callback</i> válida, envia uma requisição do arquivo para o servidor Vice que é o depositório do volume que contém o arquivo. Coloca a cópia do arquivo no sistema de arquivos local, insere seu nome local na lista da cache e retorna o nome local para o UNIX.	Transfere uma cópia do arquivo e uma <i>promessa de callback</i> para a estação de trabalho. Registra a promessa de <i>callback</i> .	
<i>read(File Descriptor, Buffer, length)</i>	Executa uma operação de leitura UNIX normal na cópia local.			
<i>write(File Descriptor, Buffer, length)</i>	Executa uma operação de escrita UNIX normal na cópia local.			
<i>close(File Descriptor)</i>	Fecha a cópia local e notifica Venus de que o arquivo foi fechado.	Se a cópia local tiver sido alterada, envia uma cópia para o servidor Vice que é o depositório do arquivo.	Substitui o conteúdo do arquivo e envia um <i>callback</i> para todos os outros clientes que contêm <i>promessas de callback</i> no arquivo.	

Figura 12.14 Implementação de chamadas de sistema de arquivos no AFS.

token for válido, então a cópia armazenada na cache poderá ser aberta e usada, sem referência ao Vice.

Quando uma estação de trabalho é reiniciada após uma falha ou desligamento, Venus tenta manter o máximo possível dos arquivos armazenados na cache no disco local, mas não pode presumir que os *tokens* de promessa de *callback* estejam corretos, pois alguns *callbacks* podem ter sido perdidos. Portanto, antes do primeiro uso de cada arquivo, ou diretório armazenados na cache após uma reinicialização, Venus gera uma requisição de validação de cache contendo o carimbo de tempo da modificação do arquivo para o servidor que é o depositório do arquivo. Se o carimbo de tempo estiver atualizado, o servidor responderá com *válido* e o *token* será reinstalado. Se o carimbo de tempo mostrar que o arquivo está desatualizado, então o servidor responderá com *cancelado* e o *token* será configurado como *cancelado*. Os *callbacks* devem ser renovados antes de uma operação *open*, caso um tempo *T* (normalmente, da ordem de alguns minutos) tenha decorrido desde que o arquivo foi posto na cache sem ter havido comunicação do servidor. Isso serve para tratar de possíveis falhas de comunicação, as quais podem resultar na perda de mensagens de *callback*.

Esse mecanismo baseado em *callbacks* para manutenção da consistência da cache foi adotado como a estratégia que oferecia maior escalabilidade, de acordo com a avaliação no protótipo (AFS-1) de um mecanismo baseado em carimbos de tempo semelhante àquele usado no NFS. No AFS-1, um processo Venus, contendo uma cópia de um arquivo na cache,

<i>Fetch(fid) → attr, data</i>	Retorna os atributos (status) e, opcionalmente, o conteúdo do arquivo identificado pelo <i>fid</i> e registra nele uma promessa de <i>callback</i> .
<i>Store(fid, attr, data)</i>	Atualiza os atributos e (opcionalmente) o conteúdo de um arquivo especificado.
<i>create() → fid</i>	Cria um novo arquivo e registra nele uma promessa de <i>callback</i> .
<i>Remove(fid)</i>	Exclui o arquivo especificado.
<i>SetLock(fid, mode)</i>	Estabelece uma trava (<i>lock</i>) no arquivo ou diretório especificado. O modo da trava pode ser compartilhado ou exclusivo. As travas que não são removidas expiram após 30 minutos.
<i>ReleaseLock(fid)</i>	Destrava o arquivo ou diretório especificado.
<i>RemoveCallback(fid)</i>	Informa o servidor que um processo Venus removeu um arquivo de sua cache.
<i>BreakCallback(fid)</i>	Esta chamada é feita por um servidor Vice para um processo Venus. Ela cancela a promessa de <i>callback</i> no arquivo identificado por <i>fid</i> .

Nota: As operações de diretório e administrativas (Rename, Link, Makedir, Removedir, GetTime, CheckToken, etc.) não são mostradas.

Figura 12.15 Os principais componentes da interface do serviço Vice.

interroga o processo Vice em cada operação *open* para determinar se o carimbo de tempo na cópia local está de acordo com a do servidor. A estratégia baseada em *callback* tem maior escalabilidade porque resulta na comunicação entre cliente e servidor e na atividade do servidor somente quando o arquivo tiver sido atualizado, enquanto a estratégia de carimbo de tempo causa uma interação cliente-servidor em cada operação *open*, mesmo quando existe uma cópia local válida. Como a maioria dos arquivos não é acessada concorrentemente, e as operações *read* predominam sobre as operações *write* na maioria dos aplicativos, o mecanismo de *callback* causa uma importante redução no número de interações cliente-servidor.

O mecanismo de *callback* usado no AFS-2, e em versões posteriores do AFS, exige que os servidores Vice mantenham algum estado em nome de seus clientes Venus, ao contrário do que ocorre no AFS-1, no NFS e em nosso modelo de serviço de arquivos. O estado depende do cliente e consiste em uma lista dos processos Venus para os quais foram emitidas promessas de *callback* para cada arquivo. Essas listas de *callback* devem ser mantidas no caso de falhas do servidor – elas são mantidas nos discos do servidor e atualizadas com operações atômicas.

A Figura 12.15 mostra as chamadas de RPC fornecidas pelos servidores AFS para operações em arquivos (isto é, a interface fornecida pelos servidores AFS para processos Venus).

Semântica de atualização • O objetivo desse mecanismo de consistência de cache é obter a melhor aproximação possível da semântica de arquivo de cópia única, sem uma degradação séria de desempenho. Uma implementação estrita da semântica de cópia única, para primitivas de acesso a arquivo do UNIX, exigiria que os resultados de cada operação *write* em um arquivo fossem distribuídos para todos os *sites* que o contêm em suas caches, antes que mais acessos pudessem ocorrer. Isso não pode ser feito em sistemas de larga escala; em vez disso, o mecanismo de promessa de *callback* mantém uma aproximação da semântica de cópia única.

Para o AFS-1, a semântica de atualização pode ser declarada formalmente em termos muito simples. Para um cliente C operando em um arquivo F , cujo depositório é um servidor S , as seguintes garantias de atualização das cópias de F são mantidas:

após uma operação *open* bem-sucedida: $\text{latest}(F, S)$

após uma operação *open* malsucedida: $\text{failure}(S)$

após uma operação *close* bem-sucedida: $\text{updated}(F, S)$

após uma operação *close* malsucedida: $\text{failure}(S)$

onde $\text{latest}(F, S)$ denota uma garantia de que o valor corrente de F , em C , é igual ao valor em S , $\text{failure}(S)$ denota que a operação *open* (ou *close*) não foi efetuada em S (e a falha pode ser detectada por C) e $\text{updated}(F, S)$ denota que o valor de C de F foi propagado com êxito para S .

Para o AFS-2, a garantia de atualização de *open* é ligeiramente mais fraca, e a declaração formal da garantia correspondente é mais complexa. Isso porque um cliente pode abrir uma cópia antiga de um arquivo após ele ter sido atualizado por outro cliente. Isso ocorre se uma mensagem de *callback* é perdida, por exemplo, como resultado de uma falha da rede. No entanto, existe um tempo máximo T durante o qual um cliente pode permanecer sem saber da existência de uma versão mais recente de um arquivo. Portanto, temos a seguinte garantia:

após uma operação *open* bem-sucedida: $\text{latest}(F, S, 0)$
ou ($\text{lostCallback}(S, T)$ e $\text{inCache}(F)$ e
 $\text{latest}(F, S, T)$)

onde $\text{latest}(F, S, T)$ denota que a cópia de F vista pelo cliente não está há mais do que T segundos desatualizada, $\text{lostCallback}(S, T)$ denota que uma mensagem de *callback* de S para C foi perdida em algum momento durante os últimos T segundos e $\text{inCache}(F)$, que o arquivo F estava na cache em C antes que a operação *open* fosse tentada. A declaração formal anterior expressa o fato de que a cópia colocada na cache de F em C , após uma operação *open*, é a versão mais recente presente no sistema, ou que uma mensagem de *callback* foi perdida (devido a uma falha de comunicação) e a versão que já estava na cache foi utilizada. A versão colocada na cache não estará há mais do que T segundos desatualizada (T é uma constante do sistema representando o intervalo em que as promessas de *callback* devem ser renovadas. Na maioria das instalações, o valor de T é configurado em cerca de 10 minutos.)

De acordo com seu objetivo – fornecer um serviço de arquivo distribuído de larga escala, compatível com o UNIX –, o AFS não fornece nenhuma mecanismo para o controle de atualizações concorrentes. O algoritmo de consistência de cache descrito anteriormente entra em ação somente nas operações *open* e *close*. Uma vez que um arquivo tenha sido aberto, o cliente pode acessar e atualizar a cópia local da maneira que quiser, sem o conhecimento de quaisquer processos em outras estações de trabalho. Quando o arquivo é fechado, uma cópia é retornada para o servidor substituindo a versão corrente.

Se clientes, em diferentes estações de trabalho, abrem, escrevem e fecham (com as operações *open*, *write* e *close*) o mesmo arquivo concorrentemente, todas as atualizações, menos a resultante da última operação *close*, serão perdidas silenciosamente (nenhum relatório de erro é fornecido). Os clientes devem implementar o controle de concorrência independentemente, caso exijam isso. Por outro lado, quando dois processos clientes na mesma estação de trabalho abrem um arquivo, eles compartilham a mesma cópia colocada na cache, e as atualizações são realizadas da maneira normal do UNIX – bloco por bloco.

Embora a semântica de atualização seja diferente, dependendo das localizações dos processos concorrentes que estão acessando um arquivo, e não seja precisamente igual àquela fornecida pelo sistema de arquivos padrão do UNIX, elas são suficientemente parecidas para que a maioria dos programas UNIX existentes funcione corretamente.

12.4.3 Outros aspectos

Modificações no núcleo do UNIX • Notamos que o servidor Vice é um processo usuário executando no computador servidor e o servidor é dedicado ao estabelecimento de um serviço AFS. O núcleo do UNIX nos computadores que usam AFS é alterado para que o Vice possa efetuar operações de arquivo em termos de manipuladores de arquivo, em vez dos descritores de arquivo convencionais do UNIX. Essa é a única modificação no núcleo exigida pelo AFS e é necessária se o Vice não precisar manter o estado do cliente (como os descritores de arquivo).

Banco de dados de localização • Cada servidor contém uma cópia de um banco de dados de localização totalmente replicado, fornecendo um mapeamento de nomes de volume para servidores. Quando um volume é movido, podem ocorrer imprecisões temporárias nesse banco de dados, mas elas são inofensivas, pois o encaminhamento de informações é deixado para trás, no servidor a partir do qual o volume é movido.

Threads • As implementações de Vice e Venus utilizam uma biblioteca de *threads* não preemptiva para permitir que as requisições sejam processadas concorrentemente no cliente (em que vários processos de usuário podem ter requisições de acesso a arquivo em andamento concorrentemente) e no servidor. No cliente, as tabelas que descrevem o conteúdo da cache e o banco de dados de volumes são mantidos na memória compartilhada entre as *threads* do Venus.

Rélicas somente de leitura • Os volumes que contêm arquivos lidos frequentemente, mas raramente modificados, como os diretórios de comandos de sistema UNIX */bin* e */usr/bin* e o diretório de páginas de manual */man*, podem ser replicados como volumes somente de leitura em vários servidores. Quando isso é feito, existe apenas uma réplica para leitura e escrita, e todas as atualizações são direcionadas a ela. A propagação das alterações para as rélicas somente de leitura é realizada após a atualização por parte de um procedimento operacional explícito. As entradas no banco de dados de localização para volumes replicados dessa maneira são do tipo uma para muitas, e o servidor para cada requisição de cliente é selecionado de acordo com as cargas e a acessibilidade do servidor.

Transferências de grandes volumes • O AFS transfere arquivos entre clientes e servidores em porções de 64 Kbytes. O uso de um pacote desse tamanho é uma ajuda importante para o desempenho, minimizando o efeito da latência da rede. Assim, o projeto do AFS permite que o uso da rede seja otimizado.

Uso de cache parcial de arquivo • A necessidade de transferir o conteúdo inteiro dos arquivos para os clientes, mesmo quando o requisito do aplicativo é ler apenas uma pequena parte do arquivo, é uma fonte de ineficiência óbvia. A versão 3 do AFS eliminou esse requisito, permitindo que os dados dos arquivos sejam transferidos e colocados na cache em blocos de 64 Kbytes, enquanto ainda mantém a semântica de consistência e outras características do protocolo AFS.

Desempenho • O principal objetivo do AFS é a escalabilidade; portanto, seu desempenho com grandes números de usuários é de particular interesse. Howard *et al.* [1988] fornecem detalhes de extensivas medidas de desempenho, que foram feitas usando um *benchmark* especialmente desenvolvido para o AFS, que posteriormente foi usado para a avaliação de sistemas de arquivos distribuídos. Evidentemente, o uso de cache para o arquivo inteiro e o protocolo de *callback* levaram a cargas substancialmente menores nos servidores. Satyanarayanan [1989a] informa que foi medida uma carga de servidor de 40%, com 18 clientes executando um *benchmark* padrão, contra uma carga de 100% para o NFS executando o mesmo *benchmark*. Satyanarayanan atribui grande parte da vantagem do desempenho do AFS à redução na carga do servidor derivada do uso de *callbacks* para notificar os clientes

sobre atualizações nos arquivos, se comparada ao mecanismo de carimbos de tempo usado no NFS para verificar a validade das páginas colocadas na cache nos clientes.

Suporte remoto • A versão 3 do AFS suporta várias células administrativas, cada uma com seus próprios servidores, clientes, administradores de sistema e usuários. Cada célula é um ambiente completamente autônomo, mas uma federação de células pode cooperar para apresentar aos usuários um espaço de nomes de arquivo uniforme e transparente. O sistema resultante foi amplamente implantado pela Transarc Corporation, e foi publicado um levantamento detalhado dos padrões de utilização e desempenho resultantes [Spasojevic e Satyanarayanan 1996]. O sistema foi instalado em mais de 1.000 servidores, em mais de 150 *sites*. O levantamento mostrou taxas de utilização da cache no intervalo de 96–98% para acessos a uma amostra de 32.000 volumes de arquivo, contendo 200 Gbytes de dados.

12.5 Aprimoramentos e mais desenvolvimentos

Vários avanços foram feitos no projeto de sistemas de arquivos distribuídos desde o surgimento do NFS e do AFS. Nesta seção, descreveremos os avanços que melhoram o desempenho, a disponibilidade e a escalabilidade dos sistemas de arquivos distribuídos convencionais. Avanços mais radicais são descritos em outras partes deste livro, incluindo a manutenção da consistência em sistemas de arquivos de leitura e escrita replicados para suportar operação desconectada e alta disponibilidade, nos sistemas Bayou e Coda (Seções 18.4.2 e 18.4.3), e uma arquitetura com grande escalabilidade para a implantação de fluxos de dados em tempo real com garantia de qualidade, no servidor de arquivos de vídeo Tiger (Seção 20.6.1).

Melhorias no NFS • Vários projetos de pesquisa trataram da questão da semântica de atualização de cópia única, ampliando o protocolo NFS para incluir as operações *open* e *close* e adicionar um mecanismo de *callback* para permitir que o servidor notifique os clientes sobre a necessidade de invalidar entradas da cache. Descreveremos aqui dois desses trabalhos; seus resultados parecem indicar que esses aprimoramentos podem ser acomodados sem complexidade ou custos de comunicação extras.

Alguns trabalhos recentes da Sun, e de outros desenvolvedores de NFS, foram direcionados de forma a tornar os servidores NFS mais acessíveis e úteis em redes de longa distância. Embora o protocolo HTTP, suportado pelos servidores Web, ofereça um método eficiente e com alta escalabilidade para tornar arquivos inteiros disponíveis para os clientes por meio da Internet, ele é menos útil para programas aplicativos que exigem acesso a partes de arquivos grandes, ou para aqueles que atualizam partes de arquivos. O desenvolvimento do WebNFS (descrito a seguir) possibilita que programas aplicativos se tornem clientes de servidores NFS, em qualquer parte da Internet (usando o protocolo NFS diretamente, em vez de usar indiretamente, por meio de um módulo do núcleo). Isso, junto a bibliotecas apropriadas da linguagem Java e de outras linguagens de programação de rede, deve oferecer a possibilidade de implementar aplicativos de Internet que compartilhem dados diretamente, como jogos para vários usuários ou clientes de grandes bancos de dados dinâmicos.

Obtenção da semântica de atualização de cópia única: a arquitetura de servidor sem estado do NFS trouxe grandes vantagens na robustez e na facilidade de implementação do NFS, mas impossibilita a obtenção de uma semântica de atualização de cópia única precisa (não há garantia de que os efeitos de gravações concorrentes feitas por diferentes clientes no mesmo arquivo sejam os mesmos de um sistema UNIX único, quando vários processos gravam em um arquivo local). Ela também impede o uso de *callbacks* notificando os clientes sobre alterações em arquivos, e isso resulta em frequentes requisições de *getattr* dos clientes para verificar a existência de modificação no arquivo.

Dois sistemas de pesquisa foram desenvolvidos para tratar desses inconvenientes. O Spritely NFS [Srinivasan e Mogul 1989, Mogul 1994] é uma evolução do sistema de arquivos desenvolvido na Berkeley para o sistema operacional distribuído Sprite [Nelson *et al.* 1988]. O Spritely NFS é uma implementação do protocolo NFS com a inclusão de chamadas *open* e *close*. Os módulos dos clientes devem enviar uma operação *open* quando um processo local usuário abre um arquivo que está no servidor. Os parâmetros da operação *open* do Sprite especificam um modo (leitura, gravação ou ambos) e incluem contagens do número de processos locais que têm o arquivo aberto correntemente para leitura e para escrita. Analogamente, quando um processo local fecha um arquivo remoto, uma operação *close* é enviada para o servidor, com contagens atualizadas de leitores e escritores. O servidor registra esses números em uma *tabela de arquivos abertos*, com o endereço IP e o número de porta do cliente.

Quando o servidor recebe uma operação *open*, ele verifica a *tabela de arquivos abertos* de outros clientes que têm o mesmo arquivo aberto e envia mensagens de *callback* para esses clientes, instruindo-os a modificar suas estratégias de uso de cache. Se a operação *open* especificar modo de escrita, então ela falhará, caso qualquer outro cliente tenha o arquivo aberto para escrita. Outros clientes que tiverem o arquivo aberto para leitura serão instruídos a invalidar as partes do arquivo colocadas na cache de forma local.

Para operações *open* que especificam o modo de leitura, o servidor envia uma mensagem de *callback* para todo cliente que esteja escrevendo, dizendo a ele para parar de usar a cache (isto é, para usar um modo de operação rigorosamente de escrita direta – *write-through*), e instrui a todos os clientes que estão lendo para que parem de colocar o arquivo na cache (para que todas as chamadas de leitura locais resultem em uma requisição para o servidor).

Essas medidas resultam em um serviço de arquivo que mantém a semântica de atualização de cópia única do UNIX às custas do transporte de algum estado relacionado ao cliente no servidor. Elas também permitem alguns ganhos de eficiência no tratamento de escritas na cache. Se o estado relacionado ao cliente for mantido em memória volátil no servidor, ele será vulnerável às falhas do servidor. O Spritely NFS implementa um protocolo de recuperação que interroga uma lista de clientes que abriram arquivos recentemente no servidor, para recuperar a *tabela de arquivos abertos* inteira. A lista de clientes é armazenada no disco, sendo atualizada de forma relativamente rara, e é pessimista – ela pode incluir, com segurança, mais clientes do que aqueles que tinham arquivos abertos no momento da falha. Clientes defeituosos também podem resultar em excesso de entradas na *tabela de arquivos abertos*, mas elas serão removidas quando o cliente reiniciar.

Quando o Spritely NFS foi avaliado com relação ao NFS, versão 2, apresentou uma melhoria de desempenho modesta. Isso se deu graças ao uso melhorado da escrita em cache. Mudanças no NFS, versão 3, provavelmente resultariam pelo menos em uma melhoria igual, mas os resultados do projeto Spritely NFS certamente indicam que é possível obter semântica de atualização de cópia única sem perda de desempenho substancial, embora às custas de alguma complexidade de implementação extra nos módulos cliente e servidor e da necessidade de um mecanismo de recuperação para restaurar o estado após uma falha de servidor.

NQNFS: o projeto NQNFS (Not Quite NFS) [Macklem 1994] tinha objetivos semelhantes ao Spritely NFS – adicionar consistência de cache mais precisa ao protocolo NFS e melhorar o desempenho por intermédio do uso melhor da cache. Um servidor NQNFS mantém estado similar relacionado ao cliente, a respeito de arquivos abertos, mas utiliza arrendamentos (Seção 5.4.3) para ajudar na recuperação, após uma falha de servidor. O servidor estabelece um limite superior para o tempo durante o qual um cliente pode manter um arrendamento para um arquivo aberto. Se o cliente quiser continuar além desse tempo, precisará renovar o arrendamento. *Callbacks* são usadas de maneira semelhante ao Spritely

NFS, para pedir aos clientes que invalidem suas caches quando ocorrer uma requisição de escrita, mas se os clientes não responderem, o servidor simplesmente esperará até que seus arrendamentos expirem, antes de responder à nova requisição de escrita.

WebNFS: o advento da Web e dos *applets* Java levou ao reconhecimento, por parte da equipe de desenvolvimento do NFS e outros, de que alguns aplicativos de Internet poderiam tirar proveito do acesso direto aos servidores NFS sem grande parte das sobrecargas associadas à simulação das operações de arquivo do UNIX incluídas nos clientes NFS padrão.

O objetivo do WebNFS (descrito nas RFCs 2055 e 2056 [Callaghan 1996a, 1996b]) é permitir que navegadores Web, programas Java e outros aplicativos interajam diretamente com um servidor NFS para acessar arquivos que são publicados, usando um *manipulador de arquivo público* para acessar arquivos relativos a um diretório raiz público. Esse modo de uso anula o serviço *mount* e o serviço mapeador de porta (descritos no Capítulo 5). Os clientes WebNFS interagem com um servidor NFS em um número de porta bem conhecido (2049). Para acessar arquivos pelo nome de caminho, eles emitem requisições de *lookup* usando um manipulador de arquivo público. O manipulador de arquivo público tem um valor interpretado de forma especial pelo sistema de arquivos virtual no servidor. Devido à alta latência das redes de longa distância, é usada uma variante da operação *lookup* para pesquisar um nome de caminho de várias partes em uma única requisição.

Assim, o WebNFS permite o desenvolvimento de clientes que acessam partes de arquivos armazenados em servidores NFS de *sites* remotos com sobrecargas de configuração mínimas. Há suporte para controle de acesso e autenticação, mas, em muitos casos, o cliente solicitará apenas acesso de leitura para arquivos públicos e, nesse caso, a opção de autenticação poderá ser desativada. Ler uma parte de um único arquivo localizado em um servidor NFS que suporta WebNFS exige o estabelecimento de uma conexão TCP e duas chamadas de RPC – uma para a operação *lookup* de várias partes e outra para a operação *read*. O tamanho do bloco de dados lido não é limitado pelo protocolo NFS.

Por exemplo, um serviço de previsão de tempo poderia publicar um arquivo em seu servidor NFS, contendo um grande banco de dados de dados climáticos atualizados frequentemente, com um URL do tipo:

nfs://data.weather.gov/weatherdata/global.data

Um cliente *WeatherMap* interativo, que mostrasse mapas do clima, poderia ser construído em Java, ou em qualquer outra linguagem, que ofereça suporte a biblioteca de procedimentos WebNFS. O cliente lê apenas as partes do arquivo */weatherdata/global.data* que são necessárias para construir os mapas específicos solicitados por um usuário, enquanto um aplicativo semelhante, que usasse HTTP para acessar um servidor de dados climáticos, teria que transferir o banco de dados inteiro para o cliente, ou solicitar o suporte de um programa servidor de propósito específico para fornecer os dados solicitados.

NFS versão 4: uma nova versão do protocolo NFS foi apresentada em 2000. Os objetivos do NFS versão 4 estão descritos no RFC 2624 [Shepler 1999] e no livro de Brent Callaghan [Callaghan 1999]. Assim como o WebNFS, ela pretende facilitar o uso do NFS em redes de longa distância e em aplicativos de Internet. Ela inclui os recursos do WebNFS, mas a introdução de um novo protocolo também apresenta a oportunidade de fazer aprimoramentos mais radicais. (O WebNFS ficou restrito às alterações no servidor que não envolviam essa adição de novas operações no protocolo.)

O NFS versão 4 explora os resultados que surgiram das pesquisas feitas no projeto do servidor de arquivos na última década, como o uso de *callbacks* ou de arrendamentos para manter a consistência. O NFS versão 4 suporta recuperação dinâmica de falhas do servidor, permitindo que os sistemas de arquivos sejam movidos para novos servidores

de forma transparente. A capacidade de mudança de escala é aprimorada pelo uso de servidores *proxies* de uma maneira análoga ao seu uso na Web.

Aprimoramentos do AFS • Mencionamos que o DCE/DFS, o sistema de arquivos distribuído incluído no *Distributed Computing Environment*, da Open Software Foundation [www.opengroup.org], era baseado no Andrew File System. O projeto do DCE/DFS vai além do AFS, particularmente em sua estratégia para consistência da cache. No AFS, os *callbacks* são gerados apenas quando o servidor recebe uma operação *close* para um arquivo que foi atualizado. O DFS adotou uma estratégia semelhante para o Spritely NFS e para o NQNFS, isto é, gera *callbacks* assim que um arquivo é atualizado. Para atualizar um arquivo, um cliente deve obter um *token write* do servidor, especificando um intervalo de bytes que o cliente pode atualizar no arquivo. Quando um *token write* é solicitado, os clientes que contêm cópias do mesmo arquivo para leitura recebem *callbacks* de revogação. *Tokens* de outros tipos são usados para obter consistência de atributos de arquivo colocados na cache e outros metadados. Todos os *tokens* têm um tempo de vida associado, e os clientes devem renová-los após sua expiração.

Aprimoramentos na organização do armazenamento • Houve um progresso considerável na organização dos dados de arquivos armazenados em discos. O impulso para grande parte desse trabalho surgiu das cargas maiores e da maior confiabilidade que os sistemas de arquivos distribuídos precisam suportar, e ele resultou em sistemas de arquivos com desempenho substancialmente melhor. Os principais resultados desse trabalho são:

RAID (Redundant Arrays of Inexpensive Disks): trata-se de um modo de armazenamento [Patterson *et al.* 1988, Chen *et al.* 1994] no qual blocos de dados são segmentados em porções de tamanho fixo e armazenados em “tiras” em vários discos (*striping*), junto a códigos redundantes de correção de erro que permitem aos blocos de dados serem completamente reconstruídos e, no caso de falhas de disco, as operações de leitura e escrita continuarem normalmente. O RAID também produz um desempenho consideravelmente melhor do que um único disco, pois as tiras (*strips*) que constituem um bloco são lidas e escritas concorrentemente.

LFS (Log-structured file storage): assim como o Spritely NFS, está técnica foi originada no projeto do sistema operacional distribuído Sprite, em Berkeley [Rosenblum e Ousterhout 1992]. Os autores observaram que, quando grandes quantidades de memória principal eram empregadas como cache em servidores de arquivos, melhor era o desempenho da leitura, porém o das operações de escrita continuava medíocre. Isso era proveniente das altas latências associadas à escrita de blocos de dados individuais no disco e às atualizações associadas nos blocos de metadados (isto é, os blocos conhecidos como *i-nodes*, que contêm atributos de arquivo e ponteiros para os blocos em um arquivo).

A solução do LFS é acumular um conjunto de escritas na memória e, então, efetivá-las no disco em grandes segmentos, adjacentes e de tamanho fixo. Eles são chamados de *segmentos de log* porque os blocos de dados e metadados são armazenados rigorosamente na ordem em que foram atualizados. Um segmento de *log* tem 1 Mbyte, ou mais, de tamanho, e é armazenado em uma única trilha do disco, eliminando as latências do cabeçote do disco associadas à escrita de blocos individuais. Os blocos de dados e metadados, novos ou atualizados, são sempre escritos no disco exigindo a manutenção de um mapa dinâmico (na memória, com um *backup* persistente) apontando para os blocos de *i-nodes*. Também é exigida a coleta de lixo de blocos antigos, com compactação de blocos “ativos”, para deixar áreas de armazenamento adjacentes livres para o armazenamento de segmentos de *log*. Este último é um processo bastante complexo; ele é executado em *background* por um componente cha-

mado *limpador* (*cleaner*). Alguns algoritmos limpadores sofisticados foram desenvolvidos para isso, com base nos resultados de simulações.

Apesar desses custos extras, o ganho de desempenho geral é significativo; Rosenblum e Ousterhout mediram um desempenho de escrita de até 70% da largura de banda disponível no disco, comparado com menos de 10% para um sistema de arquivos UNIX convencional. A estrutura do *log* também simplifica a recuperação após falhas do servidor. O sistema de arquivos Zebra [Hartman e Ousterhout 1995], desenvolvidos como uma continuação do trabalho original do LFS, combina escritas estruturadas em *log* com uma estratégia de RAID distribuído – os segmentos de *log* são subdivididos em seções com dados de correção de erro e escritos nos discos em diferentes nós de rede. É obtido um desempenho de quatro a cinco vezes o do NFS para escrita de arquivos grandes, com menores ganhos para arquivos curtos.

Novas estratégias de projeto • A disponibilidade de redes de alto desempenho (como a ATM e a Gigabit Ethernet) tem estimulado vários esforços para fornecer sistemas de armazenamento persistente que distribuem dados de arquivo de maneira altamente favorável à escalabilidade e à tolerância a falhas entre muitos nós em uma intranet, separando as responsabilidades da leitura e da escrita de dados das de gerenciar os metadados e atender às requisições dos clientes. A seguir, esboçamos dois desses desenvolvimentos.

Essas estratégias escalam melhor do que os servidores mais centralizados que descrevemos nas seções anteriores. Geralmente, elas exigem um alto nível de confiança entre os computadores que cooperam para fornecer o serviço, pois incluem um protocolo de nível mais baixo para a comunicação com os nós que contêm dados (algo muito parecido com uma API de “disco virtual”). Portanto, sua abrangência provavelmente é limitada a uma única rede local.

xFS: um grupo da Universidade da Califórnia, em Berkeley, propôs uma arquitetura de sistema de arquivos de rede sem servidor e desenvolveu um protótipo de implementação, o xFS [Anderson *et al.* 1996]. Sua estratégia foi motivada por três fatores:

1. a oportunidade oferecida pelas redes locais rápidas de permitir que vários servidores de arquivos transfiram grandes volumes de dados de forma concorrente para os clientes;
2. a maior demanda de acesso a dados compartilhados;
3. as limitações fundamentais dos sistemas baseados em servidores de arquivos centralizados.

A respeito de (3), eles se referem ao fato de que a construção de servidores NFS de alto desempenho exige *hardware* relativamente caro, com várias CPUs, controladoras de discos e de rede, e que existem limites para o processo de particionamento do espaço de arquivos – a distribuição de arquivos compartilhados em sistemas de arquivos separados, montados em diferentes servidores. Eles também apontam para o fato de que um servidor central representa um único ponto de falha.

O xFS é sem servidor, no sentido de que ele distribui as responsabilidades de processamento de um servidor de arquivos para um conjunto de computadores disponíveis em uma rede local, com uma granularidade de arquivos individuais. As responsabilidades de armazenamento são distribuídas independentemente do gerenciamento e de outras responsabilidades de serviço: o xFS implementa um sistema de armazenamento RAID em *software*, dividindo dados de arquivos entre discos de vários computadores (a esse respeito, ele é um precursor do Tiger Video File System, descrito no Capítulo 20), junto a uma técnica de estruturação de *log* de uma maneira semelhante ao sistema de arquivos Zebra.

A responsabilidade pelo gerenciamento de cada arquivo pode ser alocada para qualquer um dos computadores que suportam o serviço xFS. Isso é obtido por meio de

uma estrutura de metadados chamada de *mapa de gerência*, que é replicada em todos os clientes e servidores. Os identificadores de arquivo incluem um campo que atua como índice no mapa de gerência, e cada entrada no mapa identifica o computador correntemente responsável por gerenciar o arquivo correspondente. Várias outras estruturas de metadados, semelhantes àquelas encontradas em outros sistemas de armazenamento estruturado em *log* e RAID, são usadas para o gerenciamento do armazenamento de arquivos estruturado em *log* e no armazenamento em tiras de discos.

Foi construído um protótipo preliminar do xFS e seu desempenho foi avaliado. O protótipo estava incompleto no momento em que a avaliação foi realizada – a implementação da recuperação de falha estava inacabada e o esquema de armazenamento estruturado em *log* não tinha um componente limpador para recuperar o espaço ocupado por *logs* antigos e arquivos compactados.

As avaliações de desempenho realizadas com esse protótipo usaram 32 Sun SPARCStations, com um e com dois processadores (*dual*), conectados em uma rede de alta velocidade. As avaliações compararam o serviço de arquivo do xFS executando em até 32 estações de trabalho com o NFS e o AFS executando, cada um, em uma máquina *dual*. As larguras de banda de leitura e escrita obtidas com o xFS, usando 32 servidores, ultrapassaram as do NFS e do AFS, em um único servidor de processador *dual*, por um fator de aproximadamente dez. A diferença no desempenho foi muito menos pronunciada quando o xFS foi comparado com o NFS e o AFS usando o *benchmark* padrão do AFS. Porém, de modo geral, os resultados indicam que a arquitetura de processamento e armazenamento altamente distribuída do xFS apresentam uma tendência promissora para a obtenção de uma melhor escalabilidade em sistemas de arquivos distribuídos.

Frangipani: o Frangipani é um sistema de arquivos distribuído com alta escalabilidade, desenvolvido e implantado no Digital Systems Research Center (agora, Compaq Systems Research Center) [Thekkath *et al.* 1997]. Seus objetivos são muito parecidos com os do xFS e, assim como o xFS, ele os aborda com um projeto que separa as responsabilidades de armazenamento persistente das outras ações de um serviço de arquivos. No entanto, o serviço do Frangipani é estruturado como duas camadas totalmente independentes. A camada inferior é fornecida pelo sistema de disco virtual distribuído Petal [Lee e Thekkath 1996].

O Petal fornece uma abstração de disco virtual distribuído sobre os discos localizados em vários servidores, em uma rede local. A abstração de disco virtual tolera a maioria das falhas de *hardware* e de *software* com a ajuda de réplicas dos dados armazenados e harmoniza, por meio do reposicionamento dos dados, a carga de forma autônoma nos servidores. Os discos virtuais do Petal são acessados por meio de um *driver* de disco UNIX, usando operações de entrada e saída de blocos padrão, de modo que eles podem ser usados para suportar a maioria dos sistemas de arquivos. O Petal acrescenta entre 10 e 100% na latência de acessos a disco, mas a estratégia de uso da cache resulta em desempenhos de saída de leitura e escrita pelo menos tão bons quanto os das unidades de disco locais.

Os módulos servidores do Frangipani são executados dentro do núcleo do sistema operacional. Assim como no xFS, a responsabilidade por gerenciar arquivos e as tarefas associadas (incluindo serviço de travamento de arquivos para os clientes) é atribuída dinamicamente aos nós, e todas as máquinas enxergam um espaço de nomes de arquivo unificado, com acesso coerente (com semântica aproximada à da cópia única) para os arquivos compartilhados que podem ser atualizados. Os dados são armazenados em um formato estruturado em *log* e em tiras, no disco virtual Petal. O uso do Petal desobriga o Frangipani da necessidade de gerenciar espaço físico em disco, resultando em uma implementação de sistema de arquivos distribuído muito mais simples. O Frangipani pode simular as interfaces de vários serviços de arquivos existentes, incluindo o NFS e o DCE/DFS. O desempenho do Frangipani é pelo menos tão bom quanto o da implementação da Digital do sistema de arquivos UNIX.

12.6 Resumo

Os principais problemas de projeto para sistemas de arquivos distribuídos são:

- o uso eficiente de cache no cliente para obter um desempenho igual ou melhor do que o dos sistemas de arquivos locais;
- a manutenção da consistência entre várias cópias de arquivos de cliente colocadas em cache, quando essas cópias são atualizadas;
- a recuperação após uma falha do cliente ou do servidor;
- alto desempenho de saída para leitura e escrita de arquivos de todos os tamanhos;
- capacidade de mudança de escala.

Os sistemas de arquivos distribuídos são bastante empregados em sistemas computacionais institucionais, e seu desempenho tem sido o assunto de muita otimização. O NFS possui um protocolo sem estado (*stateless*), mas tem mantido, com a ajuda de alguns aprimoramentos relativamente pequenos no protocolo, sua posição inicial como tecnologia de sistema de arquivos distribuído dominante contando com implementações otimizadas e suporte de *hardware* de alto desempenho.

O AFS demonstrou a exequibilidade de uma arquitetura relativamente simples, usando o estado do servidor para reduzir o custo da manutenção de caches de cliente coerentes. O AFS supera o NFS em muitas situações. Avanços recentes têm empregado a disposição de dados em tiras (*striping*) entre vários discos e escrita estruturada em *log* para melhorar ainda mais o desempenho e a capacidade de escala.

Os sistemas de arquivos distribuídos mais modernos têm alta escalabilidade, fornecem bom desempenho entre redes locais e remotas, mantêm a semântica de atualização de arquivos com cópia única e toleram falhas e se recuperam delas. Os requisitos futuros incluem o suporte para usuários móveis com operação desconectada, e reintegração automática e garantia de qualidade do serviço para satisfazer a necessidade de armazenamento persistente e a distribuição de fluxos de dados multimídia e outros fluxos dependentes do tempo. As soluções para esses requisitos serão discutidas nos Capítulos 18 e 20.

Exercícios

- 12.1 Por que não existe nenhuma operação *open* ou *close* em nossa interface para o serviço de arquivos simples ou para o serviço de diretório? Quais são as diferenças entre nossa operação de serviço de diretório *Lookup* e a operação *open* do UNIX? *páginas 532–534*
- 12.2 Descreva, em linhas gerais, os métodos pelos quais um módulo cliente poderia simular a interface de sistema de arquivos UNIX usando nosso modelo de serviço de arquivos. *páginas 532–534*
- 12.3 Escreva uma função *PathLookup(Pathname, Dir) → UFID* que implemente a operação *Lookup* para nomes de caminho do tipo UNIX, com base em nosso modelo de serviço de diretório. *páginas 532–534*
- 12.4 Por que os UFIDs devem ser exclusivos entre todos os sistemas de arquivos possíveis? Como a exclusividade dos UFIDs é garantida? *página 535*
- 12.5 Até que ponto o Sun NFS se desvia da semântica de atualização de arquivo com cópia única? Construa um cenário no qual dois processos usuário compartilhando um arquivo funcionariam corretamente em um único computador UNIX, mas observariam inconsistências ao serem executados em computadores diferentes. *página 542*
- 12.6 O Sun NFS pretende suportar sistemas distribuídos heterogêneos por meio de um serviço de arquivo independente do sistema operacional. Quais são as principais decisões que o de-

senvolvedor de um servidor NFS para um sistema operacional que não seja o UNIX teria de tomar? Quais restrições um sistema de armazenamento deve obedecer para ser conveniente para a implementação de servidores NFS? *página 536*

- 12.7 Quais dados o módulo cliente NFS deve conter sobre cada processo usuário? *páginas 536–537*
- 12.8 Descreva, em linhas gerais, as implementações de módulo cliente para as chamadas de sistema UNIX *open()* e *read()*, usando as chamadas de RPC do NFS da Figura 12.9, (i) sem e (ii) com uma cache no cliente. *páginas 538, 542*
- 12.9 Explique por que a interface RPC das primeiras implementações do NFS é potencialmente insegura. A brecha de segurança foi fechada no NFS 3 pelo uso de criptografia. Como a chave de criptografia é mantida em segredo? A segurança da chave é adequada? *páginas 539, 544*
- 12.10 Após o *timeout* de uma chamada de RPC para acessar um arquivo em um sistema de arquivos montado incondicionalmente, o módulo cliente NFS não retorna o controle para o processo usuário que originou a chamada. Por quê? *página 539*
- 12.11 Como o *automounter* do NFS ajuda a melhorar o desempenho e a capacidade de mudança de escala do NFS? *página 541*
- 12.12 Quantas chamadas de *Lookup* são necessárias para solucionar um nome de caminho de cinco partes (por exemplo, */usr/users/jim/code/xyz.c*) para um arquivo armazenado em um servidor NFS? Qual é o motivo para realizar a transformação passo a passo? *página 540*
- 12.13 Qual condição deve ser atendida pela configuração das tabelas de montagem nos computadores clientes para que a transparência de acesso seja obtida em um sistema de arquivos baseado no NFS? *página 540*
- 12.14 Como o AFS obtém o controle quando é executada, por um cliente, uma chamada de sistema de abertura (ou fechamento) referindo-se a um arquivo no espaço de arquivo compartilhado? *página 549*
- 12.15 Compare a semântica de atualização do UNIX ao acessar arquivos locais com as do NFS e do AFS. Sob quais circunstâncias os clientes poderiam conhecer as diferenças? *páginas 542, 554*
- 12.16 Como o AFS lida com o risco de perda de mensagens de *callback*? *página 552*
- 12.17 Quais características do projeto do AFS o tornam mais escalável que o NFS? Quais os limites para sua escalabilidade, supondo que servidores podem ser adicionados conforme for exigido? Quais desenvolvimentos recentes oferecem maior escalabilidade? *páginas 545, 556, 561*

13

Serviço de Nomes

- 13.1 Introdução
- 13.2 Serviço de nomes e o Domain Name System
- 13.3 Serviço de diretório
- 13.4 Estudo de caso: Global Name Service
- 13.5 Estudo de caso: X.500 Directory Service
- 13.6 Resumo

Este capítulo apresenta o serviço de nomes como um serviço distinto a ser usado por processos clientes para, a partir do nome simbólico de um recurso ou objeto, obter seus atributos como, por exemplo, seu endereço. As entidades nomeadas podem ser de vários tipos e podem ser gerenciadas por diferentes serviços. Por exemplo, os serviços de nomes são frequentemente usados para conter os endereços e outros detalhes, de usuários, computadores, domínios de rede, serviços e objetos remotos. Além dos serviços de nomes, descreveremos os serviços de diretório, os quais pesquisam por serviços a partir de alguns de seus atributos.

Os problemas básicos de projeto do serviço de nomes, como sua estrutura e seu gerenciamento, e as operações suportadas por ele são descritos em linhas gerais e ilustrados no contexto do Domain Name Service da Internet.

Também examinaremos como os serviços de nomes são implementados, abordando aspectos como a navegação por um conjunto de servidores de nome ao resolver um nome, o armazenamento em cache dos resultados da resolução e a replicação de nomes e atributos para aumentar o desempenho e a disponibilidade.

São incluídos mais dois estudos de caso: o Global Name Service (GNS) e o X.500 Directory Service, incluindo o LDAP.

13.1 Introdução

Em um sistema distribuído, são usados nomes para fazer referência a uma ampla variedade de recursos, como computadores, serviços, objetos remotos e arquivos, assim como usuários. A atribuição de nomes é um problema facilmente desprezado, mas com certeza fundamental no projeto de sistemas distribuídos. Os nomes facilitam a comunicação e o compartilhamento de recursos. É necessário um nome para pedir a um sistema de computador para atuar sobre um recurso específico escolhido entre muitos; por exemplo, é necessário um nome na forma de um URL para acessar uma página Web específica. Os processos não podem compartilhar recursos em particular a não ser que possam atribuir nomes a eles de forma consistente. Os usuários não podem se comunicar por meio de um sistema distribuído, a menos que possam dar nomes uns aos outros, por exemplo, com endereços de *e-mail*.

Os nomes não são a única maneira útil de identificação: outras formas são os atributos descritivos. Às vezes, os clientes não sabem o nome da entidade em particular que procuram, mas têm alguma informação que a descreve. Ou então, o cliente solicita um serviço (em vez de uma entidade em particular que o implemente) e conhece algumas das características que o serviço exigido deve ter.

Este capítulo apresenta os serviços de nomes, os quais fornecem aos clientes dados sobre objetos nomeados em sistemas distribuídos, e o conceito relacionado dos serviços de diretório, os quais fornecem dados sobre objetos que satisfazem determinada descrição. Descreveremos estratégias adotadas no projeto e na implementação desses serviços, usando o DNS (Domain Name Service), o GNS (Global Name Service) e o X.500 como estudos de caso. Começaremos examinando os conceitos fundamentais dos nomes e atributos.

13.1.1 Nomes, endereços e outros atributos

Todo processo que exige acesso a um recurso específico deve possuir um nome ou um identificador para ele. Exemplos de nomes legíveis por seres humanos são nomes de arquivo, como `/etc/passwd`, URLs, como `http://www.cdk5.net/`, e nomes de domínio Internet, como `www.cdk5.net`. O termo *identificador* às vezes é usado para se referir aos nomes interpretados apenas por programas. Referências de objeto remotas e manipuladores de arquivo NFS são exemplos de identificadores. Os identificadores são escolhidos pela eficiência com que podem ser pesquisados e armazenados pelo *software*.

Needham [1993] faz uma distinção entre um nome *puro* e outros tipos de nomes. Os nomes puros são simplesmente padrões de bits não interpretados. Os nomes que não são puros contêm informações sobre o objeto que nomeiam; em particular, podem conter informações sobre a localização do objeto. Os nomes puros sempre precisam ser pesquisados antes de poderem ser usados. No outro extremo está o *endereço* de um objeto: um valor que identifica a localização do objeto, em vez do objeto em si. Os endereços são eficientes para acessar objetos, mas os objetos às vezes podem ser movidos; portanto, os endereços são inadequados como meio de identificação. Por exemplo, os endereços de *e-mail* dos usuários normalmente precisam mudar quando eles trocam de organizações ou de provedores de serviços de Internet; eles não são autossuficientes para fazer referência a um indivíduo específico com o passar do tempo.

Dizemos que um nome é *resolvido* quando ele é convertido em dados sobre o recurso ou objeto nomeado, frequentemente para invocar uma ação sobre ele. A associação entre um nome e um objeto é chamada de *vínculo* (*binding*). Em geral, os nomes são vinculados a *atributos* dos objetos nomeados, em vez de à implementação dos objetos em si. Atributo é o valor de uma propriedade associada a um objeto. Um atributo importante de

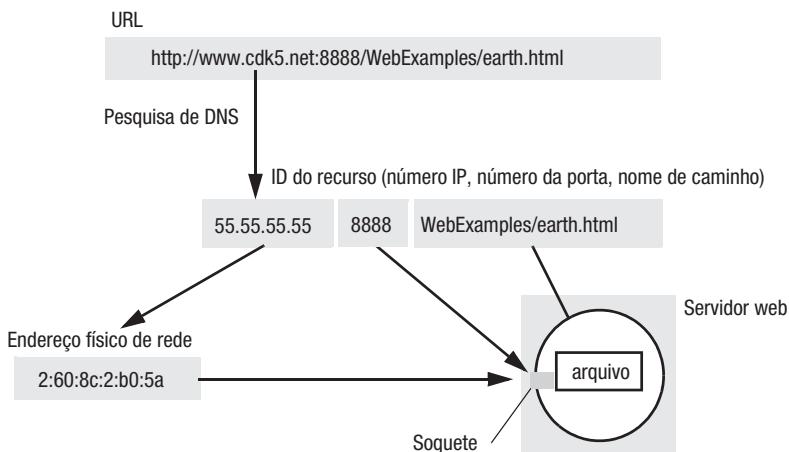


Figura 13.1 Composição de um nome de domínio para identificar um recurso a partir de um URL.

uma entidade, que normalmente é relevante em um sistema distribuído, é seu endereço. Por exemplo:

- O DNS faz o mapeamento de nomes de domínio Internet para atributos de um computador: seu endereço IP, o tipo de entrada (por exemplo, uma referência a um servidor de correio eletrônico ou a outro computador) e, por exemplo, o período de tempo durante o qual a entrada desse computador permanecerá válida.
- O serviço de diretório X500 pode ser usado para fazer o mapeamento do nome de uma pessoa em atributos, incluindo seu endereço de *e-mail* e número de telefone.
- O serviço de nomes (*Naming Service*) e o serviço de negociação (*Trading Service*) do CORBA foram apresentados no Capítulo 8. O serviço de nomes faz o mapeamento do nome de um objeto remoto para sua referência de objeto remota, enquanto o serviço de negociação (*Trading Service*) faz esse mesmo mapeamento junto a um número arbitrário de atributos descrevendo o objeto em termos inteligíveis por usuários humanos.

Note que um “endereço” pode ser frequentemente considerado apenas como outro nome que deve ser pesquisado, ou que pode conter tal nome. Um endereço IP deve ser pesquisado para se obter um endereço físico de rede, como um endereço Ethernet (MAC). Analogamente, os navegadores Web e os clientes de *e-mail* utilizam o DNS para interpretar os nomes de domínio nos URLs e endereços de *e-mail*. A Figura 13.1 mostra a resolução da primeira parte de um nome de domínio de um URL, primeiro por meio do DNS, em um endereço IP, e depois, no passo final do roteamento pela Internet, por meio de ARP, em um endereço Ethernet, para o servidor Web. A última parte do URL é resolvida pelo sistema de arquivos do servidor Web para localizar o arquivo relevante.

Nomes e serviços • Muitos nomes usados em um sistema distribuído são específicos para algum serviço em particular. Por exemplo, os usuários do site de rede social `twitter.com` têm nomes, como `@magma poetry`, que nenhum outro serviço resolve. Além disso, um cliente pode usar um nome específico do serviço ao solicitar que esse serviço efetue uma operação sobre um objeto ou recurso nomeado que gerencia. Por exemplo, para solicitar a exclusão de

um arquivo, deve-se fornecer o seu nome para o serviço de arquivos; e, ao enviar um sinal para um processo, é necessário identificá-lo junto ao serviço de gerenciamento de processos. Esses nomes são usados apenas no contexto do serviço que gerencia os objetos nomeados, exceto quando os clientes se comunicam a respeito de objetos compartilhados.

Às vezes, os nomes também são necessários para referenciar às entidades de um sistema distribuído que estão fora da abrangência de um serviço. Os principais exemplos dessas entidades são usuários (com nomes próprios e endereços de *e-mail*), computadores (com nomes como *www.cdk5.net*) e os próprios serviços (como *serviço de arquivos*, *serviço de impressora*). No *middleware* baseado em objetos, os nomes se referem aos objetos remotos que fornecem serviços ou aplicativos. Note que muitos desses nomes devem ser legíveis e significativos para seres humanos, pois os usuários e administradores de sistema precisam se referir aos principais componentes e à configuração de sistemas distribuídos; os programadores precisam se referir a serviços em programas; e os usuários precisam se comunicar por meio do sistema distribuído e discutir quais serviços estão disponíveis em diferentes partes dele. Dada a conectividade fornecida pela Internet, esses requisitos de atribuição de nomes são potencialmente mundiais na abrangência.

Uniform Resource Identifiers • Os URIs (Uniform Resource Identifiers) [Berners-Lee *et al.* 2005] surgiram da necessidade de identificar recursos Web e outros recursos de Internet, como as caixas de correio eletrônico. Um objetivo importante era identificar esses recursos de maneira coerente, de modo que todos pudessem ser processados por um *software* comum, como os navegadores. Os URIs são uniformes no sentido de que sua sintaxe incorpora a de muitos tipos de identificadores de recurso individuais (isto é, *esquemas* de URI) e de que existem procedimentos para gerenciar o espaço de nomes global desses esquemas. A vantagem da uniformidade é que ela facilita a introdução de novos tipos de identificador, assim como o uso em novos contextos dos tipos já existentes, sem invalidar a sua atual utilização.

Por exemplo, se alguém inventasse um novo tipo de URI *widget*, então os URIs que começassem com *widget*: teriam que obedecer à sintaxe de URI global, assim como todas as regras locais definidas para o esquema identificador de *widget*. Esses URIs identificariam recursos de *widget* de uma maneira bem definida. Contudo, mesmo um *software* já existente que não acessasse recursos de *widget* ainda poderia processar URIs *widget* – por exemplo, gerenciando diretórios que os contivesse. Recorrendo a um exemplo de incorporação de identificadores existente, isso foi feito para os números de telefone, prefixando-se o nome de esquema *tel* e padronizando-se a representação de números de telefone, como em *tel:+1-816-555-1212*. Esses URIs *tel* se destinam a usos como *links* Web que fazem ligações telefônicas quando invocados.

Uniform Resource Locators: alguns URIs contêm informações para localizar e acessar um recurso; outros são nomes de recurso puros. O conhecido termo *Uniform Resource Locator (URL)* é frequentemente usado para URIs que fornecem informações de localização e especificam o método para acessar um recurso, incluindo os URLs *http*, apresentados na Seção 1.6. Por exemplo, *http://www.cdk5.net/* identifica a página Web no caminho dado por / no computador *www.cdk5.net* e especifica o protocolo HTTP usado para acessá-lo. Outro exemplo é um URL *mailto*, como *mailto:fred@flintstone.org*, que identifica a caixa de correio eletrônico associada ao endereço eletrônico fornecido.

Os URLs são identificadores eficientes para acessar recursos, mas sofrem da desvantagem de que, se um recurso for excluído ou mudar, digamos, de um site Web para outro, então poderão existir *links* “quebrados” para o recurso, contendo o URL antigo. Se um usuário clicar em um *link* quebrado de um recurso Web, o servidor Web responderá

que o recurso não pode ser encontrado ou – talvez, pior – fornecerá um recurso diferente, que agora ocupa a mesma localização.

Uniform Resource Names: os URNs (Uniform Resource Names) são URIs utilizados como nomes de recurso puros, em vez de localizadores. Por exemplo, o URI:

mid:0E4FC272-5C02-11D9-B115-000A95B55BC8@hpl.hp.com

é um URN que identifica a mensagem de *e-mail* contida em seu campo *Message-Id*. O URI distingue essa mensagem de qualquer outra mensagem de *e-mail*. Porém, ele em si não fornece o endereço da mensagem em algum meio de armazenamento, e é necessária uma operação de pesquisa para encontrá-la.

Uma subárvore especial dos URIs que começam com *urn*: foi reservada para os URNs – embora, como mostra o exemplo *mid*: nem todos os URNs são URIs *urn*:. Todos estes URIs prefixados com *urn* são da forma *urn:espaçodeNome:espaçodeNome-nomeEspecifico*. Por exemplo, *urn:ISBN:0-201-62433-8* identifica livros que apresentam o nome 0-201-62433-8 no esquema de atribuição de nomes padrão ISBN. Como outro exemplo, o nome (inventado) *urn:doi:10.555/music-pop-1234* se refere à publicação chamada *music-pop-1234* no esquema de atribuição de nomes da editora conhecida como *10.555* no esquema DOI (Digital Object Identifier) [www.doi.org].

Existem serviços de resolução (serviços de nomes, na terminologia deste capítulo), como o *Handle System*, para resolver URNs do tipo DOI [www.handle.net] em atributos de recurso, mas nenhum deles é amplamente usado. Na verdade, o debate continua nas comunidades de pesquisa da Web e Internet sobre até que ponto uma categoria separada de URNs é necessária. Uma escola de pensamento diz que “os bons URLs não mudam” – em outras palavras, que todo mundo deve atribuir URLs aos recursos com garantias sobre a continuidade de sua referência. Contra esse ponto de vista está a observação de que nem todo mundo está em uma posição favorável para dar tais garantias, pois exige os meios para se manter o controle de um nome de domínio e administrar os recursos com cuidado.

13.2 Serviços de nomes e o Domain Name System

Um *serviço de nomes* armazena informações sobre um conjunto de nomes textuais, na forma de vínculos entre os nomes e nos atributos das entidades que denotam, como usuários, computadores, serviços e objetos. O conjunto é frequentemente subdividido em um ou mais *contextos* de atribuição de nomes: subconjuntos individuais dos vínculos que são gerenciados como uma unidade. A principal operação que um serviço de nomes suporta é a resolução de um nome – isto é, pesquisar atributos de determinado nome. Descreveremos a implementação da resolução de nomes na Seção 13.2.2. Também são necessárias operações para criar novos vínculos, excluir vínculos, listar nomes vinculados e para adicionar e excluir contextos.

Devido ao requisito dos sistemas distribuídos serem abertos, o gerenciamento de nomes é completamente separado dos outros serviços, o que traz as seguintes motivações:

Unificação: frequentemente, é conveniente que recursos gerenciados por diferentes serviços utilizem o mesmo esquema de atribuição de nomes. Os URIs são um bom exemplo disso.

Integração: nem sempre é possível prever a abrangência do compartilhamento em um sistema distribuído. Pode-se tornar necessário compartilhar e, portanto, nomear recursos que foram criados em diferentes domínios administrativos. Sem um serviço de nomes comum, os domínios administrativos podem usar convenções de atribuição de nomes totalmente diferentes.

Requisitos gerais do serviço de nomes • Originalmente, os serviços de nomes eram muito simples, pois foram projetados apenas para atender à necessidade de vincular nomes a endereços em um único domínio de gerenciamento, correspondendo a uma única LAN ou WAN. A interconexão de redes, e a maior escala dos sistemas distribuídos, geraram um problema de mapeamento de nomes muito maior.

O Grapevine [Birrell *et al.* 1982] foi um dos primeiros serviços de nomes extensíveis e de múltiplos domínios. Ele foi projetado para ser escalável no número de nomes e na carga de requisições que poderia manipular.

O Global Name Service, desenvolvido no Digital Equipment Corporation Systems Research Center [Lampson 1986], é um descendente do Grapevine com objetivos ambiciosos, incluindo:

Manipular um número essencialmente arbitrário de nomes e servir a um número arbitrário de organizações administrativas. Por exemplo, o sistema deve ser capaz de manipular os nomes de todos os documentos do mundo.

Um tempo de ciclo de vida longo. Muitas mudanças irão ocorrer na organização do conjunto de nomes e nos componentes que implementam o serviço durante seu tempo de ciclo de vida.

Alta disponibilidade. A maior parte dos outros sistemas depende do serviço de nomes; e eles não podem funcionar quando o serviço de nomes está com defeito.

Isolamento de falhas. Para que falhas locais não façam o serviço inteiro falhar.

Tolerância à suspeita. Um sistema aberto grande não pode ter nenhum componente que seja confiável para todos os clientes do sistema.

Dois exemplos de serviços de nome que se concentraram no objetivo de escalabilidade para grandes números de objetos, como documentos, são o serviço de nomes Globe [van Steen *et al.* 1998] e o Handle System [www.handle.net]. Bem mais conhecido é o DNS (Domain Name System) da Internet, apresentado no Capítulo 3, que nomeia computadores (e outras entidades) na Internet.

Nesta seção, discutimos os principais problemas de projeto de serviços de nomes, dando exemplos do DNS. A seguir, apresentaremos um estudo de caso mais detalhado do DNS.

13.2.1 Espaços de nomes

Espaço de nome é o conjunto de todos os nomes válidos reconhecidos por um serviço em particular. O serviço tentará pesquisar um nome válido, mesmo que o nome não venha a corresponder a nenhum objeto – ou seja, esteja *desvinculado*. Os espaços de nomes exigem uma definição sintática para separar nomes válidos de nomes inválidos. Por exemplo, “...” não é aceitável como nome DNS de um computador, enquanto www.cdk99.net é válido (mesmo que esteja desvinculado).

Os nomes podem ter uma estrutura interna que represente sua posição em um espaço de nomes hierárquico, como os nomes de caminho no sistema de arquivos, ou em uma hierarquia organizacional, como os nomes de domínio Internet; ou ainda, eles podem ser escolhidos em um conjunto simples de identificadores numéricos ou simbólicos. Uma vantagem importante da hierarquia é que ela facilita o gerenciamento de espaços de nome grandes. Cada parte de um nome hierárquico é resolvida em relação a um contexto separado, de tamanho relativamente pequeno, e o mesmo nome pode ser usado com diferentes significados, em diferentes contextos, para se adequar às diferentes situações de uso. No caso dos sistemas de arquivos, cada diretório representa um contexto. Assim,

/etc/passwd é um nome hierárquico com dois componentes. O primeiro, *etc*, é transformado em relação ao contexto */*, ou raiz, e a segunda parte, *passwd*, é relativa ao contexto */etc*. O nome */old/etc/passwd* pode ter um significado diferente, pois seu segundo componente é resolvido em outro contexto. Analogamente, o mesmo nome */etc/passwd* pode ser resolvido para arquivos diferentes nos contextos de dois computadores diferentes.

Os espaços de nome hierárquicos são potencialmente infinitos, de modo que eles permitem que um sistema cresça indefinidamente. Em contraste, os espaços de nomes planos normalmente são finitos; seu tamanho é determinado pela fixação de um comprimento máximo permitido para os nomes. Outra vantagem em potencial de um espaço de nome hierárquico é que diferentes contextos podem ser gerenciados por diferentes pessoas ou organizações.

A estrutura dos URLs *http* foi apresentada no Capítulo 1. O espaço de nomes URL também inclui *nomes relativos*, como *../images/figure1.jpg*. Quando um navegador ou outro cliente Web encontra um nome relativo assim, ele usa o recurso no qual o nome está incorporado para determinar o nome do servidor e o diretório ao qual esse nome de caminho se refere.

Os nomes DNS são *strings* chamadas de *nomes de domínio*. Alguns exemplos são: *www.cdk5.net* (um computador), *net*, *com* e *ac.uk* (os três últimos são domínios).

O espaço de nomes DNS tem uma estrutura hierárquica: um nome de domínio consiste em um ou mais *strings* chamados de *componentes do nome* ou *rótulos*, separados por um delimitador. Não há nenhum delimitador no início ou no final de um nome de domínio, embora a raiz do espaço de nome DNS às vezes seja referenciada como ‘.’ para propósitos administrativos. Os componentes do nome são *strings* imprimíveis não nulos que não contêm ‘.’. Em geral, o *prefixo* de um nome é a sua parte inicial que contém zero ou mais componentes. Por exemplo, no DNS, *www* e *www.cdk5* são prefixos de *www.cdk5.net*. Os nomes DNS não levam em consideração letras maiúsculas e minúsculas; portanto, *www.cdk5.net* e *WWW.CDK5.NET* têm o mesmo significado.

Os servidores DNS não reconhecem nomes relativos: todos os nomes se referem à raiz global. Entretanto, na prática, as implementações dos *softwares* cliente mantêm uma lista de nomes de domínio que são anexados automaticamente a todo nome de componente único, antes da transformação. Por exemplo, o nome *www* apresentado no domínio *cdk5.net* provavelmente se refere a *www.cdk5.net*; o *software* cliente anexará o domínio padrão *cdk5.net* e tentará resolver esse nome. Se isso falhar, então mais nomes de domínio padrão poderão ser anexados; finalmente, o nome (absoluto) *www* é apresentado à raiz para resolução (uma operação que, é claro, falhará neste caso). Contudo, nomes com mais de um componente normalmente são apresentados intactos para o DNS – como nomes absolutos.

Aliases • Um *alias* é um nome adicional, criado para fornecer as mesmas informações de outro nome, isto é, semelhante a um vínculo simbólico entre os nomes de caminho de um arquivo. Os *aliases* permitem que nomes mais convenientes substituam outros mais complicados e que nomes alternativos sejam usados para a mesma entidade por diferentes pessoas. Um exemplo é o uso comum de redutores de URL, frequentemente utilizados nas postagens do Twitter e em outras situações em que o tamanho é importante. Por exemplo, usando redirecionamento na Web, *http://bit.ly/ctqjvH* se refere a *http://cdk5.net/additional/rmi/programCode/ShapeListClient.java*. Como outro exemplo, o DNS permite fazer *aliases* em que um nome de domínio é definido para representar outro. Os *aliases* são geralmente usados para especificar os nomes das máquinas que executam um servidor Web ou um servidor FTP. Por exemplo, o nome *www.cdk5.net* é um *alias* para

cdk5.net. Isso tem a vantagem de que os clientes podem usar um ou outro nome para o servidor Web e, se o servidor Web mudar para outro computador, apenas a entrada para *cdk5.net* precisa ser atualizada no banco de dados DNS.

Domínios de atribuição de nomes • Um *domínio de atribuição de nomes* é um espaço de nome para o qual existe uma única autoridade administrativa global para atribuir nomes dentro dele. Essa autoridade tem o controle geral de quais nomes podem ser vinculados dentro do domínio, mas está livre para delegar essa tarefa.

No DNS, os domínios são conjuntos de nomes de domínio; sintaticamente, o nome de um domínio é o sufixo comum dos nomes de domínio que estão dentro dele, mas não pode ser distinguido de outra forma, por exemplo, do nome de um computador. Por exemplo, *net* é um domínio que contém *cdk5.net*. Note que o termo “nome de domínio” é potencialmente confuso, pois apenas alguns nomes de domínio identificam domínios (outros identificam computadores).

A administração de domínios pode ser delegada para subdomínios. O domínio *dcs.qmul.ac.uk* – o Departamento de Ciência da Computação do Queen Mary, Universidade de Londres, no Reino Unido – pode conter qualquer nome que o departamento queira. No entanto, o nome de domínio *dcs.qmul.ac.uk*, em si, teve de ser combinado junto às autoridades da escola que gerenciam o domínio *qmul.ac.uk*. Analogamente, *qmul.ac.uk* teve de ser aceito pela autoridade registrada para *ac.uk*, etc.

A responsabilidade por um domínio de atribuição de nomes normalmente anda lado a lado com a de gerenciar e manter atualizada a parte correspondente do banco de dados armazenado em um servidor de nome autoridade sobre o domínio (*authoritative name server*) e usado pelo serviço de nomes. Em geral, os nomes pertencentes a diferentes domínios de atribuição de nomes são armazenados por servidores de nome distintos, gerenciados pelas autoridades correspondentes.

Combinação e personalização de espaços de nomes • O DNS fornece um espaço de nomes global e homogêneo, no qual um determinado nome se refere à mesma entidade, independentemente de quem pesquise o nome. Em contraste, alguns serviços de nomes permitem que espaços de nomes distintos – às vezes, espaços de nomes heterogêneos – sejam incorporados a eles; e alguns serviços de nomes permitem, ainda, que o espaço de nomes seja personalizado de acordo com as necessidades de grupos individuais, usuários ou mesmo processos.

Integração: a prática de montar sistemas de arquivos no UNIX e no NFS (veja a Seção 12.3) fornece um exemplo no qual uma parte de um espaço de nome é convenientemente incorporada à outra. No entanto, considere a integração dos sistemas de arquivos inteiros de dois (ou mais) computadores UNIX chamados *red* e *blue*. Cada computador tem sua própria raiz, com nomes de arquivo coincidentes. Por exemplo, */etc/passwd* se refere a um arquivo em *red* e a um arquivo diferente em *blue*. A maneira óbvia de integrar os sistemas de arquivos é substituir a raiz de cada computador por uma “super-raiz” e montar o sistema de arquivos de cada computador nessa super-raiz, digamos, como */red* e */blue*. Então, os usuários e programas poderiam se referir a */red/etc/passwd* e a */blue/etc/passwd*. Contudo, a nova convenção de atribuição de nomes, por si só, faria os programas dos dois computadores que ainda usam o nome antigo */etc/passwd* funcionar de forma errada. Uma solução é deixar o conteúdo da raiz antiga em cada computador e incorporar os sistemas de arquivos montados */red* e */blue* nos dois computadores (supondo que isso não produza conflitos de nome com o conteúdo da raiz antiga).

A moral é que sempre podemos integrar espaços de nomes criando um contexto de raiz de nível mais alto, mas isso pode provocar um problema de compatibilidade com

versões anteriores. A correção do problema da compatibilidade, por sua vez, deixa-nos com espaços de nomes mistos e a inconveniência de ter de transladar nomes antigos entre os usuários dos dois computadores.

Heterogeneidade: o espaço de nome DCE (Distributed Computing Environment) [OSF 1997] permite a incorporação de espaços de nomes heterogêneos dentro dele. Os nomes DCE podem conter *junções*, que são semelhantes aos pontos de montagem do NFS e do UNIX (veja a Seção 12.3), exceto por permitirem a montagem de espaços de nomes heterogêneos. Por exemplo, considere o nome DCE completo *.../dcs.qmul.ac.uk/principals/Jean.Dollimore*. A primeira parte desse nome, *.../dcs.qmul.ac.uk*, denota um contexto chamado *célula*. O componente seguinte é uma junção. Por exemplo, a junção *principals* é um contexto contendo principais de segurança, no qual o componente final, *Jean.Dollimore*, pode ser pesquisado e no qual esses nomes de principal têm sua própria sintaxe. Analogamente, em *.../dcs.qmul.ac.uk/files/pub/reports/TR2000-99*, a junção *files* é um contexto associado a um diretório do sistema de arquivos, no qual o componente final, *pub/reports/TR2000-99*, é pesquisado e no qual o espaço de nome de arquivo tem sintaxe distinta. As duas junções, *principals* e *files*, são as raízes de espaços de nomes heterogêneos, implementados por serviços de nomes heterogêneos.

Customização: vimos, no exemplo anterior da incorporação de sistemas de arquivos montados com o NFS, que às vezes os usuários preferem construir seus espaços de nomes de forma independente, em vez de compartilhar um único espaço de nomes. A montagem do sistema de arquivos permite que os usuários importem arquivos que estão armazenados em servidores e são compartilhados, enquanto os outros nomes continuam a fazer referência a arquivos locais, não compartilhados, e que podem ser administrados de forma autônoma. Contudo, até os mesmos arquivos acessados a partir de dois computadores diferentes podem ser montados em diferentes pontos e, assim, ter nomes diferentes. Não compartilhando o espaço de nomes inteiro, os usuários precisam traduzir nomes entre computadores.

O serviço de nomes Spring [Radia *et al.* 1993] apresenta a capacidade de construir espaços de nomes dinamicamente e compartilhar contextos de atribuição de nomes individuais de forma seletiva. Mesmo dois processos diferentes no mesmo computador podem ter contextos de nomes diferentes. Os contextos de nomes do Spring são objetos de primeira classe que podem ser compartilhados em torno de um sistema distribuído. Por exemplo, suponha que um usuário no computador *red* deseje executar um programa em *blue* que usa nome de caminhos como */etc/passwd*, mas que esses nomes sejam resolvidos para arquivos do sistema de arquivos de *red* e não no de *blue*. Isso pode ser obtido no Spring, passando-se uma referência do contexto de nomes local de *red* para *blue* e usando-o como contexto de nomes do programa. O Plan 9 [Pike *et al.* 1993] também permite que os processos tenham seus próprios espaços de nomes do sistema de arquivos. Um recurso novo do Plan 9 (mas que também pode ser implementado no Spring) é que os diretórios físicos podem ser ordenados e fusionados em um único diretório lógico. O efeito é que um nome no diretório lógico único é pesquisado na sucessão de diretórios físicos até que haja uma correspondência, quando, então, os atributos são retornados. Isso elimina a necessidade de fornecer listas de caminhos ao procurar arquivos de programa ou de biblioteca.

13.2.2 Resolução de nomes

Para o caso mais comum de espaços de nome hierárquicos, a resolução de nomes é um processo iterativo ou recursivo pelo qual um nome é repetidamente apresentado a diferentes contextos de atribuição de nomes para pesquisar os atributos aos quais ele se refere. Um contexto de atribuição de nomes faz o mapeamento de determinado nome

diretamente em um conjunto de atributos primitivos (como os de um usuário) ou faz o mapeamento a um novo contexto de atribuição de nomes e em um nome derivado a ser apresentado a esse contexto. Para resolver um nome, ele é primeiro apresentado a um contexto de atribuição de nomes inicial; enquanto o resultado da resolução for um novo contexto, ou um nome derivado, o processo continua de forma iterativa. Ilustramos isso no início da Seção 13.2.1, com o exemplo de */etc/passwd*, no qual *etc* é apresentado ao contexto / e depois *passwd* é apresentado ao contexto */etc*.

Outro exemplo da natureza iterativa da transformação é o uso de *alias*. Por exemplo, quando um servidor de DNS é solicitado a resolver um *alias* como *www.dcs.qmul.ac.uk*, ele primeiro resolve o *alias* no outro nome de domínio (neste caso, *traffic.dcs.qmul.ac.uk*), o qual deve ser novamente resolvido para produzir um endereço IP.

Em geral, o uso de *alias* possibilita a presença de ciclos no espaço de nomes, caso em que a resolução poderá nunca terminar. Duas soluções são, primeiro, abandonar um processo de resolução após um número limite de sucessivas resoluções; e, segundo, deixar os administradores vetarem os *aliases* que introduziriam ciclos.

Servidores de nomes e navegação • Qualquer serviço de nomes, como o DNS, que armazene um banco de dados muito grande e que seja usado por uma população grande, não armazenará todas as suas informações de atribuição de nomes em um único servidor. Tal servidor seria um gargalo e um ponto de falha crítico. Todos os serviços de nomes muito utilizados devem usar replicação para obter alta disponibilidade. Veremos que o DNS especifica que cada subconjunto de seu banco de dados é replicado em pelo menos dois servidores independentes.

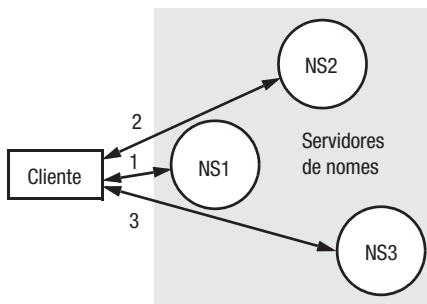
Mencionamos anteriormente que os dados pertencentes a um domínio de atribuição de nomes normalmente é armazenado por um servidor de nome local gerenciado pela autoridade responsável por esse domínio. Embora, em alguns casos, um servidor de nome possa armazenar dados de mais de um domínio, geralmente pode-se dizer que os dados são particionados nos servidores de acordo com seu domínio. Veremos que, no DNS, a maioria das entradas é relativa a computadores locais. Contudo, também existem servidores de nome para os domínios superiores, como *yahoo.com*, *ac.uk*, e para a raiz.

O particionamento de dados implica que o servidor de nome local não pode responder a todas as solicitações sem a ajuda de outros servidores de nome. Por exemplo, um servidor de nome no domínio *dcs.qmul.ac.uk* não seria capaz de fornecer o endereço IP de um computador no domínio *cs.purdue.edu*, a menos que estivesse armazenado em sua cache – certamente não na primeira vez que fosse solicitado.

O processo de localizar dados de atribuição de nomes dentre mais de um servidor para transformar um nome é chamado de *navegação*. O *software* cliente de resolução de nomes realiza a navegação em nome de um aplicativo cliente. Ele se comunica com servidores de nome, conforme for necessário, para transformar um nome. Ele pode ser fornecido como código de biblioteca e vinculado aos aplicativos clientes como, por exemplo, na implementação BIND do DNS (veja a Seção 13.2.3) ou no Grapevine [Birrell et al. 1982]. A alternativa usada no X500 é fornecer a resolução de nomes em um processo separado, compartilhado por todos os processos clientes nesse computador.

Um modelo de navegação suportado pelo DNS é conhecido como *navegação iterativa* (veja a Figura 13.2). Para transformar um nome, um cliente apresenta um nome ao servidor de nome local, o qual tenta resolvê-lo. Se o servidor de nome local tiver o nome, ele retornará o resultado imediatamente. Se não tiver, ele sugerirá outro servidor que poderá ajudar. A resolução prossegue no novo servidor. Esse procedimento é repetido, tantas vezes quanto necessário, até que o nome seja localizado ou se descubra que ele é desvinculado.

Como o DNS é projetado para conter entradas para milhões de domínios e é acessado por um número enorme de clientes, não seria possível ter todas as consultas come-



Um cliente entra em contato iterativamente com os servidores de nomes NS1-NS3 para resolver um nome.

Figura 13.2 Navegação iterativa.

çando em um servidor raiz, mesmo que ele fosse bastante replicado. O banco de dados DNS é partitionado entre os servidores de maneira a permitir que muitas consultas sejam atendidas de forma local e outras sejam atendidas sem necessidade de resolver cada parte do nome separadamente. O esquema de resolução de nomes do DNS será descrito com mais detalhes na Seção 13.2.3.

O NFS também emprega navegação iterativa na resolução de um nome de arquivo, componente por componente (veja o Capítulo 12). Isso porque o serviço de arquivo pode encontrar um vínculo simbólico ao resolver um nome. Um vínculo simbólico deve ser interpretado no espaço de nomes do sistema de arquivos do cliente, pois ele pode apontar para um arquivo em um diretório armazenado em outro servidor. O computador cliente deve determinar qual é esse servidor, pois somente o cliente conhece seus pontos de montagem.

Na *navegação por multicast*, um cliente envia o nome a ser resolvido e o tipo de objeto exigido para um grupo de servidores de nomes. Somente o servidor que contém os atributos nomeados responde à requisição. Infelizmente, entretanto, se o nome for desvinculado, a requisição será respondida com silêncio. Cheriton e Mann [1989] descrevem um esquema de navegação baseado em *multicast* no qual um servidor é incluído no grupo para responder quando o nome exigido for desvinculado.

Outra alternativa ao modelo de navegação iterativa é aquela em que um servidor de nome coordena a resolução do nome e devolve o resultado para o cliente. Ma [1992] distingue as *navegações controladas pelo servidor não recursiva e recursiva* (Figura 13.3). Na navegação não recursiva controlada pelo servidor, qualquer servidor de nome pode ser escolhido pelo cliente. Esse servidor se comunica por *multicast* ou iterativamente com seus pares, no estilo descrito anteriormente, como se fosse um cliente. Na navegação recursiva controlada pelo servidor, o cliente entra em contato com um único servidor. Se esse servidor não armazenar o nome, ele entra em contato com um de seus pares que armazena um prefixo (maior) do nome, o qual, por sua vez, tenta resolvê-lo. Esse procedimento continua recursivamente até que o nome seja resolvido.

Se um serviço de nomes abrange domínios administrativos distintos, então um cliente em execução em um domínio administrativo pode ser proibido de acessar servidores de nomes pertencentes a outro domínio. Além disso, até os servidores de nomes podem ser proibidos de descobrir a disposição dos dados de atribuição de nomes nos servidores de nomes de outro domínio administrativo. Então, tanto a navegação controlada pelo cliente como a navegação não recursiva controlada pelo servidor são inadequadas e deverá ser usada a navegação recursiva controlada pelo servidor. Servidores de nome

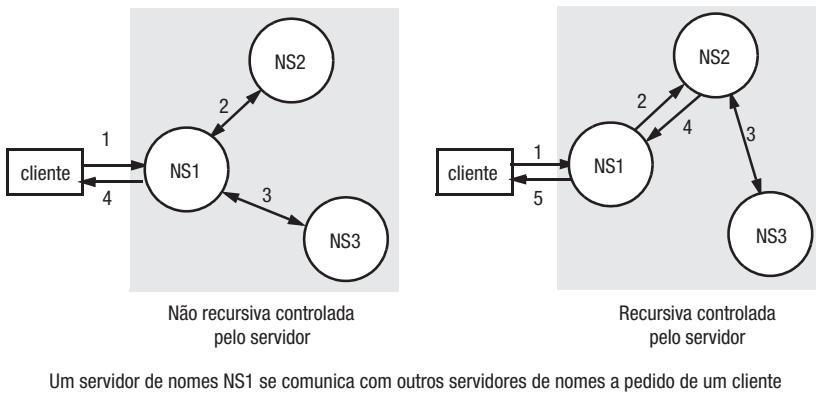


Figura 13.3 Navegação controlada pelo servidor não recursiva e recursiva.

autorizados solicitam os dados do serviço de nomes dos servidores de nome designados, gerenciados por diferentes administrações, o que retorna os atributos sem revelar onde estão armazenadas as diferentes partes do banco de dados de atribuição de nomes.

Uso de cache • No DNS e em outros serviços de nomes, o *software* cliente de resolução de nomes e os servidores mantêm uma cache com os resultados das resoluções de nomes anteriores. Quando um cliente solicita uma pesquisa de nome, o *software* de resolução de nomes consulta sua cache. Se contiver um resultado recente de uma pesquisa anterior do nome, ele o retornará para o cliente; caso contrário, começará a procurá-lo em um servidor. Esse servidor, por sua vez, pode retornar dados armazenados na cache de outros servidores.

O uso da cache é importante para o desempenho do serviço de nomes e ajuda na manutenção da disponibilidade tanto do serviço de nomes como de outros serviços, a despeito de falhas do servidor de nome. Sua função na melhoria dos tempos de resposta, economizando na comunicação com servidores de nome, é clara. A cache pode ser utilizada para eliminar servidores de nome de alto nível – o servidor raiz, em particular – do caminho de navegação, permitindo que a resolução prossiga, apesar de algumas falhas de servidor.

O uso de cache por resolvedores de nome clientes é amplamente aplicado nos serviços de nomes e é particularmente bem-sucedido, pois os dados de atribuição de nomes são alterados de forma relativamente rara. Por exemplo, é provável que informações como endereços de computadores ou de serviços permaneçam inalteradas por vários meses ou anos. Entretanto, existe a possibilidade de um serviço de nomes retornar atributos desatualizados, por exemplo, um endereço obsoleto, durante a resolução.

13.2.3 O Domain Name System

O Domain Name System é um projeto de serviço de nomes cujo banco de dados de atribuição de nomes é usado na Internet. Ele foi planejado principalmente por Mockapetris e especificado na RFC 1034 [1987] e na RFC 1035. O DNS substituiu o esquema de atribuição de nomes original da Internet, no qual todos os nomes e endereços eram mantidos em um único arquivo mestre central e carregados por *download*, via FTP, em todos os computadores que deles necessitassem [Harrenstien *et al.* 1985]. Logo se viu que esse esquema original sofria de três defeitos importantes:

- Ele não tinha capacidade de escalabilidade para suportar grandes números de computadores.

- Organizações locais desejavam administrar seus próprios sistemas de atribuição de nomes.
- Era necessário um serviço de nomes geral – e não um que servisse apenas para pesquisar endereços de computadores.

Os objetos nomeados pelo DNS são principalmente computadores – para os quais, basicamente, os endereços IP são armazenados como atributos – e o que nos referimos neste capítulo como domínios de atribuição de nomes são chamados simplesmente de *domínios* no DNS. Em princípio, entretanto, qualquer tipo de objeto pode ser nomeado, e sua arquitetura fornece abrangência para uma variedade de implementações. As organizações e os departamentos dentro delas podem gerenciar seus próprios dados de atribuição de nomes. Na Internet, milhões de nomes são vinculados ao DNS e nele pesquisados. Qualquer nome pode ser resolvido por qualquer cliente. Isso é obtido pelo particionamento hierárquico do banco de dados de nomes, pela replicação dos dados de atribuição de nomes e pelo uso de cache.

Nomes de domínio • O DNS é projetado para ser usado em várias implementações, cada uma das quais podendo ter seu próprio espaço de nomes. Na prática, entretanto, somente um é amplamente usado e é o de atribuição de nomes da Internet. O espaço de nomes DNS da Internet é particionado tanto de forma organizacional como geográfica. Os nomes estão escritos com o domínio de nível mais alto à direita. Os domínios organizacionais de nível superior (também chamados de *domínios genéricos* ou *top-level domain*) usados na Internet, originalmente, eram:

<i>com</i>	–	Organizações comerciais
<i>edu</i>	–	Universidades e outras instituições educacionais
<i>gov</i>	–	Órgãos do governo norte-americano
<i>mil</i>	–	Organizações militares dos EUA
<i>net</i>	–	Principais centros de suporte à rede
<i>org</i>	–	Organizações não mencionadas anteriormente
<i>int</i>	–	Organizações internacionais

Novos domínios de nível superior, como *biz* e *mobi*, foram adicionados no início dos anos 2000. Uma lista completa dos nomes de domínio genéricos atuais está disponível no Internet Assigned Numbers Authority no URL [www.iana.org I].

Além disso, cada país tem seus próprios domínios:

<i>us</i>	–	Estados Unidos
<i>uk</i>	–	Reino Unido
<i>fr</i>	–	França
<i>br</i>	–	Brasil
...	–	...

Os países, particularmente outros que não os EUA, frequentemente usam seus próprios subdomínios para distinguir suas organizações. O Reino Unido, por exemplo, tem os domínios *co.uk* e *ac.uk*, que correspondem a *com* e *edu* respectivamente (*ac* significa *academic community* – comunidade acadêmica).

Note que, apesar de seu sufixo de caráter geográfico *uk*, um domínio como *doit.co.uk* poderia ter dados se referindo a computadores localizados no escritório espanhol da Doit Ltd., uma empresa britânica fictícia. Em outras palavras, mesmo os nomes de domínio de caráter geográfico são convencionais e completamente independentes de suas localizações físicas.

Consultas DNS • O DNS é usado na Internet principalmente para a resolução de nomes de computador e para pesquisar servidores de correio eletrônico, como segue:

Resolução de nomes de computador: em geral, os aplicativos usam o DNS para transformar nomes de computador em endereços IP. Por exemplo, quando um navegador Web recebe um URL contendo o nome de domínio *www.dcs.qmul.ac.uk*, ele faz uma solicitação de DNS e obtém o endereço IP correspondente. Conforme foi apontado no Capítulo 4, os navegadores usam HTTP para se comunicar com o servidor Web no endereço IP dado, usando um número de porta reservado, caso não seja especificado nenhum no URL. Os serviços FTP e SMTP funcionam de maneira semelhante; por exemplo, um programa FTP pode receber o nome de domínio *ftp.dcs.qmul.ac.uk*, fazer uma solicitação de DNS para obter seu endereço IP e, depois, usar TCP para se comunicar com ele no número de porta reservado.

Localização de servidores de correio eletrônico: o *software* de correio eletrônico usa o DNS para resolver nomes de domínio para endereços IP de servidores de correio eletrônico – computadores que aceitam correspondência eletrônica para esses domínios. Por exemplo, quando o endereço *tom@dcs.rnx.ac.uk* precisa ser resolvido, o DNS é consultado com o endereço *dcs.rnx.ac.uk* e a designação de tipo “*mail*”. Ele retorna uma lista de nomes de domínio de servidores que podem aceitar e-mail destinados a *dcs.rnx.ac.uk*, se existir (e, opcionalmente, os endereços IP correspondentes). O DNS pode retornar mais de um nome de domínio para que o *software* de correio eletrônico possa tentar alternativas, caso o servidor principal esteja inacessível por algum motivo. O DNS retorna um valor de preferência (inteiro) para cada servidor de correio eletrônico, indicando a ordem em que os servidores devem ser tentados.

Outros tipos de consulta são implementadas em algumas instalações, mas são utilizadas com menos frequência do que as que acabamos de mencionar:

Resolução reversa: alguns programas de *software* exigem que, dado um endereço IP, um nome de domínio seja retornado. Isso é apenas o inverso da consulta de nome de computador normal, mas o servidor de nomes que recebe a consulta só responde se o endereço IP estiver em seu próprio domínio.

Informações sobre computadores: o DNS pode armazenar o tipo de arquitetura da máquina e o sistema operacional relacionados aos nomes de domínio dos computadores. Sugeriu-se que essa opção não fosse usada publicamente, pois fornece informações úteis para quem está tentando obter acesso não autorizado aos computadores.

Em princípio, o DNS pode ser usado para armazenar atributos arbitrários. Uma consulta é especificada por um nome de domínio, classe e tipo. Para nomes de domínio na Internet, a classe é IN. O tipo de consulta específica se um endereço IP, um servidor de correio eletrônico, um servidor de nome ou algum outro tipo de informação é exigida. Existe um domínio especial, *in-addr.arpa*, para conter endereços IP para as pesquisas reversas. O atributo da classe é usado para distinguir, por exemplo, o banco de dados de atribuição de nomes da Internet de outros bancos de dados de atribuição de nomes DNS experimentais. É definido um conjunto de tipos para determinado banco de dados; os que servem para banco de dados da Internet aparecem na Figura 13.5.

Servidores de nome DNS • O problema de escalabilidade é tratado por uma combinação do particionamento do banco de dados de atribuição de nomes e a replicação e armazenamento em cache de partes dele, próximo dos pontos onde ele é acessado. O banco de dados DNS é distribuído em uma rede lógica de servidores. Cada servidor contém parte do banco de dados de atribuição de nomes – principalmente dados do domínio local. As

consultas que dizem respeito aos computadores do domínio local são atendidas pelos servidores de dentro desse domínio. Entretanto, cada servidor registra os nomes de domínio e endereços de outros servidores de nome para que as consultas pertinentes a objetos de fora do domínio possam ser atendidas.

Os dados de atribuição de nomes DNS são divididos em zonas. Uma zona contém os seguintes dados:

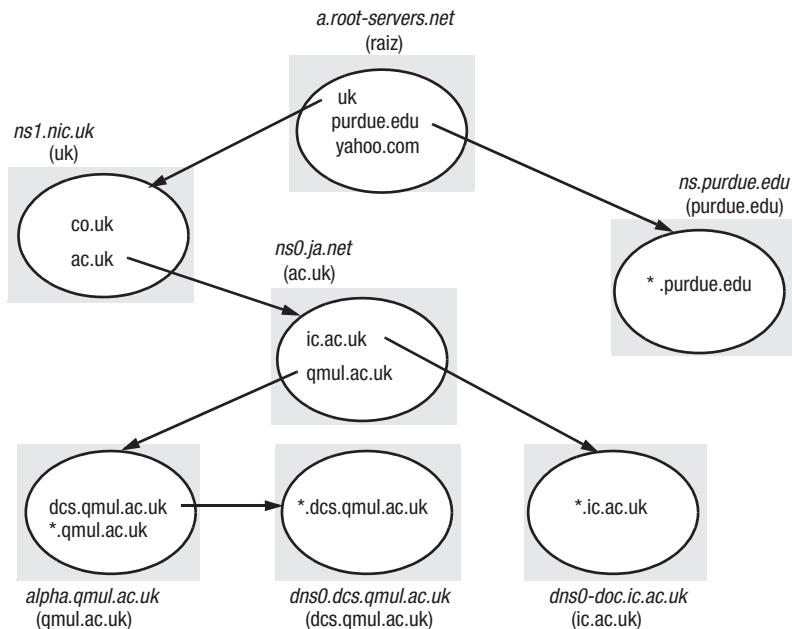
- Dados de atributo de nomes em um domínio, menos os subdomínios administrados por autoridades de nível mais baixo. Por exemplo, uma zona poderia conter dados do Queen Mary, Universidade de Londres – *qmul.ac.uk* – menos os dados mantidos pelos departamentos, por exemplo, o Departamento da Ciência da Computação – *dcs.qmul.ac.uk*.
- Os nomes e endereços de pelo menos dois servidores de nome que possuem autoridade sobre dados da zona. São versões de dados de zona que podem ser considerados razoavelmente atualizados.
- Os nomes de servidores de nome que contêm autoridade sobre dados de subdomínios delegados; e dados “de cola”, fornecendo os endereços IP desses servidores.
- Parâmetros de gerenciamento de zona, como aqueles que governam o uso da cache e a replicação de dados de zona.

Um servidor pode ter autoridade sobre dados de zero ou mais zonas. Para que os dados de atribuição de nomes estejam disponíveis, mesmo quando um único servidor falha, a arquitetura do DNS especifica que cada zona deve ser replicada de forma que pelo menos dois servidores sejam autoridades sobre seus dados.

Os administradores de sistema inserem os dados de uma zona em um arquivo mestre, o qual armazena os dados daquela zona. Existem dois tipos de servidor que são considerados como autoridade de dados. Um *servidor principal*, ou *mestre*, lê dados de zona diretamente de um arquivo mestre local. Os *servidores secundários* fazem o *download* dos dados de zona de um servidor principal. Estes se comunicam periodicamente com o servidor principal para verificar se sua versão armazenada corresponde àquela mantida pelo servidor principal. Se a cópia de um secundário estiver desatualizada, o principal enviará a ele a versão mais recente. A frequência da verificação do secundário é configurada pelos administradores como um parâmetro da zona e seu valor normalmente é uma ou duas vezes por dia.

Qualquer servidor está livre para armazenar em *cache* dados de outros servidores, para evitar a necessidade de entrar em contato com eles quando a resolução de nomes exigir os mesmos dados novamente. Nesse caso, o servidor indica nas respostas enviadas aos clientes, quando fornecidas, de que ele é um servidor não autoridade sobre os dados. Cada entrada em uma zona tem um valor de tempo de vida. Quando um servidor não autoridade armazena dados na cache provenientes de um servidor com autoridade, ele anota o tempo de vida. Ele só fornecerá para os clientes os dados de sua cache durante esse tempo; quando consultado após o período de tempo ter expirado, ele entra em contato novamente com o servidor autoridade para verificar seus dados. Essa é uma característica útil que minimiza o volume do tráfego da rede, enquanto mantém a flexibilidade para os administradores de sistema. Quando a expectativa é de que os atributos sejam raramente alterados, eles podem receber um tempo de vida grande. Se um administrador souber que os atributos provavelmente mudarão logo, poderá reduzir o tempo de vida de acordo com isso.

A Figura 13.4 mostra a organização de parte do banco de dados DNS, conforme se encontrava no ano de 2001. Esse exemplo continua válido hoje, mesmo que alguns dos dados tenham se alterado com a reconfiguração dos sistemas ao longo do tempo. Note que, na prática, os servidores raízes, como *a.root-servers.net*, contêm entradas para vários níveis de domínio, assim como entradas para nomes de domínio de primeiro nível.



Nota: Os nomes de servidor de nomes aparecem em itálico e os domínios correspondentes, entre parênteses. As setas denotam entradas no servidor de nome.

Figura 13.4 Servidores de nome DNS.

Isso serve para reduzir o número de etapas de navegação, quando os nomes de domínio são resolvidos. Os servidores de nome raiz mantêm entradas com autoridade de dados para os servidores de nome dos domínios de nível superior (*top-level domain*). Os servidores raízes também são servidores de nome autoridade para os domínios de nível superior genéricos, como *com* e *edu*. Entretanto, os servidores de nome raízes não são servidores de nome dos domínios de país. Por exemplo, o domínio *uk* tem um conjunto de servidores de nome, um dos quais era chamado *ns1.nic.net*. Esses servidores de nome conhecem os servidores de nome dos domínios de segundo nível no Reino Unido, como *ac.uk* e *co.uk*. Os servidores de nome do domínio *ac.uk* conhecem os servidores de nome de todos os domínios de universidade daquele país, como *qmul.ac.uk* ou *ic.ac.uk*. Em alguns casos, um domínio de universidade delega parte de suas responsabilidades para um subdomínio, como *dcs.qmul.ac.uk*.

As informações do domínio raiz são replicadas por um servidor principal em um conjunto de servidores secundários, conforme descrito anteriormente. Apesar disso, os servidores raízes atendem a 1.000 ou mais consultas por segundo. Todos os servidores de DNS armazenam os endereços de um ou mais servidores de nome raízes, os quais não mudam com muita frequência. Normalmente, eles também armazenam o endereço de um servidor autorizado do domínio pai. Uma consulta envolvendo um nome de domínio de três componentes, como *www.berkeley.edu*, pode ser atendida usando-se, na pior das hipóteses, duas etapas de navegação: uma para um servidor raiz que armazena uma entrada de servidor de nome apropriada e uma segunda para o servidor cujo nome é retornado.

Com referência à Figura 13.4, o nome de domínio *jeans-pc.dcs.qmul.ac.uk* pode ser pesquisado dentro de *dcs.qmul.ac.uk*, usando-se o servidor local *dns0.dcs.qmul.ac.uk*.

<i>Tipo de registro</i>	<i>Significado</i>	<i>Conteúdo principal</i>
<i>A</i>	Endereço de computador (IPv4)	Número IPv4
<i>AAAA</i>	Endereço de computador (IPv6)	Número IPv6
<i>NS</i>	Servidor de nome autoridade	Nome de domínio do servidor
<i>CNAME</i>	Nome canônico de um <i>alias</i>	Nome de domínio do <i>alias</i>
<i>SOA</i>	Marca o início dos dados de uma zona	Parâmetros que governam a zona
<i>PTR</i>	Ponteiro de nome de domínio (pesquisas reversas)	Nome de domínio

Figura 13.5 Registros de recurso DNS.

Esse servidor não armazena uma entrada para o servidor Web *www.ic.ac.uk*, mas mantém uma entrada na cache para *ic.ac.uk* (que é obtida do servidor com autoridade *ns0.ja.net*). O servidor *dns0-doc.ic.ac.uk* pode ser contatado para transformar o nome completo.

Navegação e processamento de consulta • Um cliente de DNS é chamado de *resolvedor*. Normalmente, ele é implementado como *software* de biblioteca. Ele aceita consultas, formata-as nas mensagens esperadas no protocolo DNS e se comunica com um ou mais servidores de nome para atender às pesquisas. É usado um protocolo de requisição-resposta simples, normalmente utilizando pacotes UDP na Internet (os servidores de DNS usam um número de porta conhecido). Caso a resposta não venha dentro de certo tempo (*timeout*), a consulta é enviada novamente, se necessário. O resolvidor pode ser configurado para entrar em contato com uma lista de servidores de nome iniciais, em ordem de preferência, para o caso de um ou mais estarem indisponíveis.

A arquitetura do DNS possibilita navegação recursiva, assim como navegação iterativa. O resolvidor especifica qual tipo de navegação é exigido, ao entrar em contato com um servidor de nome. Entretanto, os servidores de nome não são obrigados a implementar navegação recursiva. Conforme mencionado anteriormente, a navegação recursiva pode manter ocupadas as *threads* do servidor, significando que outras requisições podem ser retardadas.

Para economizar comunicação na rede, o protocolo DNS permite que várias consultas sejam empacotadas na mesma mensagem de requisição e que os servidores de nome enviem várias respostas correspondentes em suas mensagens de resposta.

Registros de recurso • Os dados de zona são armazenados pelos servidores de nome em arquivo, em um de vários tipos de registro de recurso. Para o banco de dados da Internet, isso inclui os tipos que aparecem na Figura 13.5. Cada registro se refere a um nome de domínio, o qual não é mostrado. As entradas da tabela se referem aos itens já mencionados, exceto que os registros *AAAA* armazenam endereços IPv6, enquanto os registros *A* armazenam endereços IPv4, e as entradas *TXT* foram incluídas para permitir que outras informações arbitrárias sejam armazenadas com relação aos nomes de domínio.

Os dados de uma zona começam com um registro de tipo *SOA*, o qual contém os parâmetros de zona que especificam, por exemplo, o número da versão e com que frequência os servidores secundários devem atualizar suas cópias. Isso é seguido por uma lista de registros de tipo *NS*, especificando os servidores de nome do domínio, e por uma lista de registros de tipo *MX*, fornecendo os nomes de domínio dos servidores de correio eletrônico, cada um prefixado por um número expressando sua preferência. Por exemplo, parte do banco de dados do domínio *dcs.qmul.ac.uk* aparece na Figura 13.6, em que o tempo de vida *ID* significa 1 dia.

nome de domínio	tempo de vida	classe	tipo	valor
<i>dcs.qmul.ac.uk</i>	1D	IN	NS	<i>dns0</i>
<i>dcs.qmul.ac.uk</i>	1D	IN	NS	<i>dns1</i>
<i>dcs.qmul.ac.uk</i>	1D	IN	MX	1 <i>mail1.qmul.ac.uk</i>
<i>dcs.qmul.ac.uk</i>	1D	IN	MX	2 <i>mail2.qmul.ac.uk</i>

Figura 13.6 Registros de dados de zona DNS.

Na sequência do banco de dados, os registros de tipo A fornecem os endereços IP dos dois servidores de nome *dns0* e *dns1*. Os endereços IP dos servidores de correio eletrônico e do terceiro servidor de nome são dados no banco de dados correspondente aos seus domínios.

A maior parte dos registros em uma zona de nível inferior, como *dcs.qmul.ac.uk*, será de tipo A e fará o mapeamento do nome de domínio de um computador para seu endereço IP. Eles podem conter alguns *alias* para os serviços conhecidos, por exemplo:

nome de domínio	tempo de vida	classe	tipo	valor
<i>www</i>	1D	IN	CNAME	<i>traffic</i>
<i>traffic</i>	1D	IN	A	138.37.95.150

Se o domínio tiver subdomínios, podem existir mais registros de tipo NS especificando seus servidores de nome, os quais também terão entradas A individuais. Por exemplo, em um ponto, o banco de dados de *qmul.ac.uk* continha os seguintes registros para os servidores de nome em seu subdomínio *dcs.qmul.ac.uk*:

nome de domínio	tempo de vida	classe	tipo	valor
<i>dcs</i>	1D	IN	NS	<i>dns0.dcs</i>
<i>dns0.dcs</i>	1D	IN	A	138.37.88.249
<i>dcs</i>	1D	IN	NS	<i>dns1.dcs</i>
<i>dns1.dcs</i>	1D	IN	A	138.37.94.248

Compartilhamento de carga de servidores de nome: em alguns *sites*, os serviços muito utilizados, como Web e FTP, são suportados por um grupo de computadores na mesma rede. Neste caso, o mesmo nome de domínio é usado para cada membro do grupo. Quando um nome de domínio é compartilhado por vários computadores, existe um registro para cada computador do grupo fornecendo seu endereço IP. Por padrão, o servidor de nome responde às consultas para as quais vários registros correspondem ao nome solicitado, retornando os endereços IP de acordo com um programa de rodízio. Sucessivos clientes recebem acesso a diferentes servidores para que estes possam compartilhar a carga de trabalho. O uso da cache tem o potencial de estragar esse esquema, pois uma vez que um servidor de nome não autoridade, ou um cliente, tenha o endereço do servidor em sua cache, ele continuará a usá-lo. Para neutralizar esse efeito, os registros recebem um tempo de vida curto.

A implementação BIND do DNS • O BIND (Berkeley Internet Name Domain) é uma implementação do DNS para computadores que executam UNIX. Os programas clientes

que executam resolução de nomes devem ser ligados a essa biblioteca. Os computadores servidores de nome DNS executam o *daemon* denominado de *named*.

O BIND permite três categorias de servidor de nome: servidores principais, servidores secundários e servidores somente de cache; o *named* implementa apenas um desses tipos, de acordo com o conteúdo de um arquivo de configuração. As duas primeiras categorias são conforme descrito anteriormente. Os servidores somente de cache leem, em um arquivo de configuração, nomes e endereços de servidores com autoridades suficientes para resolver qualquer nome. Depois disso, eles apenas armazenam esses dados e os dados que aprendem resolvendo nomes para clientes.

Uma organização típica tem um servidor principal, com um ou mais servidores secundários, que fornecem nomes atendendo a diferentes redes locais no *site*. Além disso, computadores individuais frequentemente executam seus próprios servidores somente de cache para reduzir o tráfego da rede e acelerar ainda mais os tempos de resposta.

Discussão sobre o DNS • A implementação DNS na Internet atinge tempos de resposta médios relativamente curtos para resolução de nomes, considerando o volume de dados de atribuição de nomes e a escala das redes envolvidas. Vimos que ela obtém isso por meio de uma combinação de particionamento, replicação e uso de cache para os dados de atribuição de nomes. Os objetos nomeados são principalmente computadores, servidores de nome e de correio eletrônico. Os mapeamentos de nome de computador para endereço IP mudam de forma relativamente rara, assim como as identidades dos servidores de nome e de correio eletrônico; portanto, o uso de cache e a replicação ocorrem em um ambiente relativamente estável.

O DNS permite que os dados de atribuição de nomes se tornem inconsistentes. Isto é, se os dados de atribuição de nomes forem alterados, outros servidores poderão fornecer dados antigos aos clientes por períodos na ordem de dias. Nenhuma das técnicas de replicação exploradas no Capítulo 18 é aplicada. Entretanto, a inconsistência não tem nenhuma consequência até o momento em que um cliente tenta utilizar dados antigos. O DNS não trata, por si mesmo, o modo como os endereços antigos são detectados.

Além dos computadores, o DNS também nomeia um tipo de serviço em particular: o serviço de correio eletrônico, um para cada domínio. O DNS presume que há apenas um serviço de correio eletrônico por domínio endereçado; portanto, os usuários não precisam incluir o nome desse serviço explicitamente. Os aplicativos de correio eletrônico selecionam esse serviço de forma transparente, usando o tipo de consulta apropriado ao entrar em contato com os servidores de DNS.

Em resumo, o DNS armazena uma variedade limitada de dados de atribuição de nomes, mas isso é suficiente, na medida em que os aplicativos, como o correio eletrônico, impõem seus próprios esquemas de atribuição de nomes sobre os nomes de domínio. Poderia argumentar-se que o banco de dados DNS representa o mínimo denominador comum do que seria considerado útil por muitas comunidades de usuários na Internet. O DNS não foi projetado para ser o único serviço de nomes da Internet; ele coexiste com serviços de diretório e de nome locais que armazenam os dados mais pertinentes às necessidades locais (como o Network Information Service, da Sun, que armazena senhas codificadas, por exemplo, ou o Active Directory Service, da Microsoft [www.microsoft.com]), que armazena informações detalhadas sobre todos os recursos dentro de um domínio).

O que permanece como um problema em potencial para o projeto DNS é sua rigidez com relação às alterações na estrutura do espaço de nomes e a falta de capacidade de personalizar o espaço de nomes de acordo com as necessidades locais. Esses aspectos do projeto de atribuição de nomes são considerados pelo estudo de caso do serviço de nomes global, na Seção 13.4. Antes disso, consideraremos os serviços de diretório.

13.3 Serviços de diretório

Descrevemos como os serviços de nome armazenam conjuntos de pares *<nome, atributo>* e como os atributos são pesquisados a partir de um nome. É natural considerar a dualidade dessa organização, na qual os *atributos* são utilizados como os valores a serem pesquisados. Nesses serviços, os nomes textuais podem ser considerados apenas como outro atributo. Às vezes, os usuários desejam encontrar uma pessoa, ou um recurso em particular, mas não sabem seu nome, apenas alguns de seus outros atributos. Por exemplo, um usuário pode perguntar: “qual é o nome do usuário com número de telefone 020-555 9980?”. De maneira semelhante, os usuários exigem um serviço, mas não estão preocupados com qual entidade do sistema fornece esse serviço, desde que ele esteja convenientemente acessível. Por exemplo, um usuário poderia perguntar “quais computadores neste prédio são Macintosh executando o sistema operacional Mac OS X?” ou “onde posso imprimir uma imagem colorida de alta resolução?”.

Um serviço que armazena conjuntos de vínculos entre nomes e atributos e que pesquisa entradas que correspondem a especificações baseadas no atributo é chamado de *serviço de diretório*. Exemplos são o Active Directory Services, da Microsoft, o X.500 e seu primo LDAP (descrito na Seção 13.5), o Univers [Bowman *et al.* 1990] e o Profile [Peterson 1988]. Às vezes, os serviços de diretório são chamados de *serviços de páginas amarelas* e os serviços de nomes convencionais são chamados, correspondentemente, de *serviços de páginas brancas*, em uma analogia com os tipos tradicionais de catálogos telefônicos. Os serviços de diretório também são conhecidos como *serviços de nomes baseados em atributo*.

Um serviço de diretório retorna os conjuntos de atributos de todos os objetos encontrados que correspondam a alguns atributos especificados. Portanto, o pedido ‘Telephone-Number = 020-555 9980’ poderia retornar {‘Name = John Smith’, ‘TelephoneNumber = 020-555 9980’, ‘emailAddress = john@dcs.gormenghast.ac.uk’, ...}, por exemplo. O cliente pode especificar que apenas um subconjunto dos atributos é de interesse – por exemplo, apenas os endereços de *e-mail* dos objetos correspondentes. O X.500, e alguns outros serviços de diretório, também permitem que objetos sejam pesquisados por nomes textuais hierárquicos convencionais. O UDDI (Universal Directory and Discovery Service), que foi apresentado na Seção 9.4, apresenta serviços de páginas brancas e de páginas amarelas para fornecer informações sobre organizações e os serviços Web de que dispõe.

UDDI à parte, o termo *serviço de descoberta* normalmente denota o caso especial de um serviço de diretório para serviços fornecidos por dispositivos em um ambiente de interligação em rede espontânea. Conforme a Seção 1.3.2, os dispositivos nas redes espontâneas estão sujeitos a se conectar e se desconectar de forma imprevisível. Uma diferença fundamental entre um serviço de descoberta e serviços de diretório é que o endereço de um serviço de diretório normalmente é conhecido e previamente configurado nos clientes, enquanto um dispositivo que entra em um ambiente de interligação em rede espontânea precisa contar com a comunicação por *multicast*, pelo menos na primeira vez que acessa o serviço de descoberta local. A Seção 19.2.1 descreve os serviços de descoberta em detalhes.

Os atributos são claramente mais poderosos do que os nomes, como designadores de objetos: podem ser escritos programas para selecionar objetos de acordo com especificações de atributo precisas, em que os nomes podem não ser conhecidos. Outra vantagem dos atributos é que eles não expõem a estrutura das organizações para o mundo exterior, como acontece com os nomes partionados em termos de organização. Entretanto, para muitas aplicações, a relativa simplicidade de uso dos nomes textuais torna improvável que eles sejam substituídos pela atribuição de nomes baseada em atributos.

13.4 Estudo de caso: Global Name Service

O GNS (Global Name Service) foi projetado e implementado por Lampson e colegas, no DEC Systems Research Center [Lampson 1986], para fornecer facilidades para a localização de recursos, endereços de *e-mail* e autenticação. Os objetivos de projeto do GNS já foram listados no final da Seção 13.1; eles refletem o fato de que um serviço de nomes para uso em uma rede interligada deve suportar um banco de dados de atribuição de nomes que possa ser ampliado para incluir os nomes de milhões de computadores e (eventualmente) endereços de *e-mail* de bilhões de usuários. Os projetistas do GNS também consideraram que o banco de dados de atribuição de nomes provavelmente terá um tempo de vida longo, e que ele deve continuar a operar eficientemente, enquanto cresce de pequena para grande escala, e enquanto a rede em que é baseado também evolui. A estrutura do espaço de nomes pode mudar durante esse tempo, para refletir alterações nas estruturas organizacionais. O serviço deve acomodar mudanças nos nomes dos indivíduos, das organizações e dos grupos que contém e as alterações na estrutura de atribuição de nomes, como as que ocorrem quando uma empresa é assumida por outra. Nesta descrição, vamos abordar as características do projeto que permitem acomodar tais alterações.

O banco de dados de atribuição de nomes potencialmente grande e a escala do ambiente distribuído em que o GNS é feito para operar tornam fundamental o uso de cache. Entretanto, fica extremamente difícil manter a consistência completa entre todas as cópias de uma entrada do banco de dados. A estratégia de consistência de cache adotada conta com a suposição de que as atualizações no banco de dados serão raras e que uma disseminação lenta das atualizações é aceitável, pois os clientes podem detectar e se recuperar do uso de dados de atribuição de nomes desatualizados.

O GNS gerencia um banco de dados de atribuição de nomes composto de uma árvore de diretórios contendo nomes e valores. Os diretórios são nomeados por nomes de caminho de vários componentes que se referem a uma raiz, ou relativos a um diretório de trabalho, de forma muito parecida com os nomes de arquivo em um sistema de arquivos UNIX. Cada diretório também recebe um valor inteiro, que serve como *identificador de diretório* (DI) exclusivo. Nesta seção, usaremos nomes em itálico ao nos referirmos ao DI de um diretório, de modo que *CE* é o identificador do diretório CE (Comunidade Europeia). Um diretório contém uma lista de nomes e referências. Os valores armazenados nas folhas da árvore de diretório são organizados em *árvores de valores*, para que os atributos associados aos nomes possam ser valores estruturados.

No GNS, os nomes têm duas partes: *<nome do diretório, nome do valor>*. A primeira parte identifica um diretório; a segunda se refere a uma árvore de valores, ou a alguma parte de uma árvore de valores. Por exemplo, veja a Figura 13.7, na qual os DIs estão ilustrados como valores inteiros pequenos, embora, na verdade, eles sejam escolhidos em um intervalo de valores inteiros maiores para garantir a exclusividade. Os atributos de um usuário Peter.Smith no diretório QMUL seriam armazenados na árvore de valores nomeada *<CE/UK/AC/QMUL, Peter.Smith>*. A árvore de valores inclui uma senha, a qual pode ser referenciada como *<CE/UK/AC/QMUL, Peter.Smith/senha>*, e vários endereços de correio eletrônico, cada um dos quais seria listado na árvore de valores como um único nó, com o nome *<CE/UK/AC/QMUL, Peter.Smith/caixas_de_correio>*.

A árvore de diretório é particionada e armazenada em muitos servidores, com cada partição replicada em vários servidores. A consistência da árvore é mantida em face de duas ou mais atualizações concorrentes – por exemplo, dois usuários podem tentar criar, simultaneamente, entradas com o mesmo nome, e apenas um deve ter êxito. Os diretórios replicados apresentam um segundo problema de consistência; este é tratado por um

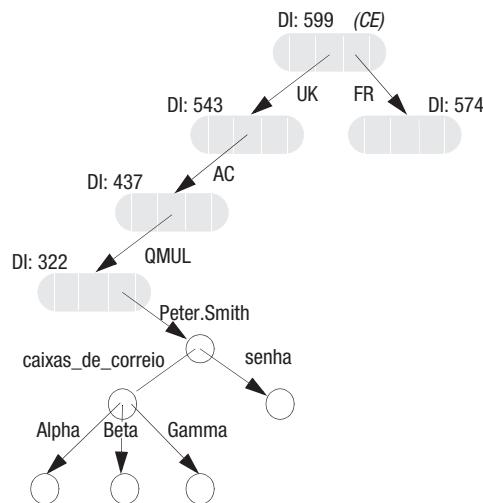


Figura 13.7 Árvore de diretório GNS e árvore de valores do usuário Peter.Smith.

algoritmo de distribuição de atualização assíncrono que garante a consistência final, mas sem garantia de que todas as cópias sejam sempre atuais. Esse nível de consistência é considerado satisfatório para o objetivo.

Acomodação de alterações • Veremos agora os aspectos do projeto relacionados ao ajuste do crescimento e da mudança na estrutura do banco de dados de atribuição de nomes. No nível dos clientes e administradores, o crescimento é acomodado por meio da ampliação da árvore de diretório da maneira usual. No entanto, podemos querer integrar as árvores de atribuição de nomes de dois serviços GNS anteriormente separados. Por exemplo, como poderíamos integrar o banco de dados cuja raiz está no diretório *CE* mostrado na Figura 13.7, com outro banco de dados de *AMÉRICA DO NORTE*? A Figura 13.8 mostra uma nova raiz *MUNDO*, introduzida acima das raízes existentes das duas árvores a serem integradas. Essa é uma técnica simples, mas como ela afeta os clientes que continuam a usar nomes que são referenciados ao que era “a raiz”, antes que a integração ocorresse? Por exemplo, </UK/AC/QMUL, Peter.Smith> era um nome usado pelos clientes antes da integração. Trata-se de um nome absoluto (pois começa com o símbolo da raiz /), mas a raiz a que ele se refere é *CE* e não *MUNDO*. *CE* e *AMÉRICA DO NORTE* são *raízes de trabalho* – contextos iniciais nos quais os nomes que começam com a raiz / devem ser pesquisados.

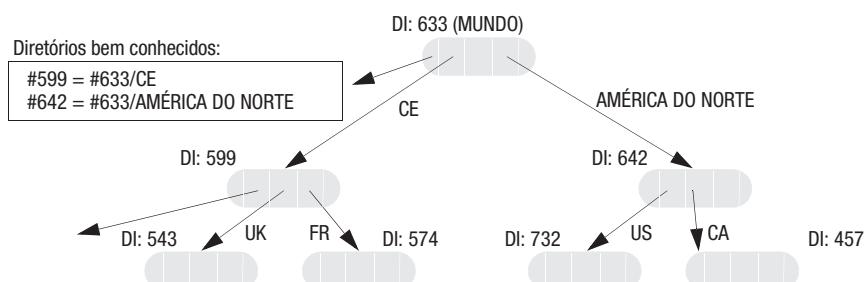


Figura 13.8 Integração de árvores sob uma nova raiz.

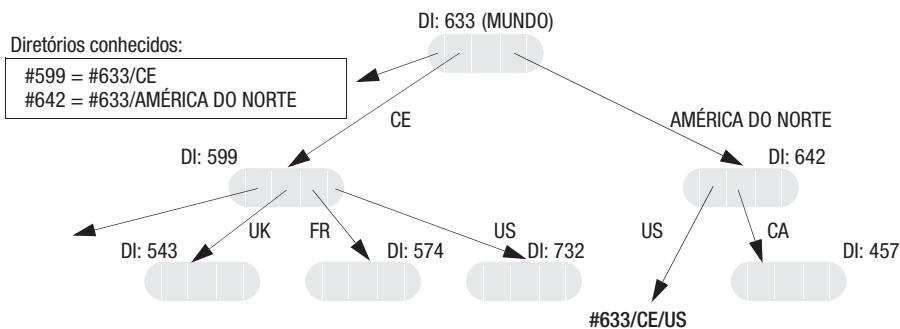


Figura 13.9 Reestruturação do diretório.

A existência de identificadores de diretório exclusivos pode ser usada para resolver esse problema. A raiz de trabalho de cada programa deve ser identificada como uma parte de seu ambiente de execução (de forma muito parecida com o que é feito para o diretório corrente de um programa). Quando um cliente na Comunidade Europeia usa um nome da forma </UK/AC/QMUL, Peter.Smith>, seu agente de usuário local, que sabe da existência da raiz de trabalho, prefixa o identificador de diretório CE (599), produzindo, assim, o nome <#599/UK/AC/QMUL, Peter.Smith>. O agente de usuário passa esse nome derivado na requisição de pesquisa para um servidor GNS. O agente de usuário pode lidar de modo semelhante com nomes relativos que se referem a diretórios de trabalho. Os clientes que sabem da existência da nova configuração também podem fornecer nomes absolutos para o servidor GNS, os quais se referem ao diretório conceitual super-raiz, contendo todos os identificadores de diretório; por exemplo, <MUNDO/EC/UK/AC/QMUL, Peter.Smith>, mas o projeto não pode supor que todos os clientes serão atualizados, para levar em conta tal alteração.

A técnica descrita anteriormente resolve o problema lógico, permitindo que usuários e programas clientes continuem a usar nomes definidos em relação a uma raiz antiga, mesmo quando uma nova raiz real é inserida, mas ela deixa um problema de implementação: em um banco de dados de atribuição de nomes distribuído que pode conter milhões de diretórios, como o serviço GNS pode localizar um diretório, dado apenas seu identificador, como #599? A solução adotada pelo GNS é listar os diretórios usados como raízes de trabalho, como CE, em uma tabela de “diretórios bem conhecidos” mantida no diretório-raiz real corrente do banco de dados de atribuição de nomes. Quando a raiz real do banco de dados de atribuição de nomes muda, como na Figura 13.8, todos os servidores GNS são informados da nova localização da raiz real. Então, eles podem interpretar normalmente nomes da forma MUNDO/CE/UK/AC/QMUL (referido à raiz real) e interpretar nomes da forma #599/UK/AC/QMUL usando a tabela de “diretórios bem conhecidos” para transformá-los nos nomes de caminho completos que começam na raiz real.

O GNS também suporta a reestruturação do banco de dados para acomodar mudança organizacional. Suponha que os Estados Unidos se tornem parte da Comunidade Europeia(!). A Figura 13.9 mostra a nova árvore de diretório. No entanto, se a subárvore US for simplesmente movida para o diretório CE, os nomes que começam com MUNDO/AMÉRICA DO NORTE/US não funcionarão mais. A solução adotada pelo GNS foi inserir um “vínculo simbólico” no lugar da entrada US original (mostrada em negrito na Figura 13.9). O procedimento de pesquisa de diretório do GNS interpreta o vínculo como um redirecionamento para o diretório US em sua nova localização.

Discussão sobre o GNS • O GNS é descendente do Grapevine [Birrell *et al.* 1982] e do Clearinghouse [Oppen e Dalal 1983], dois sistemas de atribuição de nomes bem-sucedidos, desenvolvidos pela Xerox Corporation, principalmente para propósitos de distribuição de *e-mail*. O GNS trata com êxito das necessidades de capacidade de mudança de escala e reconfiguração, mas a solução adotada para integrar e mover árvores de diretório resultou em um requisito de um banco de dados (a tabela de diretórios bem conhecidos) que deve ser replicado em cada nó. Em uma rede de larga escala, as reconfigurações podem ocorrer em qualquer nível, e essa tabela poderia crescer bastante, entrando em conflito com o objetivo da capacidade de mudança de escala.

13.5 Estudo de caso: X.500 Directory Service

O X.500 é um serviço de diretório no sentido definido na Seção 13.3. Ele pode ser usado da mesma maneira que um serviço de nomes convencional, mas é utilizado principalmente para atender a consultas descritivas, projetadas para descobrir os nomes e atributos de outros usuários ou recursos de sistema. Os usuários podem usar uma variedade de requisitos de busca e navegação em um diretório de usuários, de organizações e recursos de sistema para obter informações sobre as entidades contidas nesse diretório. É provável que os usos de tal serviço sejam bastante diversificados. Eles variam desde solicitações diretamente análogas ao uso de catálogos telefônicos, como um acesso simples às “páginas brancas” para obter o endereço de correio eletrônico de um usuário ou uma consulta de “páginas amarelas” destinada, por exemplo, a obter os nomes e números de telefone de oficinas especializadas no reparo de uma marca de carro específica até o uso do diretório para acessar detalhes pessoais, como funções de trabalho, hábitos dietéticos ou mesmo fotografias das pessoas.

Tais consultas podem se originar a partir de usuários, como o caso das “páginas amarelas” exemplificado pela pergunta sobre oficinas mencionada anteriormente, ou de processos, quando elas podem ser usadas para identificar serviços para atender a um determinado requisito funcional.

Indivíduos e organizações podem usar um serviço de diretório para tornar disponível uma ampla variedade de informações sobre si mesmos e sobre os recursos que desejam oferecer para uso na rede. Os usuários podem pesquisar o diretório em busca de informações específicas com conhecimento apenas parcial de seu nome, estrutura ou conteúdo.

As organizações de padronização ITU e ISO definiram o *X.500 Directory Service* [ITU/ISO 1997] como um serviço de rede destinado a atender esses requisitos. O padrão se refere a ele como um serviço para acessar informações sobre “entidades do mundo real”, mas provavelmente também pode ser usado para acessar informações sobre serviços de *hardware* e *software* e dispositivos. O X.500 é especificado como um serviço em nível de aplicação no conjunto de padrões OSI (Open Systems Interconnection), mas seu projeto não depende significativamente dos outros padrões OSI e pode ser visto como um projeto de um serviço de diretório de propósito geral. Aqui, vamos descrever em linhas gerais o projeto do serviço de diretório X.500 e sua implementação. Os leitores que estiverem interessados em uma descrição mais detalhada do X.500 e dos métodos para sua implementação devem estudar o livro sobre o assunto [Rose 1992]. O X.500 também é a base do LDAP (discutido a seguir) e é usado no serviço de diretório DCE [OSF 1997].

Os dados armazenados nos servidores X.500 são organizados em uma estrutura em árvore com nós nomeados, como no caso dos outros servidores de nome discutidos neste capítulo, mas no X.500, uma grande variedade de atributos é armazenada em cada nó

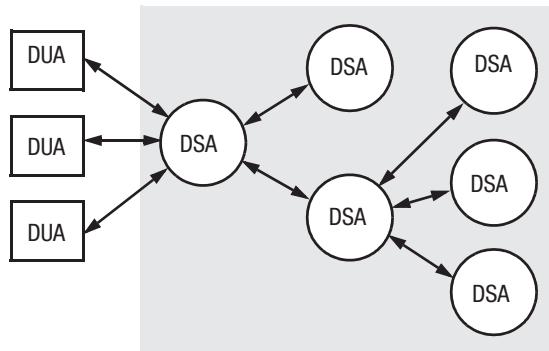


Figura 13.10 Arquitetura do serviço X.500.

da árvore, e o acesso não é apenas pelo nome, mas também pela busca de entradas com qualquer combinação de atributos exigida.

A árvore de nomes do X.500 é chamada de DIT (Directory Information Tree) e a estrutura de diretório inteira, incluindo os dados associados aos nós, é chamada de DIB (Directory Information Base). Ela se destina a ser uma única DIB integrada, contendo informações fornecidas por organizações de todo o mundo, com partes da DIB localizadas em servidores X.500 individuais. Normalmente, uma organização grande, ou de tamanho médio, forneceria pelo menos um servidor. Os clientes acessam o diretório estabelecendo uma conexão com um servidor e emitindo requisições de acesso. Os clientes podem entrar em contato com qualquer servidor para uma solicitação. Se os dados exigidos não estiverem no segmento da DIB mantida pelo servidor contatado, ele invocará outros servidores para solucionar a consulta ou redirecionará o cliente para outro servidor.

Na terminologia do padrão X.500, os servidores são DSAs (Directory Service Agents – agentes de serviço de diretório) e seus clientes se chamam DUAs (Directory User Agents – agentes de usuário de diretório). A Figura 13.10 mostra a arquitetura de software e um dos vários modelos de navegação possíveis, com cada processo cliente DUA interagindo com um único processo DSA, o qual acessa outros DSAs, conforme for necessário, para atender às requisições.

Cada entrada na DIB consiste em um nome e um conjunto de atributos. Assim como nos outros servidores de nome, o nome completo de uma entrada corresponde a um caminho pela DIT, da raiz da árvore até a entrada. Além dos nomes completos, ou *absolutos*, um DUA pode estabelecer um contexto, o qual inclui um nó de base, e depois usar nomes relativos mais curtos que forneçam o caminho do nó de base para a entrada nomeada.

A Figura 13.11 mostra a parte da Directory Information Tree que inclui a fictícia Universidade de Gormenghast, Grã-Bretanha, e a Figura 13.12 é uma das entradas de DIB associadas. A estrutura de dados das entradas na DIB e na DIT é muito flexível. Uma entrada de DIB consiste em um conjunto de atributos, em que um atributo tem um *tipo* e um ou mais *valores*. O tipo de cada atributo é denotado por um nome (por exemplo, *nomePaís*, *nomeOrganização*, *nomeComum*, *númeroTelefone*, *caixaCorreio*, *classeObjeto*). Novos tipos de atributo podem ser definidos, caso sejam exigidos. Para cada nome de tipo distinto existe uma definição de tipo correspondente, a qual inclui uma descrição do tipo e uma definição da sintaxe no ASN.1 Notation (uma notação padrão para definições de sintaxe), definindo representações para todos os valores permitidos do tipo.

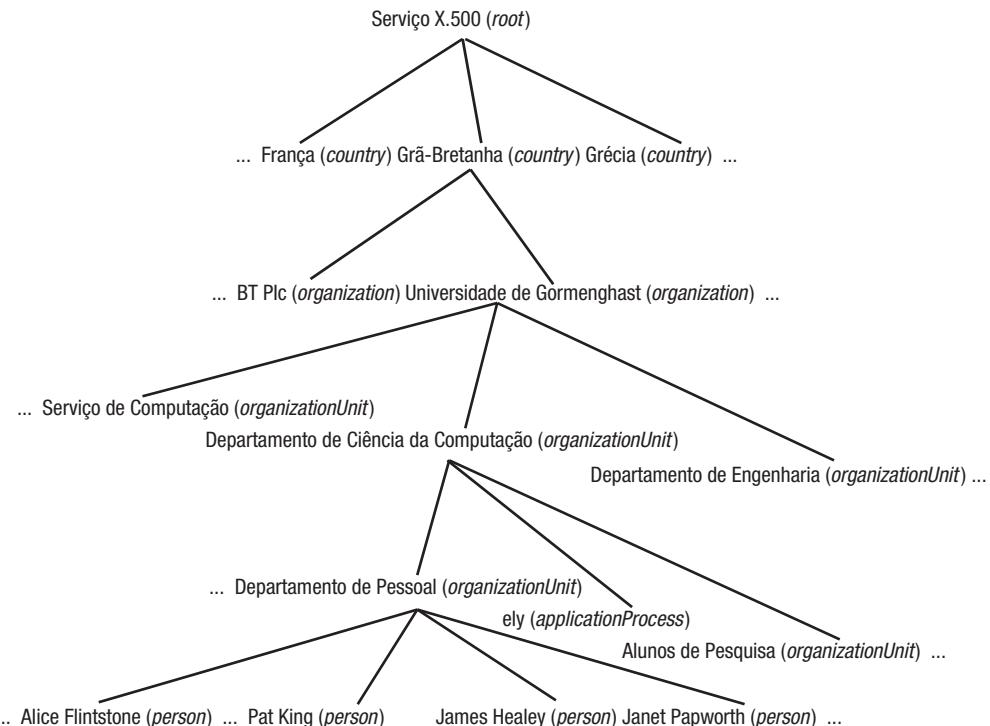


Figura 13.11 Parte da árvore de informações de diretório (DIT) do X.500.

As entradas de DIB são classificadas de maneira semelhante às estruturas de classe de objeto encontradas nas linguagens de programação orientadas a objetos. Cada entrada inclui um atributo *objectClass*, que determina a classe (ou classes) do objeto a que uma entrada se refere. *Organization*, *organizationalPerson* e *document* são exemplos de valores de *objectClass*. Mais classes podem ser definidas, conforme forem exigidas. A definição de uma classe determina quais atributos são obrigatórios e quais são opcionais para as entradas da classe dada. As definições das classes são organizadas em uma hierarquia de herança na qual todas as classes, exceto uma (chamada *topClass*), devem conter um atributo *objectClass*, e o valor do atributo *objectClass* deve ser o nome de uma ou mais classes. Se existirem diversos valores de *objectClass*, o objeto herdará os atributos obrigatórios e opcionais de cada uma das classes.

O nome de uma entrada de DIB (o nome que determina sua posição na DIT) é determinado pela escolha de um ou mais de seus atributos como *atributos distintos*. Os atributos escolhidos para esse propósito são referidos como DN (Distinguished Name) da entrada.

Agora, podemos considerar os métodos pelos quais o diretório é acessado. Existem dois tipos principais de requisições de acesso:

read: um nome absoluto ou relativo (um *nome de domínio* na terminologia do X.500) de uma entrada é fornecido, junto a uma lista dos atributos a serem lidos (ou uma indicação de que todos os atributos são exigidos). O DSA localiza a entrada nomeada navegando na DIT, fazendo requisições para outros servidores DSA quan-

<i>info</i>	
Alice Flintstone, Departamento de Pessoal, Departamento de Ciência da Computação, Universidade de Gormenghast, GB	
<i>commonName</i>	<i>uid</i>
Alice.L.Flintstone	alf
Alice.Flinton	
Alice Flintstone	alf@dcs.gormenghast.ac.uk
A. Flintstone	Alice.Flinton@dcs.gormenghast.ac.uk
<i>surname</i>	<i>roomNumber</i>
Flintstone	Z42
<i>telephoneNumber</i>	<i>userClass</i>
+44 986 33 4604	Aluno de Pesquisa

Figura 13.12 Uma entrada DIB do X.500.

do não contém as partes relevantes da árvore. Ele recupera os atributos solicitados e os retorna para o cliente.

search: esta é uma requisição de acesso baseada em atributo. Um nome de base e uma expressão de filtragem são fornecidos como argumentos. O nome de base especifica o nó na DIT a partir do qual a busca deve começar; a expressão de filtragem é uma expressão booleana que vai ser avaliada para cada nó abaixo do nó de base. O filtro especifica um critério de busca: uma combinação lógica de testes nos valores de todos os atributos de uma entrada. O comando *search* retorna uma lista de nomes (nomes de domínio) para todas as entradas abaixo do nó de base para as quais o filtro é avaliado como *VERDADEIRO*.

Por exemplo, poderia ser construído e aplicado um filtro para encontrar os *commonNames* de todos os membros do pessoal que ocupa a sala Z42 no Departamento de Ciência da Computação da Universidade de Gormenghast (Figura 13.12). Então, uma requisição *read* poderia ser usada para se obter qualquer um ou todos os atributos dessas entradas de DIB.

A busca pode ser muito dispendiosa quando aplicada a grandes partes da árvore de diretório (a qual pode residir em diversos servidores). Argumentos adicionais podem ser fornecidos para a operação *search*, para restringir a abrangência de sua busca, o tempo durante o qual uma busca pode continuar e o tamanho da lista de entradas retornada.

Administração e atualização da DIB • A interface DSA inclui operações para adicionar, excluir e modificar entradas. O controle de acesso é fornecido tanto para consultas como para operações de atualização; portanto, o acesso a partes da DIT pode ser restrito a certos usuários ou classes de usuário.

A DIB é particionada com a expectativa de que cada organização forneça pelo menos um servidor contendo os detalhes das entidades nela presentes. Partes da DIB podem ser replicadas em vários servidores.

Como padrão (ou “recomendação”, na terminologia do CCITT), o X.500 não trata de problemas de implementação. Entretanto, é bastante claro que qualquer implemen-

tação envolvendo vários servidores em uma rede de longa distância deve contar com o uso extensivo de técnicas de replicação e uso de cache, para evitar o redirecionamento demasiado das consultas.

Uma implementação, descrita em Rose [1992], é um sistema desenvolvido no University College, Londres, conhecido como QUIPU [Kille 1991]. Nessa implementação, o uso de cache e a replicação são realizados no nível das entradas de DIB individuais e no nível dos conjuntos de entradas descendentes do mesmo nó. Pressupõe-se que os valores podem se tornar inconsistentes após uma atualização, e o intervalo de tempo durante o qual a consistência é restaurada pode ser de vários minutos. Essa forma de disseminação da atualização geralmente é considerada aceitável para aplicações de serviço de diretório.

Lightweight Directory Access Protocol • A suposição do X.500 de que as organizações forneceriam suas próprias informações em diretórios públicos dentro de um sistema comum se mostrou amplamente infundada. Igualmente, sua complexidade resultou num entendimento relativamente reduzido a seu respeito. Um grupo da Universidade de Michigan propôs uma estratégia mais leve, chamada LDAP (Lightweight Directory Access Protocol), na qual um DUA acessa serviços de diretório X.500 diretamente sobre TCP/IP, em vez das camadas superiores da pilha de protocolos ISO. Isso está descrito na RFC 2251 [Wahl *et al.* 1997]. O LDAP simplifica a interface do X.500 de outras maneiras; por exemplo, ele fornece uma API relativamente simples e substitui a codificação ASN.1 por codificação textual.

Embora a especificação LDAP seja baseada no X.500, o LDAP não o exige. Uma implementação pode usar qualquer outro servidor de diretório que obedeça à especificação LDAP mais simples – em oposição à especificação X.500. Por exemplo, o Active Directory Service, da Microsoft, fornece uma interface LDAP. Ao contrário do X.500, o LDAP tem sido amplamente adotado, particularmente para serviços de diretório de intranet. Ele fornece acesso seguro aos dados do diretório, por meio de autenticação.

13.6 Resumo

Este capítulo descreveu o projeto e a implementação de serviços de nomes em sistemas distribuídos. Os serviços de nomes armazenam atributos de objetos em um sistema distribuído – em particular, seus endereços – e retornam esses atributos quando é fornecido um nome textual para ser pesquisado.

Os principais requisitos do serviço de nomes são a capacidade de manipular um número arbitrário de nomes, um tempo de vida longo, alta disponibilidade, o isolamento de falhas e a tolerância à desconfiança.

Os principais problemas de projeto são, primeiro, a estrutura do espaço de nomes – as regras sintáticas que governam os nomes. Um problema relacionado é o modelo de resolução: as regras pelas quais um nome com múltiplos componentes é transformado em um conjunto de atributos. O conjunto de nomes vinculados deve ser gerenciado. A maioria dos projetos considera que o espaço de nomes deve ser dividido em domínios – seções distintas do espaço de nomes, cada uma das quais associada a uma única autoridade de controle dos vínculos dos nomes que estão sob sua responsabilidade direta.

A implementação do serviço de nomes pode abranger diferentes organizações e comunidades de usuário. Em outras palavras, o conjunto de vínculos entre nomes e atributos é armazenado em vários servidores de nomes, cada um dos quais armazenando

pelo menos parte do conjunto de nomes dentro de um domínio de atribuição de nomes. Portanto, surge a questão da navegação – procedimento pelo qual um nome é resolvido quando as informações necessárias estão armazenadas em vários *sites*. Os tipos de navegação suportados são: iterativa, *multicast*, recursiva controlada pelo servidor e não recursiva controlada pelo servidor.

Outro aspecto importante da implementação de um serviço de nomes é o uso de replicação e cache. Ambos ajudam a tornar o serviço altamente disponível e reduzem o tempo que leva para resolver um nome.

Este capítulo considerou dois casos principais de projetos e implementações de serviço de nomes. O Domain Name System é amplamente usado para atribuição de nomes a computadores e para endereçamento de correio eletrônico na Internet; ele obtém bons tempos de resposta por meio de replicação e uso de cache. O Global Name Service é um projeto que ataca o problema da reconfiguração do espaço de nomes quando ocorrem alterações organizacionais.

Este capítulo também considerou os serviços de diretório, que fornecem dados sobre objetos e serviços correspondentes, quando os clientes fornecem descrições baseadas em atributos. O X.500 é um modelo para serviços de diretório que podem variar na abrangência desde organizações individuais até diretórios globais. Ele tem sido tipicamente usado em intranets desde a chegada do software LDAP.

Exercícios

- 13.1 Descreva os nomes (incluindo os identificadores) e os atributos usados em um serviço de arquivos distribuído, como o NFS (veja o Capítulo 12). *página 566*
- 13.2 Discuta os problemas levantados pelo uso de *alias* em um serviço de nomes e indique como eles podem ser superados, se houver solução. *página 571*
- 13.3 Explique por que a navegação iterativa é necessária em um serviço de nomes no qual diferentes espaços de nomes são parcialmente integrados, como o esquema de atribuição de nomes de arquivos fornecido pelo NFS. *página 574*
- 13.4 Descreva o problema dos nomes desvinculados na navegação por *multicast*. O que envolve a instalação de um servidor para responder às pesquisas de nomes desvinculados? *página 575*
- 13.5 Como o uso da cache ajuda a disponibilidade de um serviço de nomes? *página 576*
- 13.6 Discuta a ausência de uma distinção sintática (como o uso de um ‘.’ final) entre os nomes absolutos e relativos no DNS. *página 571*
- 13.7 Investigue sua configuração local de domínios e servidores DNS. Você pode encontrar um programa instalado, como o *dig* ou o *nslookup*, o qual permite realizar consultas individuais a servidores de nomes. *página 578*
- 13.8 Por que os servidores de DNS raízes contêm entradas para nomes de dois níveis, como *yahoo.com* e *purdue.edu*, em vez de nomes de um nível, como *edu* e *com*? *página 579*
- 13.9 Quais outros endereços de servidores de nomes os servidores DNS contêm por padrão e por quê? *página 579*
- 13.10 Por que um cliente de DNS poderia escolher a navegação recursiva em vez da navegação iterativa? Qual é a relevância da opção da navegação recursiva para a concorrência dentro de um servidor de nomes? *página 581*

- 13.11 Quando um servidor DNS poderia dar várias respostas para uma única pesquisa de nome e por quê? *página 581*
- 13.12 O GNS não garante que todas as cópias das entradas no banco de dados de atribuição de nomes sejam atualizadas. Como os clientes do GNS sabem que provavelmente receberam uma entrada desatualizada? Sob quais circunstâncias isso poderia ser prejudicial? *página 585*
- 13.13 Discuta as vantagens e os inconvenientes em potencial no uso de um serviço de diretório X.500 no lugar do DNS e dos programas de distribuição de *e-mail* na Internet. Esboce o projeto de um sistema de distribuição de *e-mail* para um conjunto de redes interligadas no qual todos os usuários e servidores de correio eletrônico são registrados em um banco de dados X.500. *página 588*
- 13.14 Quais problemas de segurança provavelmente são relevantes para um serviço de diretório, como o X.500, operando dentro de uma organização como uma universidade? *página 588*

14

Tempo e Estados Globais

- 14.1 Introdução
- 14.2 Relógios, eventos e estados de processo
- 14.3 Sincronização de relógios físicos
- 14.4 Tempo lógico e relógios lógicos
- 14.5 Estados globais
- 14.6 Depuração distribuída
- 14.7 Resumo

Neste capítulo, apresentaremos alguns assuntos relacionados à questão de tempo em sistemas distribuídos. O tempo é um problema prático importante.

Por exemplo, precisamos que os computadores espalhados pelo mundo informem o horário das transações de comércio eletrônico de modo consistente. O tempo também é uma construção teórica importante para se entender como as execuções distribuídas se desenrolam. No entanto, medir tempo é problemático nos sistemas distribuídos. Normalmente, cada computador tem seu próprio relógio físico e, havendo vários computadores, os relógios diferem entre si e é muito difícil sincronizá-los perfeitamente. Examinaremos algoritmos para a sincronização aproximada de relógios físicos e, depois, explicaremos os relógios lógicos, incluindo os relógios vetoriais, que são uma ferramenta para ordenar eventos sem saber precisamente quando eles ocorreram.

A ausência de um tempo físico global torna difícil descobrir o estado de nossos programas distribuídos quando eles são executados. Frequentemente, precisamos saber qual é o estado do processo A, quando o processo B está em determinado ponto de sua execução, mas não podemos contar com os relógios físicos para saber o que é verdade ao mesmo tempo. A segunda metade do capítulo examinará algoritmos para determinar os estados globais de computações distribuídas, a despeito da falta de um tempo global.

14.1 Introdução

Este capítulo apresenta conceitos fundamentais e algoritmos relacionados para monitorar sistemas distribuídos à medida que sua execução se desenrola e para saber o momento exato em que eventos ocorrem.

O tempo é um problema importante e interessante nos sistemas distribuídos, por vários motivos. Primeiro, ele é uma quantidade que frequentemente desejamos medir precisamente. Para saber em que hora do dia um evento específico ocorreu em um computador em particular é necessário sincronizar seu relógio com uma fonte de tempo externa confiável e aceita por todos. Por exemplo, uma transação de comércio eletrônico envolve eventos no computador do negociante e no computador de um banco. É importante, para propósitos de auditoria, que esses eventos tenham uma informação de tempo precisa.

Segundo, foram desenvolvidos algoritmos que dependem da sincronização do relógio para vários problemas de distribuição [Liskov 1993]. Entre eles estão a manutenção da consistência dos dados distribuídos (o uso de carimbos de tempo para dispor transações em série será discutido na Seção 16.6); a verificação da autenticidade de uma requisição enviada para um servidor (uma versão do protocolo de autenticação Kerberos, discutida no Capítulo 11, depende de relógios aproximadamente sincronizados); e a eliminação do processamento de atualizações replicadas (veja, por exemplo, Ladin *et al.* [1992]).

Medir o tempo pode ser problemático, devido à existência de vários pontos de referência. Einstein demonstrou, em sua Teoria da Relatividade Especial, as intrigantes consequências resultantes da observação de que a velocidade da luz é constante para todos os observadores, independentemente de sua velocidade relativa. A partir dessa suposição, ele provou, dentre outras coisas, que dois eventos considerados simultâneos em um ponto de referência não são necessariamente simultâneos de acordo com os observadores em outros pontos de referência que estão se movendo em relação a ele. Por exemplo, um observador na Terra e um observador viajando no espaço em uma nave espacial discordariam a respeito do intervalo de tempo entre os eventos, quanto mais suas velocidades relativas aumentassem.

A ordem relativa de dois eventos pode ser até invertida para dois observadores diferentes, mas isso não poderia acontecer se um evento tivesse causado a ocorrência do outro. Nesse caso, o efeito físico acompanha a causa física para todos os observadores, embora o tempo decorrido entre causa e efeito possa variar. Assim, a temporização de eventos físicos mostrou-se relativa ao observador, e a noção de Newton de tempo físico absoluto mostrou-se sem fundamento. Não existe no universo nenhum relógio físico especial ao qual possamos recorrer quando queremos medir intervalos de tempo.

A noção de tempo físico também é problemática em um sistema distribuído. Isso não se deve aos efeitos da teoria da relatividade, que são desprezíveis ou inexistentes para computadores normais (a não ser que se considere computadores viajando em naves espaciais!). O problema é baseado em uma limitação semelhante em nossa capacidade de registrar o tempo de eventos, em diferentes nós, de modo suficientemente preciso, para saber a ordem em que quaisquer dois eventos ocorreram, ou se eles ocorreram simultaneamente. Não existe um tempo global absoluto ao qual possamos recorrer. Apesar disso, às vezes precisamos observar os sistemas distribuídos e determinar se certos estados ocorreram ao mesmo tempo. Por exemplo, nos sistemas orientados a objetos, precisamos estabelecer se as referências para um objeto em particular não existem mais – se o objeto se tornou lixo (no caso em que podemos liberar sua memória). Estabelecer isso exige observações dos estados de processos (para des-

cobrir se eles contêm referências) e dos canais de comunicação entre processos (no caso em que mensagens contendo referências estejam em trânsito).

Na primeira metade deste capítulo, examinaremos métodos pelos quais os relógios de computador podem ser aproximadamente sincronizados, usando passagem de mensagens. Apresentaremos os relógios lógicos, incluindo os relógios vetoriais, os quais são usados para definir uma ordem de eventos, sem medir o tempo físico em que eles ocorreram.

Na segunda metade, descreveremos algoritmos cujo objetivo é capturar estados globais de sistemas distribuídos, à medida que eles executam.

14.2 Relógios, eventos e estados de processo

O Capítulo 2 apresentou um modelo introdutório de interação entre os processos dentro de um sistema distribuído. Vamos refinar esse modelo para nos ajudar a entender como caracterizamos a evolução do sistema quando ele executa e como indicamos o tempo dos eventos na execução de um sistema que interesse aos usuários. Começaremos considerando como ordenar e registrar o tempo dos eventos que ocorrem em um único processo.

Consideremos que um sistema distribuído consiste em um conjunto \varnothing de N processos p_i , $i = 1, 2, \dots, N$. Cada processo é executado em um único processador e os processadores não compartilham memória (o Capítulo 6 considerou brevemente o caso de processos que compartilham memória). Cada processo p_i em \varnothing tem um estado s_i que, em geral, ele transforma ao ser executado. O estado do processo inclui os valores de todas as variáveis que estão dentro dele. Seu estado também pode incluir os valores de todos os objetos do seu ambiente de sistema operacional local que ele afeta como, entre outros, os arquivos. Presumimos que os processos não podem se comunicar de nenhuma maneira, exceto por meio do envio de mensagens pela rede. Assim, por exemplo, se existem processos que operam os braços de um robô, conectados em seus respectivos nós no sistema, então eles não podem se comunicar com um aperto de mãos!

Quando cada processo p_i executa, ele efetua uma série de ações, cada uma das quais sendo uma operação de *envio* ou *recepção* de mensagens, ou uma operação que transforma o estado de p_i – que altera um ou mais valores presentes em s_i . Na prática, podemos optar por usar uma descrição de alto nível das ações, de acordo com a aplicação. Por exemplo, se os processos em \varnothing estão envolvidos em uma aplicação de comércio eletrônico, então as ações podem ser “o cliente enviou uma mensagem de requisição” ou “o servidor negociante gravou a transação no *log*”.

Definimos um evento como a ocorrência de uma única ação que um processo realiza ao ser executado – uma ação de comunicação ou uma ação de transformação de estado. A sequência de eventos dentro de um único processo p_i pode ser colocada em uma ordem total única, que denotaremos pela relação \rightarrow_i entre os eventos. Isto é, $e \rightarrow_i e'$ se e somente se o evento e ocorre antes de e' em p_i . Essa ordem é bem definida, seja o processo *multi-threaded* ou não, pois supomos que o processo é executado em um único processador.

Agora, podemos definir o histórico do processo p_i como sendo a série de eventos que ocorrem dentro dele, ordenados conforme descrevemos pela relação \rightarrow_i :

$$\text{histórico}(p_i) = h_i = \langle e^0_{p_i}, e^1_{p_i}, e^2_{p_i}, \dots \rangle$$

Relógios • Vimos como ordenar os eventos em um processo, mas não como fornecer seu carimbo de tempo – atribuir a eles uma data e uma hora do dia. Cada computador contém seu próprio relógio físico. Esses relógios são dispositivos eletrônicos que contam

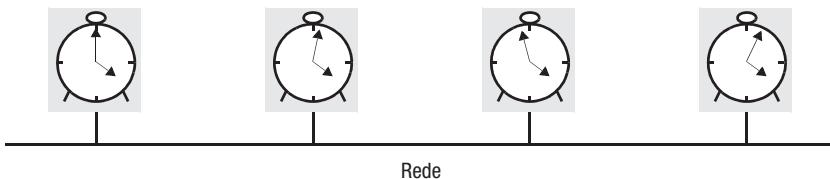


Figura 14.1 Distorção entre relógios de computador em um sistema distribuído.

as oscilações que ocorrem em um cristal com uma frequência de oscilação bem definida e que normalmente dividem essa contagem e armazenam o resultado em um registrador contador. Os dispositivos de relógio podem ser programados para gerar interrupções em intervalos regulares para que, por exemplo, possa ser implementada a fração de tempo de execução. Cada uma dessas interrupções é denominada de tiques do relógio (*clock ticks*). Contudo, não vamos nos preocupar com esse aspecto do funcionamento do relógio.

O sistema operacional lê o valor do relógio de *hardware*, $H_i(t)$, acerta sua escala e adiciona uma compensação para produzir um relógio de *software* $C_i(t) = \alpha H_i(t) + \beta$ que mede, aproximadamente, o tempo físico real t do processo p_i . Em outras palavras, quando o tempo real em um ponto de referência absoluto é t , $C_i(t)$ é a leitura no relógio de *software*. Por exemplo, $C_i(t)$ poderia ser o valor de 64 bits do número de nanosegundos decorridos no tempo t desde um tempo de referência conveniente. Em geral, o relógio não é completamente preciso e, assim, $C_i(t)$ irá diferir de t . Contudo, se C_i se comportar suficientemente bem (vamos examinar a noção de correção do relógio em breve), podemos usar seu valor para registrar o tempo de qualquer evento de p_i . Note que eventos sucessivos corresponderão a diferentes carimbos de tempo somente se a *resolução do relógio* – o período entre as atualizações do valor do relógio – for menor do que o intervalo de tempo entre os eventos sucessivos. A taxa com que os eventos ocorrem depende de fatores como o comprimento do ciclo de instrução do processador.

Desvio de relógio e derivação de relógio • Os relógios de computador, assim como os outros relógios, tendem a não estar em perfeito acordo (Figura 14.1). A diferença instantânea entre as leituras de quaisquer dois relógios é chamada de *desvio* (*skew*). Além disso, os relógios baseados em cristal usados nos computadores estão, assim como todos os outros relógios, sujeitos a *deriva de relógio* (*drift*), que significa que eles contam o tempo com diferentes velocidades e, portanto, divergem. Os osciladores de cristal estão sujeitos a variações físicas, e a consequência é que suas frequências de oscilação diferem. Além disso, a frequência de um mesmo relógio varia até com a temperatura. Existem projetos físicos que tentam compensar essa variação, mas eles não conseguem eliminá-la. A diferença no período de oscilação entre dois relógios pode ser extremamente pequena, mas a diferença acumulada em muitas oscilações leva a uma diferença perceptível nos contadores registrados pelos dois relógios, independentemente da precisão com que foram inicializados no mesmo valor. A *taxa de deriva* (*drift rate*) de um relógio é a defasagem (diferença na leitura) entre o relógio local e um relógio de referência nominal perfeito, por unidade de tempo, medida pelo relógio de referência. Para relógios normais, baseados em um cristal de quartzo, isso dá cerca de 10^{-6} segundos de diferença em cada segundo – uma diferença de 1 segundo a cada 1.000.000 segundos, ou 11,6 dias. A taxa de derivação dos relógios de quartzo de alta precisão é de cerca de 10^{-7} ou 10^{-8} .

Tempo Universal Coordenado • Os relógios de computador podem ser sincronizados com fontes externas de tempo altamente precisas. Os relógios físicos mais precisos usam osciladores atômicos, cuja taxa de deriva é de cerca de uma parte em 10^{13} . A saída desses relógios atômicos é usada como padrão para o tempo real decorrido, conhecido como *International Atomic Time* (tempo atômico internacional). Desde 1967, um segundo padrão foi definido como 9.192.631.770 períodos de transição entre duas camadas hiperfinas do estado fundamental do Césio-133 (Cs^{133}).

Os segundos, anos e outras unidades de tempo que usamos têm suas raízes no tempo astronômico. Eles foram originalmente definidos em termos da rotação da Terra sobre seu eixo e sua translação em torno do Sol. Entretanto, o período de rotação da Terra sobre seu eixo está ficando gradualmente maior, principalmente devido ao atrito das marés. Os efeitos atmosféricos e as correntes de convecção dentro do núcleo da Terra também causam aumentos e diminuições no período de rotação da Terra. Portanto, o tempo astronômico e o tempo atômico têm a tendência de entrar em descompasso.

O *Tempo Universal Coordenado* – abreviado como UTC (do seu equivalente em francês) – é um padrão internacional para contagem de tempo. Ele é baseado no tempo atômico, mas ocasionalmente é inserido – ou, mais raramente, excluído – o conhecido segundo bissexto, para manter a sincronização com o tempo astronômico. Os sinais UTC são sincronizados e transmitidos regularmente de estações de rádio terrestres e satélites cobrindo muitas partes do planeta. Por exemplo, nos Estados Unidos, a estação de rádio WWV transmite sinais de temporização em várias frequências de ondas curtas. As fontes de satélite incluem o GPS (*Global Positioning System*).

Receptores estão disponíveis comercialmente. Comparados com o UTC perfeito, os sinais recebidos das estações terrestres têm uma precisão da ordem de 0,1 a 10 milissegundos, dependendo da estação usada. Os sinais recebidos do GPS têm precisão de cerca de 1 microsegundo. Computadores com receptores agregados podem sincronizar seus relógios com esses sinais de temporização.

14.3 Sincronização de relógios físicos

Para saber em que hora do dia os eventos ocorrem nos processos de nosso sistema distribuído \wp – por exemplo, para propósitos de contabilidade –, é necessário sincronizar os relógios C_i dos processos com uma fonte de tempo externa de referência. Esta é a *sincronização externa*. E se os relógios C_i são sincronizados com um grau de precisão conhecido, então podemos medir o intervalo entre dois eventos que ocorrem em diferentes computadores, recorrendo aos seus relógios locais – mesmo que eles não estejam necessariamente sincronizados com uma fonte de tempo externa. Esta é a *sincronização interna*. Definimos mais rigorosamente esses dois modos de sincronização sobre um intervalo de tempo real I como segue:

Sincronização externa: para um limite de sincronização $D > 0$ e para uma fonte S de tempo UTC, a relação $|S(t) - C_i(t)| < D$, é verdadeira para $i = 1, 2, \dots, N$ e para todos os tempos reais t em I . Outra maneira de expressar isso é que os relógios C_i são *precisos* dentro do limite D .

Sincronização interna: para um limite de sincronismo $D > 0$, a relação $|C_i(t) - C_j(t)| < D$ é verdadeira para todo $i, j = 1, 2, \dots, N$ e para todos os tempos reais t em I . Outra maneira de expressar isso é que os relógios C_i concordam dentro do limite D .

Os relógios sincronizados internamente não são necessariamente sincronizados externamente, pois podem se desviar coletivamente de uma fonte de tempo externa, mesmo que concordem uns com os outros. Entretanto, a partir das definições, conclui-se que, se o sistema \wp for sincronizado externamente com um limite D , então o mesmo sistema será sincronizado internamente com um limite de $2D$.

Várias noções de *correção* para relógios foram sugeridas. É comum definir um relógio de *hardware* H como correto, se sua taxa de deriva está dentro de um limite conhecido $\rho > 0$ (um valor derivado de outro fornecido pelo fabricante, como 10^{-6} segundos/segundo). Isso significa que o erro na medida do intervalo entre os tempos reais t e t' ($t' > t$) é limitado:

$$(1 - \rho)(t' - t) \leq H(t') - H(t) \leq (1 + \rho)(t' - t)$$

Essa condição impossibilita saltos no valor de relógios de *hardware* (durante a operação normal). Às vezes, também exigimos que nossos relógios de *software* obedeçam à condição, mas uma condição menos exigente de *monotonicidade* pode bastar. Monotonicidade é a condição de que um relógio C apenas sempre avance:

$$t' > t \Rightarrow C(t') > C(t)$$

Por exemplo, o recurso *make* do UNIX é uma ferramenta usada para compilar apenas os arquivos fontes que foram modificados desde a última vez em que foram compilados. As datas de modificação de cada par correspondente de arquivos fonte e objeto são comparadas para determinar essa condição. Se, em um computador cujo relógio estivesse adiantado, fosse feita a compilação de um arquivo fonte e, em seguida, o relógio fosse atrasado para acertá-lo e, logo na sequência, o arquivo fosse modificado, poderia parecer que o arquivo fonte foi modificado antes da compilação. Erroneamente, o recurso *make* não compilaria o arquivo fonte.

Podemos obter a monotonicidade apesar do fato de um relógio se encontrar adiantando. Só precisamos mudar a taxa em que as atualizações são feitas para o tempo repassado às aplicações. Isso pode ser conseguido no *software*, sem mudar a taxa dos tiques do relógio de *hardware* físico – lembre-se de que $C_i(t) = \alpha H_i(t) + \beta$, onde estamos livres para escolher os valores de α e β .

Uma condição de correção mista, que às vezes é aplicada, é exigir que um relógio obedeça à condição de monotonicidade e que sua taxa de deriva seja limitada entre os pontos de sincronização, mas permitir que o valor do relógio salte para frente nesses pontos de sincronização.

Um relógio que não aplica as condições de correção é definido como *falso*. Diz-se que ocorre uma *falla de colapso* do relógio quando ele para de tiquetaquear completamente; qualquer outra falha de relógio é uma *falla arbitrária*. Um exemplo de falha arbitrária é o de um relógio com o *bug* do ano 2000 (*bug Y2K*), que viola a condição de monotonicidade registrando a data, após 31 de dezembro de 1999, como 1 de janeiro de 1900, em vez de 2000; outro exemplo é um relógio com bateria fraca e cuja taxa de derivação repentinamente torna-se muito grande.

Note que, de acordo com as definições, os relógios não precisam ser precisos para estarem corretos. Como o objetivo pode ser a sincronização interna, em vez da externa, os critérios de correção se preocupam apenas com o funcionamento correto do mecanismo do relógio e não com sua configuração absoluta.

Vamos descrever agora os algoritmos de sincronização externa e interna.

14.3.1 Sincronização em um sistema síncrono

Começaremos considerando o caso mais simples possível: o da sincronização interna entre dois processos em um sistema síncrono distribuído. Em um sistema síncrono, são conhecidos os limites da taxa de deriva dos relógios, o atraso máximo de transmissão de mensagens e o tempo que leva para executar cada etapa de um processo (veja a Seção 2.4.1).

Um processo envia o tempo t de seu relógio local para o outro processo em uma mensagem m . Em princípio, o processo receptor poderia configurar seu relógio com o tempo $t + T_{trans}$, onde T_{trans} é o tempo que leva para transmissão de m entre eles. Então, os dois relógios concordariam (pois o objetivo é a sincronização interna; não importa se o relógio do processo remetente é preciso).

Infelizmente, T_{trans} está sujeito à variação e é desconhecido. Em geral, outros processos estão competindo pelos recursos com os processos a serem sincronizados, em seus respectivos nós, e outras mensagens competem com m pela rede. Contudo, sempre há um tempo de transmissão mínimo min que seria obtido se nenhum outro processo fosse executado e se não existisse nenhum outro tráfego na rede; min pode ser medido, ou estimado, de forma conservadora.

Por definição, em um sistema síncrono também existe um limite superior max para o tempo que leva para transmitir uma mensagem. Seja u a incerteza no tempo de transmissão da mensagem, tal que $u = (max - min)$. Se o receptor configurar seu relógio como $t + min$, então o desvio do relógio poderá ser no máximo u , pois, na verdade, a mensagem pode ter demorado o tempo max para chegar. Analogamente, se ele configurar seu relógio como $t + max$, o desvio poderá novamente ter o tamanho de u . Entretanto, se ele configurar seu relógio no ponto médio, $t + (max + min)/2$, então o desvio será no máximo $u/2$. Em geral, para um sistema síncrono, o limite ótimo que pode ser obtido para o desvio do relógio ao se sincronizar N relógios é $u(1 - 1/N)$ [Lundelius e Lynch 1984].

A maioria dos sistemas distribuídos encontrados na prática é assíncrona: os fatores que mais influenciam os atrasos de mensagem não estão ligados ao seu efeito, e não existe nenhum limite superior max para os atrasos na transmissão de mensagens. Isso é particularmente verdadeiro no caso da Internet. Para um sistema assíncrono, podemos dizer somente que $T_{trans} = min + x$, onde $x \geq 0$. O valor de x não é conhecido em um caso em particular, embora uma distribuição de valores possa ser mensurável para uma determinada instalação.

14.3.2 Método de Cristian para sincronização de relógios

Cristian [1989] sugeriu o uso de um servidor de tempo, conectado a um dispositivo que recebe sinais de uma fonte UTC, para sincronizar computadores externamente. Ao receber uma requisição, o processo servidor S fornece o tempo, de acordo com seu relógio,

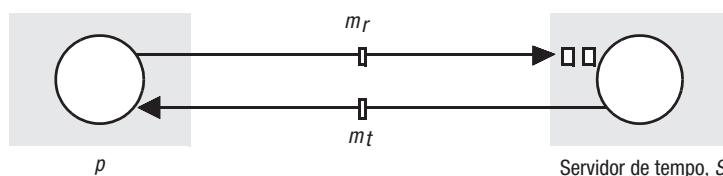


Figura 14.2 Sincronização de relógio usando um servidor de tempo.

como mostrado Figura 14.2. Cristian observou que, embora não haja um limite superior para os atrasos na transmissão de mensagens em um sistema assíncrono, frequentemente os tempos de viagem de ida e volta para mensagens trocadas entre pares de processos são razoavelmente curtos – uma pequena fração de segundo. Ele descreve o algoritmo como *probabilístico*: o método só obtém sincronização se os tempos de viagem de ida e volta observados entre cliente e servidor forem suficientemente curtos, comparados com a precisão exigida.

Um processo p solicita o tempo em uma mensagem m_r e recebe o valor de tempo t em uma mensagem m_t (t é inserido em m_t no último ponto possível antes da transmissão do computador de S). O processo p registra o tempo de viagem de ida e volta total T_{viagem} que leva para enviar a requisição m_r e receber a resposta m_t . Ele pode medir esse tempo com precisão razoável, se sua taxa de deriva de relógio for pequena. Por exemplo, o tempo de viagem de ida e volta deve ser da ordem de 1 a 10 milissegundos em uma rede local, no qual um relógio com uma taxa de deriva de 10^{-6} segundos/segundo varia no máximo por 10^{-5} milissegundos.

Uma estimativa simples do tempo para o qual p deve configurar seu relógio é $t + T_{viagem}/2$, que presume que o tempo decorrido é dividido igualmente, antes e depois de S ter colocado t em m_t . Normalmente, essa é uma suposição razoavelmente precisa, a menos que as duas mensagens sejam transmitidas por redes diferentes. Se o valor do tempo de transmissão mínimo min for conhecido, ou puder ser estimado conservadoramente, então podemos determinar a precisão desse resultado, como segue.

O mais cedo que S poderia ter colocado o tempo em m_t seria min após p ter enviado m_r . O mais tarde que ele poderia ter feito isso seria min antes que m_t chegasse a p . Portanto, o tempo, de acordo com o relógio de S , para a chegada da mensagem de resposta está no intervalo $[t + min, t + T_{viagem} - min]$. A largura desse intervalo é $T_{viagem} - 2min$, de modo que a precisão é $\pm(T_{viagem}/2 - min)$.

A variabilidade pode ser tratada até certo ponto, fazendo-se várias requisições para S (espaçando-os para que congestionamentos transitórios na rede possam terminar) e pegando-se o valor mínimo de T_{viagem} para fazer a estimativa mais precisa. Quanto maior a precisão exigida, menor a probabilidade de obtê-la. Isso porque os resultados mais precisos são aqueles nos quais as duas mensagens são transmitidas em um tempo próximo a min – um evento improvável em uma rede ocupada.

Discussão sobre o algoritmo de Cristian • Conforme foi descrito, o método de Cristian sofre do problema associado a todos os serviços implementados por um único servidor: o de que o único servidor de tempo pode falhar e, assim, tornar a sincronização temporariamente impossível. Por isso, Cristian sugeriu que o tempo fosse fornecido por um conjunto de servidores de tempo sincronizados, cada um com um receptor de sinais de tempo UTC. Por exemplo, um cliente poderia enviar sua requisição em *multicast* para todos os servidores e utilizar apenas a primeira resposta obtida.

Note que um servidor de tempo defeituoso que respondesse com valores de tempo espúrios, ou um servidor de tempo impostor que respondesse com tempos deliberadamente incorretos, poderia prejudicar um sistema de computação. Esses problemas estão fora dos objetivos do trabalho descrito por Cristian [1989], o qual presume que as fontes de sinais de tempo externas têm verificação automática. Cristian e Fetzer [1994] descrevem uma família de protocolos probabilísticos para sincronização interna do relógio, cada um dos quais tolerando certas falhas. Srikanth e Toueg [1987] descreveram pela primeira vez um algoritmo excelente com relação à precisão dos relógios sincronizados, embora tolere apenas algumas falhas. Dolev *et al.* [1986] mostraram que, se é o número

de relógios defeituosos de um total de N , então devemos ter $N > 3f$, se os outros relógios, corretos, ainda forem capazes de entrar em acordo. O problema do tratamento com relógios defeituosos é parcialmente resolvido pelo algoritmo Berkeley, que será descrito a seguir. O problema da interferência mal-intencionada na sincronização de relógios pode ser resolvido com técnicas de autenticação.

14.3.3 O algoritmo Berkeley

Gusella e Zatti [1989] descrevem um algoritmo de sincronização interna que desenvolveram para conjuntos de computadores executando o UNIX Berkeley. Nele, um computador é escolhido para atuar como *mestre*. Ao contrário do protocolo de Cristian, esse computador faz periodicamente uma consulta sequencial nos outros computadores cujos relógios devem ser sincronizados, chamados de *escravos*. Os escravos enviam de volta seus valores de relógio. O mestre faz uma estimativa dos tempos locais desses relógios, observando os tempos de viagem de ida e volta (semelhante à técnica de Cristian) e faz a média dos valores obtidos (incluindo a leitura de seu próprio relógio). O balanço das probabilidades é que essa média cancela as tendências dos relógios individuais de estarem adiantados ou atrasados. A precisão do protocolo depende de um tempo máximo nominal de viagem de ida e volta entre o mestre e os escravos. O mestre elimina todas as leituras ocasionais associadas aos tempos maiores do que esse máximo.

Em vez de enviar para os outros computadores o tempo corrente atualizado – o que introduziria mais incerteza, devido ao tempo de transmissão da mensagem –, o mestre envia o valor pelo qual o relógio de cada escravo individual exige ajuste. Esse valor pode ser positivo ou negativo.

O algoritmo elimina as leituras de relógios defeituosos. Tais relógios teriam um efeito adverso significativo, caso fosse tirada uma média normal. O mestre tira uma *média tolerante a falhas*, ou seja, escolhe-se um subconjunto dos relógios que não diferem uns dos outros por mais do que um valor especificado, e é feita a média das leituras apenas desses relógios.

Gusella e Zatti descrevem uma experiência envolvendo 15 computadores, cujos relógios foram sincronizados dentro de cerca de 20 a 25 milissegundos, usando seu protocolo. A taxa de deriva dos relógios locais foi medida como menos de 2×10^{-5} e o tempo de viagem de ida e volta máximo obtido foi de 10 milissegundos.

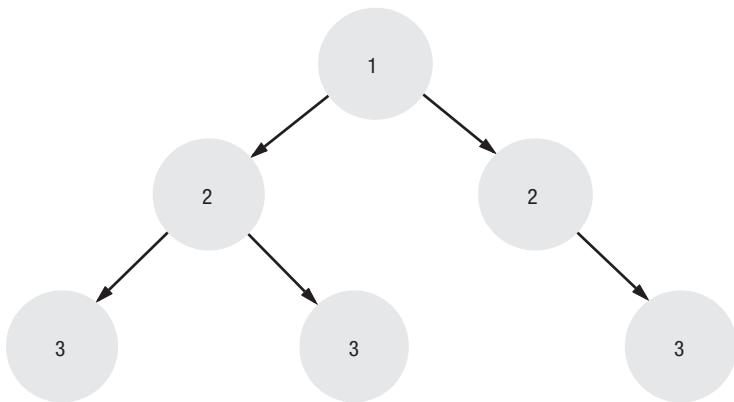
Se o mestre falhar, então outro poderá ser escolhido para assumir o controle e funcionar exatamente como seu predecessor. A Seção 15.3 discute alguns algoritmos de eleição de propósito geral. Note que, neles, não há garantias da escolha de um novo mestre dentro de um limite de tempo – e, portanto, a diferença entre dois relógios poderia ser ilimitada, caso eles fossem usados.

14.3.4 O Network Time Protocol

O método de Cristian e o algoritmo Berkeley se destinam principalmente ao uso dentro de intranets. O NTP (Network Time Protocol) [Mills 1995] define uma arquitetura para um serviço de tempo e um protocolo para distribuir informações de tempo pela Internet.

Os principais objetivos de projeto e as características do NTP são os seguintes:

Fornecer um serviço que permita aos clientes na Internet serem sincronizados precisamente com o UTC, a despeito dos atrasos de mensagem, grandes e variáveis, encontrados na comunicação via Internet. O NTP emprega técnicas estatísticas para a filtragem de dados de temporização e faz discriminação entre a qualidade dos dados de tempo de diferentes servidores.



Nota: as setas indicam controle de sincronização, os números fornecem o stratum.

Figura 14.3 Um exemplo de sub-rede de sincronização em uma implementação de NTP.

Fornecer um serviço confiável que possa sobreviver a longas perdas de conectividade. Existem servidores redundantes e caminhos redundantes entre os servidores. Os servidores podem ser reconfigurados para que continuem a fornecer o serviço, caso um deles se torne inatingível.

Permitir que os clientes sejam sincronizados de forma suficientemente frequente para compensar as taxas de deriva encontradas na maioria dos computadores. O serviço é projetado para suportar um grande número de clientes e servidores.

Fornecer proteção contra interferência no serviço de tempo, seja mal-intencionada ou acidental. O serviço de tempo usa técnicas de autenticação para verificar se os dados de temporização são originários das fontes confiáveis conhecidas. Ele também valida os endereços de retorno das mensagens que recebe.

O serviço NTP é fornecido por uma rede de servidores localizados na Internet. Os *servidores primários* são conectados diretamente a uma fonte de tempo, como um relógio de rádio recebendo UTC; os *servidores secundários* são sincronizados com os servidores principais. Os servidores são conectados em uma hierarquia lógica chamada *sub-rede de sincronização* (veja a Figura 14.3), cujos níveis são chamados de *strata*. Os servidores primários ocupam o *stratum 1*: eles estão na raiz. Os servidores do *stratum 2* são secundários, sincronizados diretamente com os servidores primários; os servidores do *stratum 3* são sincronizados com os servidores do *stratum 2* e assim por diante. Os servidores de nível mais baixo (folha) são executados nas estações de trabalho dos usuários.

Os relógios pertencentes aos servidores com números de *stratum* mais altos estão sujeitos a serem menos precisos do que aqueles com números de *stratum* baixos, pois erros são introduzidos em cada nível de sincronização. O NTP também leva em conta, na avaliação da qualidade dos dados de temporização mantidos por um servidor em particular, os atrasos da viagem de ida e volta total da mensagem até a raiz.

A sub-rede de sincronização pode ser reconfigurada quando servidores se tornam inatingíveis ou quando ocorrem falhas. Se, por exemplo, a fonte de UTC de um servidor primário falha, ele pode se tornar um servidor secundário do *stratum 2*. Se a fonte de

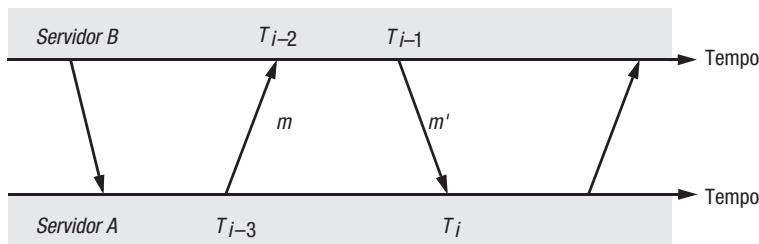


Figura 14.4 Mensagens trocadas entre dois pares NTP.

sincronismo normal de um servidor secundário falha, ou se torna inatingível, ele pode ser sincronizado com outro servidor.

Os servidores NTP são sincronizados entre si de três maneiras: *multicast*, chamada de procedimento e modo simétrico. O *modo multicast* se destina a ser utilizado em uma rede local de alta velocidade. Periodicamente, um ou mais servidores de tempo enviam em *multicast* a informação de tempo para servidores que estão em execução em outros computadores conectados na rede local, os quais configuraram seus relógios pressupondo um pequeno atraso. Esse modo pode obter apenas uma precisão relativamente baixa, mas considerada suficiente para muitos propósitos.

O *modo de chamada de procedimento* é semelhante ao funcionamento do algoritmo de Cristian, descrito anteriormente. Nesse modo, um servidor aceita requisições de outros computadores, os quais ele processa respondendo com seu carimbo de tempo (leitura corrente do relógio). Esse modo é conveniente quando é exigida uma precisão melhor do que a que pode ser obtida com o *multicast* – ou quando *multicast* não é suportado por *hardware*. Por exemplo, servidores de arquivos na mesma rede local, ou em uma rede local vizinha, que precisem manter informações precisas do tempo de acesso a arquivos, poderiam entrar em contato com um servidor local no modo de chamada de procedimento.

Finalmente, o *modo simétrico* se destina a ser utilizado pelos servidores que fornecem informações de tempo em redes locais e pelos níveis hierarquicamente mais altos (número de *stratum* mais baixos) da sub-rede de sincronização, em que uma maior precisão necessita ser obtida. Dois servidores operando no modo simétrico trocam mensagens com informações de temporização. Esses dados são armazenados como parte de uma associação entre os servidores que são mantidos para melhorar a precisão de sua sincronização com o passar do tempo.

Em todos os modos, as mensagens são enviadas de maneira não confiável, usando o protocolo de transporte UDP. No modo de chamada de procedimento e no modo simétrico, os processos trocam pares de mensagens. Cada mensagem carrega carimbos de tempo de eventos de mensagem recentes: os tempos locais de quando a mensagem NTP anterior entre o par foi enviada e recebida e o tempo local de quando a mensagem corrente foi transmitida. O destinatário da mensagem NTP anota o tempo local ao receber a mensagem. Os quatro tempos, T_{i-3} , T_{i-2} , T_{i-1} , e T_i aparecem na Figura 14.4 para as mensagens m e m' enviadas entre os servidores A e B. Note que, no modo simétrico, ao contrário do algoritmo de Cristian descrito anteriormente, pode haver um atraso não desprezível entre a chegada de uma mensagem e o envio da próxima. Além disso, mensagens podem ser perdidas, mas os três carimbos de tempo transportados em cada mensagem são válidos.

Para cada par de mensagens enviadas entre dois servidores, o NTP calcula uma *compensação* c_i , que é uma estimativa da compensação real entre os dois relógios, e um *atraso* a_i , que é o tempo de transmissão total das duas mensagens. Se o valor real da compensação do relógio em B , relativa à de A , for c e, se os tempos de transmissão reais de m e m' forem t e t' respectivamente, então temos:

$$T_{i-2} = T_{i-3} + t + c \text{ e } T_i = T_{i-1} + t' - c$$

Isso leva a:

$$a_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

e:

$$c = c_i + (t' - t)/2, \text{ onde } c_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$$

Usando o fato de que $t, t' \geq 0$, pode-se mostrar que $c_i - a_i/2 \leq c \leq c_i + a_i/2$. Assim, c_i é uma estimativa da compensação e a_i é uma medida da precisão dessa estimativa.

Os servidores NTP aplicam um algoritmo de filtragem de dados em sucessivos pares $\langle c_i, a_i \rangle$, para fazer uma estimativa da compensação c e calcular a qualidade dessa estimativa com base em uma quantidade estatística chamada *filtro de dispersão*. Uma dispersão relativamente alta representa dados relativamente não confiáveis. Os oito pares $\langle c_i, a_i \rangle$ mais recentes são mantidos. Assim como acontece com o algoritmo de Cristian, o valor de c_j que corresponde ao valor mínimo a_j é escolhido para fazer a estimativa de c .

Entretanto, o valor da compensação obtido da comunicação com uma única fonte não é necessariamente usado sozinho para controlar o relógio local. Em geral, um servidor NTP se envolve nas trocas de mensagem com vários de seus pares. Além da filtragem de dados aplicada nas trocas com cada par, o NTP aplica um algoritmo de seleção de par. Esse algoritmo examina os valores obtidos das trocas com cada um dos vários pares, procurando valores relativamente não confiáveis. A saída desse algoritmo pode fazer um servidor mudar o par que normalmente utiliza para obter sincronismo.

Os pares com números de *stratum* mais baixos são mais favorecidos do que os que possuem números de *stratum* mais altos, pois estão mais próximos das fontes de tempo principais. Além disso, aqueles com a *dispersão de sincronização* mais baixa são relativamente favorecidos. Essa é a soma das dispersões de filtro medidas entre o servidor e a raiz da sub-rede de sincronização. (Os pares trocam dispersões de sincronização nas mensagens, permitindo que esse total seja calculado.)

O NTP emprega um modelo de laço com bloqueio de fase [Mills 1995], o qual modifica a frequência de atualização do relógio local de acordo com observações de sua taxa de deriva. Para dar um exemplo simples, se é descoberto um relógio que sempre avança no tempo na taxa de, digamos, quatro segundos por hora, então sua frequência pode ser reduzida ligeiramente (no *software* ou no *hardware*) para compensar isso. Assim, a deriva do relógio entre intervalos de sincronização é reduzida.

Mills cita precisão de sincronização na ordem de dezenas de milissegundos na Internet e de um milissegundo em redes locais.

14.4 Tempo lógico e relógios lógicos

Do ponto de vista de um único processo, os eventos são ordenados exclusivamente pelos tempos dados pelo relógio local. Entretanto, conforme Lamport [1978] mostrou, como

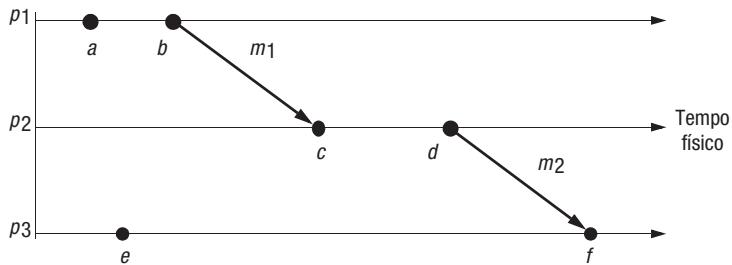


Figura 14.5 Eventos ocorrendo em três processos.

não podemos sincronizar perfeitamente os relógios em um sistema distribuído, em geral não podemos usar o tempo físico para descobrir a ordem de qualquer par de eventos arbitrários que ocorram dentro dele.

Em geral, podemos usar um esquema semelhante à causalidade física, mas aplicada aos sistemas distribuídos, para ordenar alguns dos eventos que ocorrem em diferentes processos. Essa ordenação é baseada em dois pontos simples e intuitivamente óbvios:

- Se dois eventos ocorreram no mesmo processo p_i ($i = 1, 2, \dots, N$), então eles ocorreram na ordem em que p_i os observou – essa é a ordem \rightarrow_i que definimos anteriormente.
- Quando uma mensagem é enviada entre processos, o evento de envio da mensagem ocorreu antes do evento de receção da mensagem.

Lamport chamou a ordenação parcial obtida pela generalização desses dois relacionamentos de relação acontece antes (*happened-before*). Às vezes, ela também é conhecida como relação de *ordenação causal* (*causal ordering*) ou *ordenação causal potencial* (*potential causal ordering*).

Podemos definir a relação acontece antes (AA), denotada por \rightarrow , como segue:

AA1: Se \exists processo $p_i: e \rightarrow_i e'$, então $e \rightarrow e'$.

AA2: Para qualquer mensagem m , $send(m) \rightarrow receive(m)$
– onde $send(m)$ é o evento de envio da mensagem e $receive(m)$ é o evento de sua receção.

AA3: Se e, e' e e'' são eventos tais que $e \rightarrow e'$ e $e' \rightarrow e''$, então $e \rightarrow e''$.

Assim, se e e e' são eventos e se $e \rightarrow e'$, então, podemos encontrar uma série de eventos e_1, e_2, \dots, e_n que ocorrem em um ou mais processos, tal que $e = e_1$ e $e' = e_n$ e, para $i = 1, 2, \dots, N - 1$, AA1 ou AA2 se aplica entre e_i e e_{i+1} . Isto é, ou eles ocorrem sucessivamente no mesmo processo ou existe uma mensagem m tal que $e_i = send(m)$ e $e_{i+1} = receive(m)$. A sequência de eventos e_1, e_2, \dots, e_n não precisa ser única.

A relação \rightarrow está ilustrada na Figura 14.5 para o caso de três processos p_1, p_2 e p_3 . Pode-se notar que $a \rightarrow b$, pois os eventos ocorrem nessa ordem no processo p_1 ($a \rightarrow_i b$) e, semelhantemente, $c \rightarrow d$. Além disso, $b \rightarrow c$, pois esses eventos são o envio e a receção da mensagem m_1 e, semelhantemente, $d \rightarrow f$. Combinando essas relações, também podemos dizer que, por exemplo, $a \rightarrow f$.

Na Figura 14.5, também pode-se perceber que nem todos os eventos estão relacionados pela relação \rightarrow . Por exemplo, $a \not\rightarrow e$ e $e \not\rightarrow a$, pois eles ocorrem em diferentes

processos e não existe nenhum encadeamento de mensagens entre eles. Dizemos que eventos como a e e , que não são ordenados por \rightarrow , são concorrentes, e escrevemos isso como $a \parallel e$.

A relação \rightarrow captura um fluxo de dados entre dois eventos. Note, entretanto, que, em princípio, os dados podem fluir de maneiras diferentes da passagem de mensagens. Por exemplo, se Smith insere um comando em seu processo para enviar uma mensagem e , então, telefona para Jones, que faz seu processo enviar outra mensagem, então o envio da primeira mensagem claramente *aconteceu antes* da segunda. Infelizmente, como, para essa coordenação, nenhuma mensagem de rede foi enviada entre os processos emitentes, não podemos modelar esse tipo de relacionamento em nosso sistema.

Outro ponto a notar é que, se a relação acontece antes vale entre dois eventos, então o primeiro poderia ou não ter causado o segundo. Por exemplo, se um servidor recebe uma mensagem de requisição e , subsequentemente, envia uma resposta, então claramente a transmissão da resposta é causada pela transmissão da requisição. Entretanto, a relação \rightarrow captura apenas a causalidade em potencial, e dois eventos podem estar relacionados por \rightarrow , mesmo que não exista nenhuma conexão real entre eles. Um processo poderia, por exemplo, receber uma mensagem e , subsequentemente, enviar outra mensagem, mas que ele envia a cada cinco minutos, sem nenhuma relação específica com a primeira mensagem. Nenhuma causalidade real foi envolvida, mas a relação \rightarrow ordenaria esses eventos.

Relógios lógicos • Lamport inventou um mecanismo simples por meio do qual a ordenação acontece antes pode ser capturada numericamente, chamado de *relógio lógico*. O relógio lógico de Lamport é um contador de *software* que aumenta a contagem monotonicamente e cujo valor não precisa ter nenhum relacionamento em particular com qualquer relógio físico. Cada processo p_i mantém seu próprio relógio lógico, L_i , que utiliza para aplicar os conhecidos *carimbos de tempo de Lamport* nos eventos. Denotamos o carimbo de tempo do evento e em p_i como $L_i(e)$ e com $L(e)$, o carimbo de tempo do evento e no processo em que ela ocorreu.

Para capturar a relação acontece antes \rightarrow , os processos atualizam seus relógios lógicos e transmitem os valores de seus relógios lógicos em mensagens, como segue:

RL1: L_i é incrementado antes da ocorrência de um evento no processo p_i :

$$L_i := L_i + 1$$

- RL2:
- (a) Quando um processo p_i envia uma mensagem m , m leva “de carona” (*piggybacking*) o valor $t = L_i$.
 - (b) Na recepção (m, t) , um processo p_j calcula $L_j; \max(L_p, t)$ e, então, aplica RL1 antes de registrar o carimbo de tempo do evento $receive(m)$.

Embora incrementemos os relógios por 1, poderíamos ter escolhido qualquer valor positivo. Isso pode ser facilmente mostrado, por indução, em qualquer sequência de eventos relacionando dois eventos e e e' , que $e \rightarrow e' \Rightarrow L(e) < L(e')$.

Note que o inverso não é verdadeiro. Se $L(e) < L(e')$, então não podemos inferir que $e \rightarrow e'$. Na Figura 14.6, ilustramos o uso de relógios lógicos para o exemplo dado na Figura 14.5. Cada um dos processos p_1, p_2 e p_3 tem seu relógio lógico inicializado em 0. Os valores de relógio dados são aqueles imediatamente após o evento ao qual eles são adjacentes. Note que, por exemplo, $L(b) > L(e)$, mas $b \parallel e$.

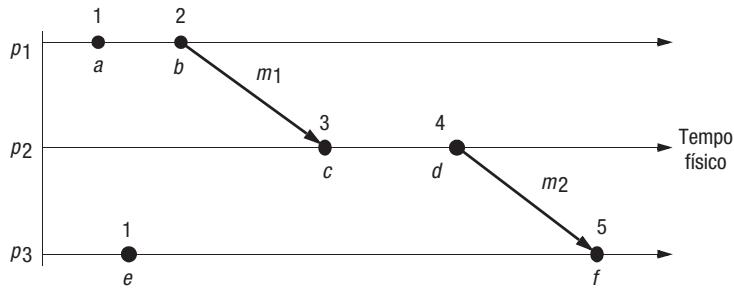


Figura 14.6 Carimbos de tempo de Lamport para os eventos mostrados na Figura 14.5.

Relógios lógicos totalmente ordenados • Alguns pares de eventos distintos, gerados por diferentes processos, têm carimbos de tempo de Lamport numericamente idênticos. Entretanto, podemos criar uma ordem total nos eventos – isto é, uma ordem para a qual todos os pares de eventos distintos são ordenados levando em conta os identificadores dos processos em que os eventos ocorrem. Se e é um evento ocorrendo em p_i com carimbo de tempo local T_i e e' é um evento ocorrendo em p_j com carimbo de tempo local T_j , definimos os carimbos de tempo globais lógicos para esses eventos como (T_i, i) e (T_j, j) , respectivamente. E definimos $(T_i, i) < (T_j, j)$, se e somente se $T_i < T_j$ ou $T_i = T_j$ e $i < j$. Essa ordenação não tem nenhum significado físico geral (pois os identificadores de processo são arbitrários), mas às vezes ela é útil. Lamport a utilizou, por exemplo, para ordenar a entrada de processos em uma seção crítica.

Relógios vetoriais • Mattern [1989] e Fidge [1991] desenvolveram relógios vetoriais para superar a deficiência dos relógios de Lamport: o fato de que, a partir de $L(e) < L(e')$, não podemos concluir que $e \rightarrow e'$. Um relógio vetorial para um sistema de N processos é um vetor de N inteiros. Cada processo mantém seu próprio relógio vetorial V_i , o qual utiliza para gerar carimbos de tempo dos eventos locais. Assim como nos carimbos de tempo de Lamport, os processos levam “de carona” os carimbos de tempo vetoriais nas mensagens que trocam entre si e existem regras simples para atualizar os relógios, como segue:

- RV1: Inicialmente, $V_i[j] = 0$, para $i, j = 1, 2, \dots, N$.
- RV2: Imediatamente antes de p_i gerar o carimbo de tempo de um evento, ele configura $V_i[i] := V_i[i] + 1$.
- RV3: p_i inclui o valor $t = V_i$ em cada mensagem que envia.
- RV4: Quando p_i recebe um carimbo de tempo t em uma mensagem, ele configura $V_i[j] := \max(V_i[j], t[j])$, para $j = 1, 2, \dots, N$. Considerar o máximo de duas componentes de carimbos de tempo vetoriais dessa maneira é conhecido como operação de *merge* (integração).

Para um relógio vetorial V_i , $V_i[i]$ é o número dos eventos em que p_i indicou o carimbo de tempo e $V_i[j]$ ($j \neq i$) é o número dos eventos ocorridos em p_j nos quais p_i potencialmente foi afetado. (O processo p_j pode ter indicado o carimbo de tempo de mais eventos nesse ponto, mas ainda nenhuma informação sobre eles fluiu para p_i nas mensagens.)

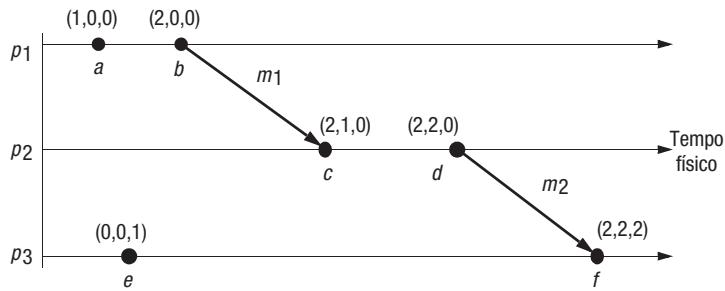


Figura 14.7 Carimbos de tempo vetoriais para os eventos mostrados na Figura 14.5.

Podemos comparar carimbos de tempo vetoriais, como segue:

$$V = V' \text{ sse } V[j] = V'[j] \text{ para } j = 1, 2, \dots, N$$

$$V \leq V' \text{ sse } V[j] \leq V'[j] \text{ para } j = 1, 2, \dots, N$$

$$V < V' \text{ sse } V \leq V' \wedge V \neq V'$$

Seja $V(e)$ o carimbo de tempo vetorial aplicado pelo processo em que e ocorre. É simples mostrar, por indução, no comprimento de qualquer sequência de eventos relacionados a dois eventos e e e' , que $e \rightarrow e' \Rightarrow V(e) < V(e')$. O Exercício 10.13 leva o leitor a mostrar o inverso: se $V(e) < V(e')$, então, $e \rightarrow e'$.

A Figura 14.7 mostra os carimbos de tempo vetoriais dos eventos da Figura 14.5. Pode-se notar, por exemplo, que $V(a) < V(f)$, que reflete o fato de que $a \rightarrow f$. Analogamente, podemos identificar quando dois eventos são concorrentes comparando seus carimbos de tempo. Por exemplo, que $c \parallel e$ pode ser visto dos fatos de que nem $V(c) \leq V(e)$ nem $V(e) \leq V(c)$.

Os carimbos de tempo vetoriais têm a desvantagem, comparadas com os carimbos de tempo de Lamport, de ocupar espaço de armazenamento e carga útil de mensagem proporcional a N , o número de processos. Charron-Bost [1991] mostrou que, se somos capazes de identificar se dois eventos são concorrentes ou não, inspecionando seus carimbos de tempo, então a dimensão N é inevitável. Entretanto, existem técnicas para armazenar e transmitir volumes de dados menores, à custa do processamento necessário para reconstruir as versões completas dos vetores. Raynal e Singhal [1996] apresentam uma narrativa sobre algumas dessas técnicas. Eles também descrevem a noção de *relógios de matriz*, na qual os processos mantêm estimativas dos tempos vetoriais de outros processos, assim como de seus próprios tempos.

14.5 Estados globais

Nesta e na próxima seção, examinaremos o problema de descobrir se uma propriedade em particular é verdadeira em um sistema distribuído quando ele a executa. Começaremos dando os exemplos da coleta de lixo distribuída, da detecção de impasses, da detecção de término e da depuração de programas:

Coleta de lixo distribuída: um objeto é considerado lixo se não existem mais referências a ele em nenhuma parte do sistema distribuído. A memória ocupada por esse objeto pode ser reivindicada quando ele for reconhecido como lixo. Para veri-

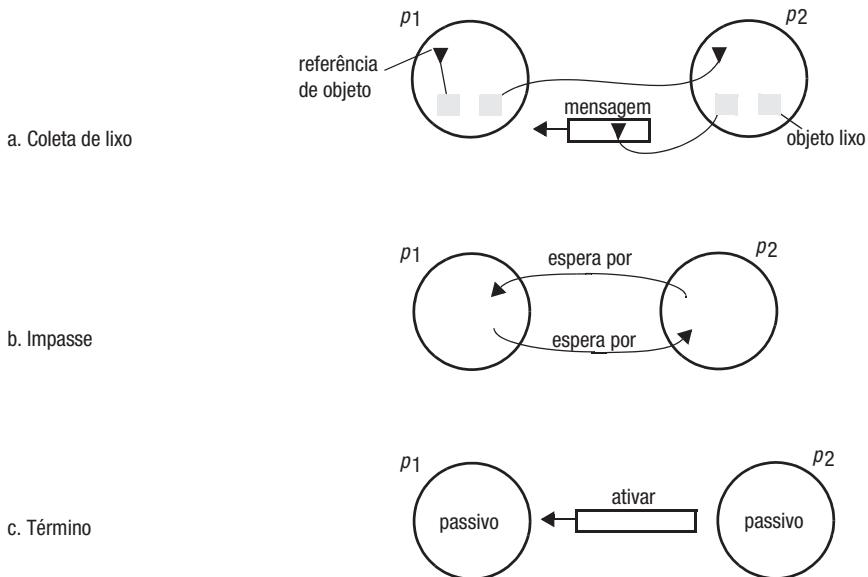


Figura 14.8 Detectando propriedades globais.

ficar se um objeto é lixo, devemos ver se não existem referências a ele em nenhuma parte do sistema. Na Figura 14.8a, o processo p_1 tem dois objetos, ambos com referências – um deles é referenciado localmente pelo próprio p_1 e o outro é referenciado remotamente por p_2 . O processo p_2 tem um objeto que é lixo, sem referências a ele em nenhuma parte do sistema. Ele também tem um objeto para o qual nem p_1 , nem p_2 , tem uma referência, mas existe uma referência a ele em uma mensagem que está em trânsito entre os processos. Isso mostra que, quando consideramos as propriedades de um sistema, devemos incluir o estado dos canais de comunicação, assim como o estado dos processos.

Detecção de impasses distribuída: um impasse (*deadlock*) distribuído ocorre quando cada processo de uma coleção de processos espera que outro envie uma mensagem para ele, e há um ciclo no grafo desse relacionamento “espera por”. A Figura 14.8b mostra que cada um dos processos p_1 e p_2 espera por uma mensagem do outro; portanto, esse sistema nunca fará progresso.

Detecção de término distribuída: o problema aqui é detectar se um algoritmo distribuído terminou. Detectar o término é um problema que parece enganosamente fácil de resolver: à primeira vista, parece que é necessário apenas testar se cada processo parou. Para ver que isso não é assim, considere um algoritmo distribuído executado por dois processos p_1 e p_2 , cada um dos quais podendo solicitar valores um do outro. Instantaneamente, podemos verificar que um processo ou está ativo ou está passivo – um processo passivo não está envolvido em nenhuma atividade propriamente dita, mas está preparado para responder com um valor solicitado pelo outro. Suponha que tenhamos descoberto que p_1 e p_2 são passivos (Figura 14.8c). Para ver que não podemos concluir que o algoritmo terminou, considere o seguinte cenário: quando testamos o estado de p_1 (passivo, no caso), havia uma mensagem a caminho para ele emitida por p_2 , o qual se tornou passivo imediatamente após

enviá-la. Ao receber a mensagem, p_1 se tornou novamente ativo – após descobrirmos que ele era passivo. O algoritmo não tinha terminado.

Os fenômenos do término e do impasse são semelhantes sob alguns aspectos, mas trata-se de problemas diferentes. Primeiro, um impasse pode afetar apenas um subconjunto dos processos em um sistema, enquanto o término considera todos os processos. Segundo, o estado ativo/passivo de um processo não é o mesmo que esperar em um ciclo de impasse: um processo em impasse está tentando executar uma ação, pela qual outro processo espera; um processo passivo não está envolvido em nenhuma atividade.

Depuração distribuída: os sistemas distribuídos são complicados de depurar [Bonnaire *et al.* 1995] e é preciso cuidado ao se estabelecer o que ocorreu durante a execução. Por exemplo, Smith escreveu uma aplicação na qual cada processo p_i contém uma variável x_i ($i = 1, 2, \dots, N$). As variáveis mudam à medida que o programa executa, mas são obrigadas a estar sempre dentro de um valor δ uma da outra. Infelizmente, há um erro no programa e ele suspeita que, sob certas circunstâncias, $|x_i - x_j| > \delta$ para alguns valores de i e j , violando suas restrições de consistência. Seu problema é que esse relacionamento deve ser avaliado quanto aos valores das variáveis que ocorrem ao mesmo tempo.

Cada um dos problemas anteriores tem soluções específicas, adequadas a eles, mas todos ilustram a necessidade de observar um estado global e, portanto, motivam uma estratégia geral.

14.5.1 Estados globais e cortes consistentes

Em princípio, é possível observar a sucessão de estados de um processo individual, mas a questão de como verificar um estado global do sistema – o estado do conjunto de processos – é muito mais difícil de resolver.

O problema básico é a ausência de um tempo global. Se todos os processos tivessem relógios perfeitamente sincronizados, poderíamos concordar com um tempo no qual cada processo registraria seu estado – o resultado seria um estado global real do sistema. A partir do conjunto de estados de processo, poderíamos identificar, por exemplo, se os processos estariam em um impasse. No entanto, não podemos obter uma sincronização perfeita do relógio; portanto, esse método não está disponível para nós.

Assim, poderíamos perguntar se é possível reunir um estado global significativo a partir dos estados locais gravados em diferentes tempos reais. A resposta é um categórico “sim”, mas para ver isso, primeiro apresentaremos algumas definições.

Voltemos ao nosso sistema geral \wp de N processos p_i ($i = 1, 2, \dots, N$), cuja execução queremos estudar. Dissemos anteriormente que uma série de eventos ocorre em cada processo e que podemos caracterizar a execução de cada processo por seu histórico:

$$\text{histórico}(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \dots \rangle$$

Analogamente, podemos considerar qualquer prefixo finito do histórico do processo:

$$h_i^k = \langle e_i^0, e_i^1, \dots, e_i^k \rangle$$

Cada evento é uma ação interna do processo (por exemplo, a atualização de uma de suas variáveis) ou o envio e a recepção de mensagens pelos canais de comunicação que ligam os processos.

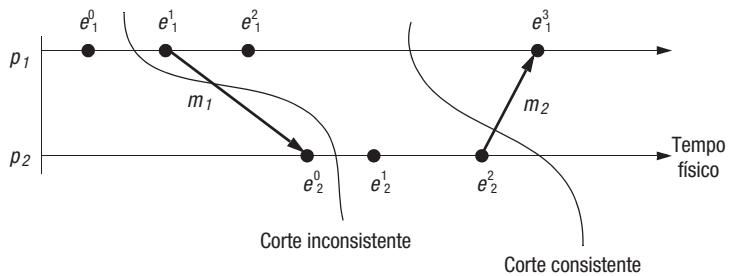


Figura 14.9 Cortes.

Em princípio, podemos gravar o que ocorreu na execução de φ . Cada processo pode gravar os eventos que ocorrem e a sucessão de estados pelos quais passa. Denotamos por s_i^k o estado do processo p_i imediatamente antes que o k -ésimo evento ocorra, de modo que s_i^0 é o estado inicial de p_i . Notamos nos exemplos anteriores que o estado dos canais de comunicação, às vezes, é relevante. Em vez de introduzir um novo tipo de estado, fazemos os processos registrarem o envio ou a recepção de todas as mensagens como parte de seu estado. Se descobrirmos que o processo p_i gravou o envio de uma mensagem m para o processo p_j ($i \neq j$), então, examinando se p_j recebeu essa mensagem, podemos deduzir se m faz parte do estado do canal entre p_i e p_j ou não.

Também podemos formar o histórico global de φ como a união dos históricos de processo individuais:

$$H = h_0 \cup h_1 \cup \dots \cup h_{N-1}$$

Matematicamente, podemos pegar qualquer conjunto de estados dos processos individuais para formar um estado global $S = (s_1, s_2, \dots, s_N)$, mas quais estados globais são significativos – isto é, quais estados de processo poderiam ter ocorrido ao mesmo tempo? Um estado global corresponde aos prefixos iniciais dos históricos de processo individuais. Um *corte* da execução de um sistema é um subconjunto de seu histórico global que é uma união de prefixos de históricos de processo:

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \dots \cup h_N^{c_N}$$

O estado s_i no estado global S correspondente ao corte C é o de p_i imediatamente após o último evento processado por p_i no corte – $e_i^{c_i}$ ($i = 1, 2, \dots, N$). O conjunto de eventos $\{e_i^{c_i} : i = 1, 2, \dots, N\}$ é chamado de *fronteira* do corte.

Considere os eventos que ocorrem nos processos p_1 e p_2 , mostrados na Figura 14.9. A figura mostra dois cortes, um com fronteira $\langle e_1^0, e_2^0 \rangle$ e outro com fronteira $\langle e_2^2, e_2^2 \rangle$. O corte da esquerda é *inconsistente*. Isso porque, em p_2 , ele inclui a recepção da mensagem m_1 , mas p_1 não inclui o envio dessa mensagem. Isso está mostrando um efeito sem causa. A execução real nunca esteve em um estado global que corresponda aos estados do processo nessa fronteira e, em princípio, podemos identificar isso examinando a relação \rightarrow entre os eventos. Em contraste, o corte da direita é *consistente*. Ele inclui tanto o envio como a recepção da mensagem m_1 . Ele inclui o envio, mas não a recepção da mensagem m_2 . Isso é consistente com a execução real – afinal, a mensagem levou algum tempo para chegar.

Um corte C é consistente se, para cada evento que ele contém, ele também contém todos os eventos que aconteceram antes desse evento:

Para todos os eventos $e \in C, f \rightarrow e \Rightarrow f \in C$

Um *estado global consistente* é aquele que corresponde a um corte consistente. Podemos caracterizar a execução de um sistema distribuído como uma série de transições entre os estados globais do sistema:

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \dots$$

Em cada transição, precisamente um evento ocorre em algum único processo no sistema. Esse evento é o envio de uma mensagem, a recepção de uma mensagem ou um evento interno. Se dois eventos acontecerem simultaneamente, ainda assim podemos considerar que eles ocorreram em uma ordem definida – digamos, ordenados de acordo com os identificadores de processo. (Os eventos que ocorrem simultaneamente devem ser concorrentes: nenhum aconteceu antes do outro.) Um sistema evolui dessa maneira pelos estados globais consistentes.

Uma *série (run)* é a ordenação total de todos os eventos em um histórico global consistente com cada ordem do histórico local, \emptyset_i ($i = 1, 2, \dots, N$). Uma *linearização* ou *série consistente* é uma ordenação dos eventos em um histórico global consistente com essa relação acontece antes → em H . Note que uma linearização também é uma série.

Nem todas as séries passam pelos estados globais consistentes, mas todas as linearizações passam apenas pelos estados globais consistentes. Dizemos que um estado S' *pode ser atingido* a partir de um estado S , se há uma linearização que passa por S e depois por S' .

Às vezes, podemos alterar a ordem dos eventos concorrentes dentro de uma linearização e derivar uma série que ainda passa somente pelos estados globais consistentes. Por exemplo, se dois eventos sucessivos em uma linearização são a recepção de mensagens por dois processos, então podemos trocar a ordem desses dois eventos.

14.5.2 Predicados de estado global, estabilidade, segurança e subsistência

Detectar uma condição como um impasse ou término significa avaliar o *predicado de um estado global*. O predicado de um estado global é uma função que faz o mapeamento do conjunto de estados globais de processos no sistema \wp para $\{\text{Verdadeiro}, \text{Falso}\}$. Uma das características úteis dos predicados, associada ao estado de um objeto ser lixo, do sistema estar em um impasse, ou do sistema ter terminado, é que todos são estáveis: quando o sistema entra em um estado no qual o predicado é *Verdadeiro*, ele permanece *Verdadeiro* em todos os estados futuros que podem ser atingidos a partir desse estado. Em contraste, quando monitoramos ou depuramos uma aplicação, estamos frequentemente interessados nos predicados não estáveis, como o nosso exemplo de variáveis cuja diferença supostamente é limitada. Mesmo que a aplicação atinja um estado no qual o limite seja obtido, ele não precisa permanecer nesse estado.

Também notamos, aqui, mais duas noções relevantes aos predicados do estado global: segurança (*safety*) e subsistência (*liveness*). Suponha que exista uma propriedade indesejável α , que é um predicado do estado global do sistema – por exemplo, α poderia ser a propriedade de estar em um impasse. Seja S_0 o estado original do sistema. A *segurança* com relação a α é a afirmação de que α é avaliado como *Falso* para todos os estados S que podem ser atingidos a partir de S_0 . Inversamente, seja β uma propriedade

desejável do estado global de um sistema – por exemplo, a propriedade de atingir o término. A *subsistência* com relação à β é a propriedade de que, para qualquer linearização L começando no estado S_0 , β é avaliado como *Verdadeiro* para algum estado S_L que pode ser atingido a partir de S_0 .

14.5.3 O algoritmo do instantâneo de Chandy e Lamport

Chandy e Lamport [1985] descrevem o *algoritmo do instantâneo (snapshot)* para determinar os estados globais de sistemas distribuídos, o qual apresentaremos agora. O objetivo do algoritmo é gravar um conjunto de estados de processo e do canal (um “instantâneo”) para um conjunto de processos p_i ($i = 1, 2, \dots, N$) tal que, mesmo que a combinação dos estados gravados nunca possa ter ocorrido ao mesmo tempo, o estado global gravado é consistente.

Veremos que o estado gravado pelo algoritmo do instantâneo tem propriedades convenientes para avaliar predicados globais estáveis.

O algoritmo grava o estado de forma local nos processos; ele não fornece um método para agrupar o estado global em um *site*. Um método óbvio de agrupar é fazer todos os processos enviarem o estado que gravaram para um processo coletor designado, mas não vamos tratar desse problema aqui.

O algoritmo presume que:

- nem os canais, nem os processos falham; a comunicação é confiável, de modo que toda mensagem enviada é recebida intacta, exatamente uma vez;
- os canais são unidirecionais e fornecem entrega de mensagens com ordenamento FIFO;
- o grafo de processos e canais é fortemente conectado (existe um caminho entre quaisquer dois processos);
- qualquer processo pode iniciar um instantâneo global a qualquer momento;
- enquanto o instantâneo ocorre, os processos podem continuar sua execução e enviar e receber mensagens normalmente.

Para cada processo p_i , sejam os *canais de entrada* aqueles pelos quais p_i recebe mensagens de outros processos; e os *canais de saída* aqueles nos quais p_i envia mensagens para outros processos. A ideia básica do algoritmo é a seguinte: cada processo registra seu estado e, também, para cada canal de entrada, um conjunto de mensagens recebidas nele. Para cada canal, o processo grava as mensagens que chegaram depois dele ter gravado seu estado e antes que o remetente tenha gravado seu próprio estado. Essa organização nos permite gravar os estados dos processos em diferentes momentos, mas leva em conta as distinções entre os estados de processo em termos de mensagens transmitidas mas ainda não recebidas. Se o processo p_i tiver enviado uma mensagem m para o processo p_j , mas p_j não a tiver recebido, então consideramos m como pertencente ao estado do canal entre eles.

O algoritmo segue com o uso de mensagens especiais de *marcador*, as quais são distintas das outras mensagens que os processos enviam e recebem e podem ser enviadas e recebidas enquanto os processos continuam normalmente sua execução. O marcador tem dupla função: como aviso para o receptor salvar seu próprio estado, caso ele ainda não tenha feito isso; e como uma maneira de determinar quais mensagens devem ser incluídas no estado do canal.

Regra de recepção de marcador do processo p_i

Na recepção por parte de p_i de uma mensagem de marcador pelo canal c :

```

if ( $p_i$  ainda não tiver gravado seu estado) ele
    grava seu estado de processo atual;
    grava o estado de  $c$  como o conjunto vazio;
    ativa a gravação de mensagens que chegam por outros canais de entrada;
else
     $p_i$  grava o estado de  $c$  como o conjunto de mensagens que recebeu por  $c$ 
    desde que salvou seu estado.
end if

```

Regra de envio de marcador do processo p_i

Após p_i ter gravado seu estado, para cada canal de saída c :

```

 $p_i$  envia uma mensagem de marcador por  $c$ 
(antes de enviar qualquer outra mensagem por  $c$ ).

```

Figura 14.10 Algoritmo do instantâneo de Chandy e Lamport.

O algoritmo é definido por meio de duas regras: a *regra de recepção de marcador* e a *regra de envio de marcador* (Figura 14.10). A regra de envio de marcador obriga os processos a enviarem um marcador após terem gravado seus estados, mas antes de enviarem quaisquer outras mensagens.

A regra de recepção de marcador obriga um processo que não gravou seu estado a fazer isso. Nesse caso, esse é o primeiro marcador recebido. Ele observa quais mensagens chegam posteriormente nos outros canais de entrada. Quando um processo que já salvou seu estado recebe um marcador (em outro canal), ele grava o estado desse canal como o conjunto de mensagens recebidas nele, desde que salvou seu estado.

Qualquer processo pode iniciar o algoritmo a qualquer momento. Ele age como se tivesse recebido um marcador (por um canal inexistente) e segue a regra de recepção de marcador. Assim, ele grava seu estado e começa a gravar as mensagens que chegam por todos os seus canais de entrada. Dessa maneira, vários processos podem iniciar a gravação concorrentemente (desde que os marcadores que utilizem possam ser distinguidos).

Ilustramos o algoritmo para um sistema de dois processos, p_1 e p_2 , conectados por dois canais unidirecionais, c_1 e c_2 . Os dois processos negociam “coisas”. O processo p_1 envia requisição de “uma coisa” para p_2 , por c_2 , contendo um pagamento na razão de \$10 por “coisa”. Algum tempo depois, o processo p_2 envia “coisas” para p_1 pelo canal c_1 . Os processos têm os estados iniciais mostrados na Figura 14.11. O processo p_2 já recebeu uma requisição de cinco “coisas”, as quais serão rapidamente enviadas para p_1 .

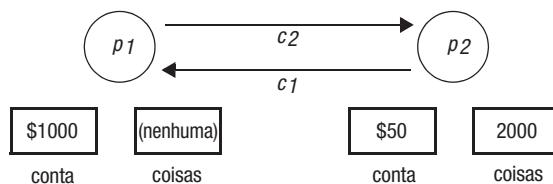


Figura 14.11 Dois processos e seus estados iniciais.

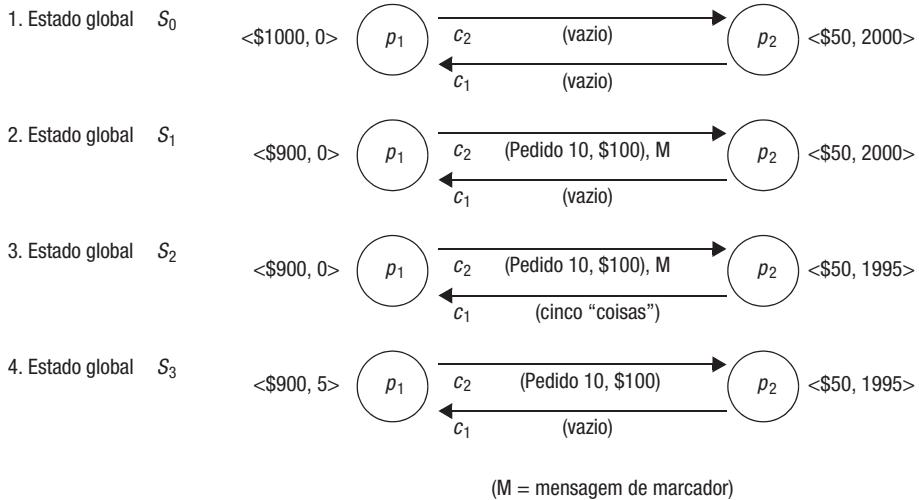


Figura 14.12 A execução dos processos da Figura 14.11.

A Figura 14.12 mostra uma execução do sistema enquanto o estado é gravado. O processo p_1 grava seu estado no estado global real S_0 , quando o estado de p_1 é $\langle \$1000, 0 \rangle$. Seguindo a regra de envio de marcador, o processo p_1 emite, então, uma mensagem de marcador por seu canal de saída c_2 , antes de enviar a próxima mensagem em nível de aplicação: (Pedido 10, \$100) pelo canal c_2 . O sistema entra no estado global S_1 .

Antes que p_2 receba o marcador, ele emite uma mensagem de aplicação (cinco “coisas”) por c_1 , em resposta ao pedido anterior de p_1 , gerando um novo estado global S_2 .

Agora, o processo p_1 recebe a mensagem de p_2 (cinco “coisas”) e p_2 recebe o marcador. Seguindo a regra de recepção de marcador, p_2 grava seu estado como $\langle \$50, 1995 \rangle$ e o do canal c_2 como a sequência vazia. Pela regra de envio de marcador, p_2 envia uma mensagem de marcador por c_1 .

Quando o processo p_1 recebe a mensagem de marcador de p_2 , ele grava o estado do canal c_1 como a única mensagem (cinco “coisas”) que recebeu após ter gravado seu estado pela primeira vez. O estado global final é S_3 .

O estado gravado final é $p_1: \langle \$1000, 0 \rangle; p_2: \langle \$50, 1995 \rangle; c_1: \langle \text{(cinco “coisas”)} \rangle; c_2: \langle \rangle$. Note que esse estado difere de todos os estados globais pelos quais o sistema realmente passou.

Término do algoritmo do instantâneo • Suponhamos que um processo que recebeu uma mensagem de marcador grave seu estado e envie mensagens de marcador através de cada canal de saída dentro de um tempo finito (mesmo quando não precisa mais enviar mensagens de aplicação por esses canais). Se houver um caminho de canais de comunicação de um processo p_i para um processo p_j ($j \neq i$), fica claro, a partir dessas suposições, que p_j gravará seu estado em um tempo finito após p_i ter gravado seu estado. Como estamos supondo que o grafo de processos e canais é fortemente conectado, segue-se que todos os processos terão gravado seus estados e os estados dos canais de entrada em um tempo finito após, inicialmente, um processo gravar seu estado.

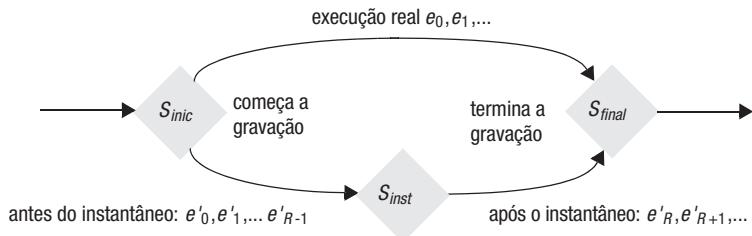


Figura 14.13 Alcançabilidade de estados no algoritmo do instantâneo.

Caracterização do estado observado • O algoritmo do instantâneo seleciona um corte a partir do histórico da execução. O corte e , portanto, o estado gravado por esse algoritmo, são consistentes. Para perceber isso, sejam e_i e e_j eventos que ocorrem em p_i e p_j respectivamente, tal que $e_i \rightarrow e_j$. afirmamos que, se e_j está no corte, então e_i está no corte. Isto é, se e_j ocorreu antes de p_j ter gravado seu estado, então e_i deve ter ocorrido antes que p_j gravasse seu estado. Isso é evidente se os dois processos são o mesmo; portanto, vamos supor que $j \neq i$. Suponha, por enquanto, o oposto do que queremos provar: que p_i gravou seu estado antes que e_i ocorresse. Considere a sequência de mensagens de $H m_1, m_2, \dots, m_H$ ($H \geq 1$), dando origem à relação $e_i \rightarrow e_j$. Pela ordem FIFO nos canais que essas mensagens atravessam, e pelas regras de envio e recepção de marcador, uma mensagem de marcador teria chegado a p_j na frente de m_1, m_2, \dots, m_H . Pela regra de recepção de marcador, p_j teria, portanto, gravado seu estado antes do evento e_j . Isso contradiz nossa suposição de que e_j está no corte, e terminamos.

Podemos estabelecer ainda uma relação de alcance entre o estado global observado e os estados globais iniciais e finais, quando o algoritmo é executado. Seja $Sys = e_0, e_1, \dots$ a linearização do sistema ao ser executado (onde dois eventos ocorreram exatamente no mesmo tempo são ordenados de acordo com os identificadores de processo). Seja S_{inic} o estado global imediatamente antes que o primeiro processo tenha gravado seu estado; seja S_{final} o estado global de quando o algoritmo do instantâneo termina, imediatamente após a ação de gravação do último estado; e seja S_{inst} o estado global gravado.

Encontraremos uma permutação de Sys , $Sys' = e'_0, e'_1, e'_2, \dots$ tal que todos os três estados S_{inic} , S_{inst} e S_{final} ocorrem em Sys' , S_{inst} pode ser atingido a partir de S_{inic} em Sys' e S_{final} pode ser atingido a partir de S_{inst} em Sys' . A Figura 14.13 mostra essa situação, na qual a linearização superior é Sys e a linearização inferior é Sys' .

Derivamos Sys' de Sys , primeiro classificando todos os eventos em Sys como eventos *anteriores ao instantâneo* ou eventos *posteriores ao instantâneo*. Um evento anterior ao instantâneo no processo p_i é aquele que ocorreu em p_i antes que ele gravasse seu estado; todos os outros eventos são posteriores ao instantâneo. É importante entender que um evento posterior ao instantâneo pode ocorrer antes de um evento anterior ao instantâneo em Sys , caso os eventos ocorram em diferentes processos. (É claro que nenhum evento posterior ao instantâneo pode ocorrer antes de um evento anterior ao instantâneo no mesmo processo.)

Mostraremos como podemos ordenar todos os eventos anteriores ao instantâneo antes dos eventos posteriores ao instantâneo, para obtermos Sys' . Suponha que e_j seja um evento posterior ao instantâneo em um processo, e que e_{j+1} seja um evento anterior ao instantâneo em um processo diferente. Não pode ser que $e_j \rightarrow e_{j+1}$. Para isso, esses dois eventos seriam o envio e a recepção de uma mensagem, respectivamente. Uma mensa-

gem de marcador teria de ter precedido a mensagem, tornando a recepção da mensagem um evento posterior ao instantâneo, mas, por suposição, e_{j+1} é um evento anterior ao instantâneo. Portanto, podemos trocar os dois eventos sem violar a relação acontece antes (isto é, a sequência resultante de eventos continua sendo uma linearização). A troca não introduz novos estados de processo, pois não alteramos a ordem em que os eventos ocorrem em nenhum processo individual.

Continuamos a trocar pares de eventos adjacentes dessa maneira, conforme for necessário, até termos ordenado todos os eventos anteriores ao instantâneo $e'_0, e'_1, e'_2, \dots, e'_{R-1}$, antes de todos os eventos posteriores ao instantâneo $e'_{R-1}, e'_R, e'_{R+2}, \dots$, com Sys' sendo a execução resultante. Para cada processo, o conjunto de eventos em $e'_0, e'_1, e'_2, \dots, e'_{R-1}$ que ocorreram nele é exatamente o conjunto de eventos que ele experimentou antes de ter gravado seu estado. Portanto, o estado de cada processo nesse ponto e o estado dos canais de comunicação é o do estado global S_{inst} gravado pelo algoritmo. Não perturbamos nenhum dos dois estados S_{inic} ou S_{final} com os quais a linearização começa e termina. Portanto, estabelecemos a relação de alcançabilidade.

Estabilidade e alcançabilidade do estado observado • A propriedade de alcançabilidade do algoritmo do instantâneo é útil para detectar predicados estáveis. Em geral, todo predíco não estável que estabelecemos como sendo *Verdadeiro* no estado S_{inst} pode ou não ter sido *Verdadeiro* na execução real cujo estado global gravamos. Entretanto, se um *predíco estável* é *Verdadeiro* no estado S_{inst} , podemos concluir que o predíco é *Verdadeiro* no estado S_{final} pois, por definição, um predíco estável que é *Verdadeiro* em um estado S , também é *Verdadeiro* em qualquer estado que possa ser atingido a partir de S . Analogamente, se o predíco for avaliado como *Falso* para S_{inst} , ele também deverá ser *Falso* para S_{inic} .

14.6 Depuração distribuída

Examinaremos agora o problema da gravação do estado global de um sistema, para que possamos fazer mais declarações úteis a respeito do fato de um estado transitório – em oposição a um estado estável – ter ocorrido em uma execução real. Em geral, é isso que exigimos na depuração de um sistema distribuído. Demos, anteriormente, um exemplo no qual cada processo em um conjunto de processos p_i tem uma variável x_i . A condição de segurança exigida nesse exemplo é $|x_i - x_j| \leq \delta$ ($i, j = 1, 2, \dots, N$); essa restrição deve ser satisfeita mesmo que um processo possa alterar o valor de sua variável a qualquer momento. Outro exemplo é um sistema distribuído controlando um sistema de dutos em uma usina, no qual estamos interessados em saber se todas as válvulas (controladas por diferentes processos) foram abertas em algum momento. Nesses exemplos, em geral, não podemos observar os valores das variáveis ou os estados das válvulas simultaneamente. O desafio é monitorar a execução do sistema com o passar do tempo – para capturar informações de rastreamento, em vez de um único instantâneo –, de modo que possamos estabelecer *post hoc* se a condição de segurança exigida foi, ou pode ter sido, violada.

O algoritmo do instantâneo de Chandy e Lamport reúne o estado de maneira distribuída, e mostramos como os processos no sistema poderiam enviar o estado que agrupam para um processo monitor coletá-las. O algoritmo que vamos descrever (atribuído a Marzullo e Neiger [1991]) é centralizado. Os processos observados enviam seus estados para um processo chamado *monitor*, o qual monta estados globalmente

consistentes a partir do que recebe. Consideramos que o monitor reside fora do sistema, observando sua execução.

Nosso objetivo é determinar os casos em que determinado predicado do estado global ϕ era definitivamente *Verdadeiro* em algum ponto na execução que observamos, e os casos em que ele era possivelmente *Verdadeiro*. A noção de “possivelmente” surge como um conceito natural, pois podemos extrair um estado global consistente S a partir de um sistema em execução, e verificar se $\phi(S)$ é *Verdadeiro*. Nenhuma observação única de um estado global consistente nos permite concluir se um predicado não estável foi avaliado como *Verdadeiro* na execução real. Contudo, podemos estar interessados em saber se eles *poderiam* ter ocorrido, na medida do que podemos identificar observando a execução.

A noção de “definitivamente” se aplica à execução real e não a uma série que extrapolamos a partir dela. Pode parecer paradoxal considerarmos o que aconteceu em uma execução real. Entretanto, é possível avaliar se ϕ era definitivamente *Verdadeiro*, considerando todas as linearizações dos eventos observados.

Agora, definiremos as noções de *possivelmente* ϕ e *definitivamente* ϕ para um predicado ϕ , em termos de linearizações de H , o histórico da execução do sistema.

possivelmente ϕ : a declaração *possivelmente* ϕ significa que existe um estado global consistente S pelo qual passa uma linearização de H tal que $\phi(S)$ é *Verdadeiro*.

definitivamente ϕ : a declaração *definitivamente* ϕ significa que para todas as linearizações L de H existe um estado global consistente S pelo qual L passa, tal que $\phi(S)$ é *Verdadeiro*.

Quando usamos o algoritmo do instantâneo de Chandy e Lamport e obtemos o estado global S_{inst} , podemos declarar *possivelmente* ϕ , se $\phi(S_{inst})$ for *Verdadeiro*. Contudo, em geral, avaliar *possivelmente* ϕ impõe uma busca por todos os estados globais consistentes derivados da execução observada. Somente se $\phi(S)$ for avaliado como *Falso* para todos os estados globais consistentes S não será o caso de *possivelmente* ϕ . Note também que, embora possamos concluir *definitivamente* ($\neg\phi$) a partir de \neg -*possivelmente* ϕ , não podemos concluir \neg -*possivelmente* ϕ a partir de *definitivamente* ($\neg\phi$). Esta última é a afirmação de que $\neg\phi$ vale em algum estado em cada linearização: ϕ pode valer para outros estados.

Agora, descreveremos: como os estados de processo são coletados; como o monitor extrai estados globais consistentes; como o monitor avalia *possivelmente* ϕ e *definitivamente* ϕ em sistemas assíncronos e síncronos.

14.6.1 Coleta do estado

Os processos observados p_i ($i = 1, 2, \dots, N$), inicialmente, enviam seus estados iniciais para o processo monitor e, depois disso, periodicamente, em *mensagens de estado*. O processo monitor grava as mensagens de estado de cada processo p_i em uma fila separada Q_i , para cada $i = 1, 2, \dots, N$.

A atividade de preparar e enviar mensagens de estado pode atrasar a execução normal dos processos observados, mas não interfere nela de outra forma. Não há necessidade de enviar o estado, exceto inicialmente e quando ele muda. Existem duas otimizações para reduzir o tráfego de mensagens de estado no monitor. Primeiro, o predicado do estado global pode depender apenas de certas partes dos estados dos processos. Por exemplo, ele pode depender apenas dos estados de variáveis específicas. Portanto, os processos observados só precisam enviar o estado relevante para o monitor. Segundo, eles só precisam enviar seus estados nos momentos em que o predicado ϕ pode se tornar *Verdadeiro* ou deixar de ser *Verdadeiro*. Não há motivos para enviar alterações no estado que não afetem o valor do predicado.

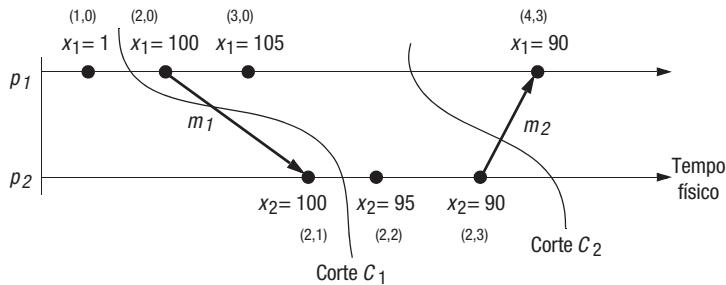


Figura 14.14 Indicações de tempo vetoriais e valores de variável para a execução da Figura 14.9.

Por exemplo, no exemplo de sistema de processos p_i que devem obedecer à restrição $|x_i - x_j| \leq \delta$ ($i = 1, 2, \dots, N$), os processos só precisam notificar o monitor quando os valores de suas próprias varáveis x_i mudam. Quando enviam seus estados, eles fornecem o valor de x_i , mas não precisam enviar nenhuma outra variável.

14.6.2 Observação de estados globais consistentes

O monitor deve montar estados globais consistentes nos quais avaliará ϕ . Lembre-se de que um corte C é consistente se, e somente se, para todos os eventos e no corte C , $f \rightarrow e \Rightarrow e \in C$.

Por exemplo, a Figura 14.14 mostra dois processos p_1 e p_2 com variáveis x_1 e x_2 , respectivamente. Os eventos mostrados nas linhas do tempo (com indicações de tempo vetoriais) são ajustes nos valores das duas variáveis. Inicialmente, $x_1 = x_2 = 0$. O requisito é $|x_1 - x_2| \leq 50$. Os processos fazem ajustes em suas variáveis, mas ajustes “grandes” fazem uma mensagem, contendo o novo valor, ser enviada para o outro processo. Quando um dos dois processos recebe uma mensagem de ajuste do outro, ele configura sua variável com um valor igual ao contido na mensagem.

Quando um dos processos p_1 ou p_2 ajusta o valor de sua variável (seja um ajuste “pequeno” ou “grande”), ele envia o valor em uma mensagem de estado para o monitor. Este último mantém as mensagens de estado em filas por processo, para efeitos de análise. Se o monitor usasse valores do corte inconsistente C_1 da Figura 14.14, então descobriria que $x_1 = 1, x_2 = 100$, violando a restrição $|x_1 - x_2| \leq 50$. Porém, esse estado nunca ocorreu. Por outro lado, valores do corte consistente C_2 mostram $x_1 = 105, x_2 = 90$.

Para que o monitor possa distinguir estados globais consistentes de estados globais inconsistentes, os processos observados incluem seus valores de relógio vetorial com suas mensagens de estado. Cada fila Q_i é mantida na ordem de envio, a qual pode ser imediatamente estabelecida examinando-se o i -ésimo componente dos carimbos de tempo vetoriais. É claro que o monitor não pode deduzir nada sobre a ordem dos estados enviados por processos diferentes a partir de sua ordem de chegada, devido às latências de mensagem variáveis. Em vez disso, ele precisa examinar os carimbos de tempo vetoriais das mensagens de estado.

Seja $S = (s_1, s_2, \dots, s_N)$ um estado global extraído das mensagens de estado recebidas pelo monitor. Seja $V(s_i)$ o carimbo de tempo vetorial do estado s_i recebida de p_i . Então, pode-se mostrar que S é um estado global consistente (EGC) se e somente se:

$$V(s_i)[i] \geq V(s_j)[i] \text{ para } i, j = 1, 2, \dots, N - (\text{Condição EGC})$$

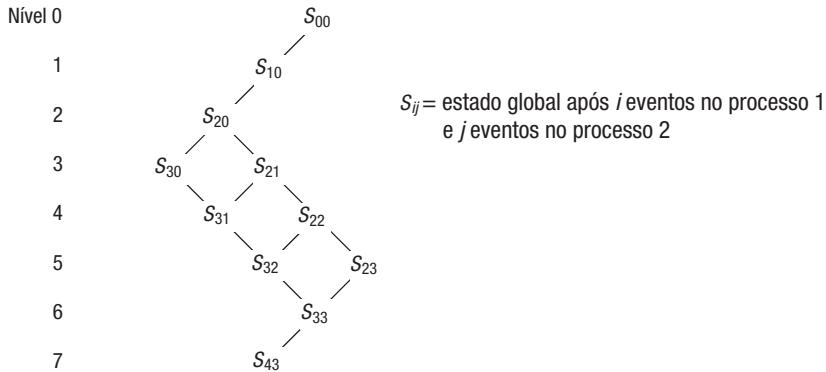


Figura 14.15 Treliça de estados globais para a execução da Figura 14.14.

Isso diz que o número de eventos de p_i , conhecidos em p_j , ao enviar s_j não é maior do que o número de eventos que tinham ocorrido em p_i quando ele enviou s_i . Em outras palavras, se o estado de um processo depende de outro (de acordo com a ordenação acontece antes), então o estado global também abrange o estado do qual ele depende.

Resumindo, agora temos um método pelo qual o monitor pode estabelecer se determinado estado global é consistente, usando os carimbos de tempo vetoriais mantidos pelos processos observados e levados de carona nas mensagens de estado enviadas a ele.

A Figura 14.15 mostra a treliça de estados globais consistentes correspondente à execução dos dois processos da Figura 14.14. Essa estrutura captura a relação de alcanceabilidade entre estados globais consistentes. Os nós denotam estados globais, e os arcos denotam as possíveis transições entre esses estados. O estado global S_{00} tem os dois processos em seu estado inicial; S_{10} tem p_2 ainda em seu estado inicial e p_1 , no próximo estado em seu histórico local. O estado S_{01} não é consistente, devido à mensagem m_1 enviada de p_1 para p_2 ; portanto, ele não aparece na treliça.

A treliça é organizada em níveis com, por exemplo, S_{00} no nível 0 e S_{10} no nível 1. Generalizando, S_{ij} está no nível $(i + j)$. Uma linearização percorre a treliça de qualquer estado global para qualquer estado global que pode ser atingido a partir dele no próximo nível – isto é, em cada etapa, algum processo experimenta um evento. Por exemplo, S_{22} pode ser atingido a partir de S_{20} , mas não pode ser atingido a partir de S_{30} .

A treliça nos mostra todas as linearizações correspondentes a um histórico. Em princípio, está claro como um monitor deve avaliar *possivelmente* ϕ e *definitivamente* ϕ . Para avaliar *possivelmente* ϕ , o monitor começa no estado inicial e percorre todos os estados consistentes que podem ser atingidos a partir desse ponto, avaliando ϕ em cada estágio. Ele para quando ϕ é avaliado como *Verdadeiro*. Para avaliar *definitivamente* ϕ , o monitor precisa tentar descobrir um conjunto de estados através dos quais todas as linearizações devem passar, e em cada um dos quais ϕ é avaliado como *Verdadeiro*. Por exemplo, se, na Figura 14.15, $\phi(S_{30})$ e $\phi(S_{21})$ são *Verdadeiros*, então, como todas as linearizações passam por esses estados, *definitivamente* ϕ é válido.

14.6.3 Avaliação de possivelmente Φ

Para avaliar *possivelmente* ϕ , o monitor precisa percorrer a treliça dos estados que podem ser atingidos, partindo do estado inicial ($s_1^0, s_2^0 \dots s_N^0$). O algoritmo aparece na Figura 14.16

1. Avaliando possivelmente ϕ para o histórico global H de N processos

```

 $L := 0;$ 
Estados := { $(s_1^0, s_2^0, \dots, s_N^0)$ };
while ( $\phi(S) = \text{Falso}$  para todo  $S \in \text{Estados}$ )
     $L := L + 1;$ 
    Alcançável := { $S'$ :  $S'$  pode ser atingido em  $H$  a partir de um  $S \in \text{Estados} \wedge \text{nível}(S') = L$ };
    Estados := Alcançável
end while
output "possivelmente  $\phi$ ";
```

2. Avaliando definitivamente ϕ para o histórico global H de N processos

```

 $L := 0;$ 
if ( $\phi(s_1^0, s_2^0, \dots, s_N^0)$ ) then Estados := {} else Estados := { $(s_1^0, s_2^0, \dots, s_N^0)$ };
while (Estados ≠ {})
     $L := L + 1;$ 
    Alcançável := { $S'$ :  $S'$  pode ser atingido em  $H$  a partir de algum  $S \in \text{Estados} \wedge \text{nível}(S') = L$ };
    Estados := { $S \in \text{Alcançável} \mid \phi(S) = \text{Falso}$ }
end while
output "definitivamente  $\phi$ ";
```

Figura 14.16 Algoritmos para avaliar *possivelmente* Φ e *definitivamente* Φ .

e presume que a execução é infinita. Ele pode ser facilmente adaptado para uma execução finita.

O monitor pode descobrir o conjunto de estados consistente no nível $L + 1$ que podem ser atingidos a partir de determinado estado consistente no nível L , pelo seguinte método. Seja $S = (s_1, s_2, \dots, s_N)$ um estado consistente. Então, um estado consistente no próximo nível, que pode ser atingido a partir de S , tem a forma $S' = (s_1, s_2, \dots, s'_1, \dots, s_N)$, que difere de S somente por conter o próximo estado (após um único evento) de algum processo p_i . O monitor pode encontrar todos esses estados percorrendo as filas de mensagens de estado Q_i ($1, 2, \dots, N$). O estado S' pode ser atingido a partir de S se e somente se:

$$\text{para } j = 1, 2, \dots, N, j \neq i: V(s_j)[j] \geq V(s'_i)[j]$$

Essa condição vem da condição *EGC* anterior e do fato de que S já ser um estado global consistente. Em geral, determinado estado pode ser atingido a partir de vários estados do nível anterior; portanto, o monitor deve tomar o cuidado de avaliar a consistência de cada estado apenas uma vez.

14.6.4 Avaliação de definitivamente Φ

Para avaliar *definitivamente* ϕ , o monitor percorre novamente a treliça de estados que podem ser atingidos, um nível por vez, partindo do estado inicial $(s_1^0, s_2^0, \dots, s_N^0)$. O algoritmo (mostrado na Figura 14.16) novamente presume que a execução é infinita, mas ele pode ser facilmente adaptado para uma execução finita. Ele mantém o conjunto *Estados*, que contém os estados no nível corrente que podem ser atingidos em uma linearização a partir do estado inicial, percorrendo apenas os estados para os quais ϕ é avaliado como *Falso*. Como tal linearização existe, não podemos declarar *definitivamente*

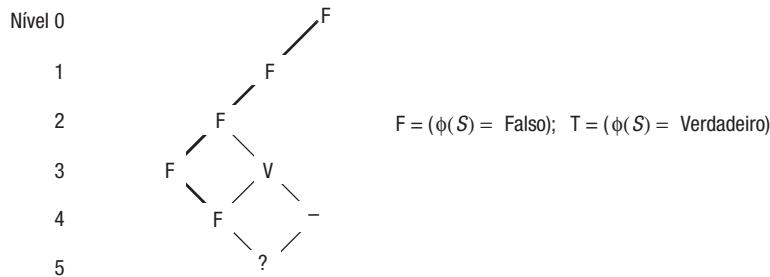


Figura 14.17 Avaliação de *definitivamente* Φ .

mente ϕ : a execução poderia ter pego essa linearização e ϕ seria *Falso* em cada estágio ao longo dela. Se atingirmos um nível para o qual não existe tal linearização, podemos concluir *definitivamente* ϕ .

Na Figura 14.17, no nível 3, o conjunto *Estados* consiste em apenas um estado, que pode ser atingido por uma linearização na qual todos os estados são *Falsos* (marcada com linhas escuras). O único estado considerado no nível 4 é o que está marcado com *F*. (O estado à sua direita não é considerado, pois ele só pode ser atingido por meio de um estado para o qual ϕ é avaliado como *Verdadeiro*.) Se ϕ é avaliado como *Verdadeiro* no estado do nível 5, podemos concluir *definitivamente* ϕ . Caso contrário, o algoritmo deve continuar além desse nível.

Custo • Os algoritmos que acabamos de descrever oferecem uma enormidade de combinações. Suponha que k seja o número máximo de eventos em um único processo. Então, os algoritmos que descrevemos acarretam $O(k^N)$ comparações (o monitor compara entre si os estados de cada um dos N processos observados).

Também existe um custo de espaço de $O(kN)$ nesses algoritmos. Entretanto, observamos que o monitor pode excluir uma mensagem contendo o estado s_i da fila Q_j , quando nenhum outro item de estado que chegue de outro processo possa estar envolvido em um estado global consistente contendo s_i . Isto é, quando:

$$V(s_j^{\text{último}})[i] > V(s_i)[i] \text{ para } j = 1, 2, \dots, N, j \neq i$$

onde $s_j^{\text{último}}$ é o último estado que o monitor recebeu do processo p_j .

14.6.5 Avaliação de possivelmente Φ e definitivamente Φ em sistemas síncronos

Os algoritmos que mostramos até agora funcionam em um sistema assíncrono: não fizemos suposição alguma quanto ao tempo. Contudo, o preço pago por isso é que o monitor pode examinar um estado global consistente $S = (s_1, s_2, \dots, s_N)$ para o qual dois estados locais s_i e s_j ocorreram em um tempo arbitrariamente longo, fora da execução real do sistema. Nosso requisito, em contraste, é considerar apenas os estados globais que a execução real poderia, em princípio, ter percorrido.

Em um sistema síncrono, suponha que os processos mantenham seus relógios físicos internamente sincronizados dentro de um limite conhecido, e que os processos observados forneçam carimbos de tempo físicos, assim como carimbos de tempo vetoriais, em suas mensagens de estado. Então, o monitor precisa considerar apenas os estados globais consistentes cujos estados locais poderiam ter existido simultaneamente,

dado o sincronismo aproximado dos relógios. Com um sincronismo suficientemente bom do relógio, eles serão em muito menor número do que todos os estados globalmente consistentes.

Agora, mostraremos um algoritmo para explorar relógios sincronizados dessa maneira. Supomos que cada processo observado p_i ($i = 1, 2, \dots, N$) e o monitor, que chamaremos de p_0 , mantêm um relógio físico C_i ($i = 0, 1, \dots, N$). Eles são sincronizados dentro de um limite conhecido $D > 0$; isto é, no mesmo tempo real:

$$|C_i(t) - C_j(t)| < D \text{ para } i, j = 0, 1, \dots, N$$

Os dois processos observados enviam seu tempo vetorial e seu tempo físico com suas mensagens de estado para o monitor. Agora, o monitor aplica uma condição que não apenas testa a consistência de um estado global $S = (s_1, s_2, \dots, s_N)$, mas também testa se cada par de estados poderia ter acontecido no mesmo tempo real, dados os valores de relógio físico. Em outras palavras, para $i, j = 1, 2, \dots, N$:

$$V(s_i)[i] \geq V(s_j)[i] \text{ e } s_i \text{ e } s_j \text{ poderiam ter ocorrido no mesmo tempo real.}$$

A primeira cláusula é a condição que usamos anteriormente. Para a segunda cláusula, note que p_i está no estado s_i a partir do momento em que notifica pela primeira vez o monitor, $C_i(s_i)$, até algum tempo local posterior $L_i(s_i)$; digamos, quando ocorre a próxima transição de estado em p_i . Para s_i e s_j terem obtido o mesmo tempo real, temos, então, considerando o limite de sincronização do relógio:

$$C_i(s_i) - D \leq C_j(s_j) \leq L_i(s_i) + D - \text{ou vice-versa (trocando } i \text{ e } j\text{).}$$

O monitor deve calcular um valor para $L_i(s_i)$, o qual é medido no relógio de p_i . Se o monitor tiver recebido de p_i , uma mensagem de estado do próximo estado de s'_i então $L_i(s_i)$ é $C_i(s'_i)$. Caso contrário, o monitor estima $L_i(s_i)$ como $C_0 - \max + D(s_i)$, onde C_0 é o valor corrente do relógio local do monitor e \max é o tempo de transmissão máximo para uma mensagem de estado.

14.7 Resumo

Este capítulo começou descrevendo a importância da indicação precisa do tempo para sistemas distribuídos. Em seguida, descreveu algoritmos para sincronizar relógios apesar da deriva entre eles e da variabilidade dos atrasos de mensagem entre computadores.

O grau da precisão da sincronização que pode realmente ser obtido atende a muitos requisitos, mas não é suficiente para determinar a ordem de dois eventos arbitrários ocorrendo em diferentes computadores. A relação acontece antes é uma ordem parcial dos eventos, que reflete um fluxo de informações entre eles – dentro de um processo, ou por meio de mensagens entre processos. Alguns algoritmos exigem que os eventos sejam colocados na ordem acontece antes; por exemplo, sucessivas atualizações feitas em cópias separadas dos dados. Os relógios de Lamport são contadores atualizados de acordo com o relacionamento acontece antes entre eventos. Os relógios vetoriais são um aprimoramento dos relógios de Lamport, pois é possível determinar, examinando seus carimbos de tempo vetoriais, se dois eventos estão ordenados pelo relacionamento acontece antes ou se são concorrentes.

Apresentamos os conceitos de eventos, históricos locais e globais, cortes, estados locais e globais, séries, estados consistentes, linearizações (séries consistentes) e alcance. Um estado, ou série consistente, é aquele que está de acordo com a relação acontece antes.

Passamos a considerar o problema da gravação de um estado global consistente pela observação da execução de um sistema. Nossa objetivo foi avaliar um predicado nesse estado. Uma classe importante de predicados são os predicados estáveis. Descrevemos o algoritmo do instantâneo de Chandy e Lamport, o qual captura um estado global consistente e nos permite fazer afirmações sobre se um predicado estável vale na execução real. Mostramos o algoritmo de Marzullo e Neiger para inferir afirmações sobre se um predicado vale ou pode ser válido no curso real. O algoritmo emprega um processo monitor para agrupar estados. O monitor examina carimbos de tempo vetoriais para extrair estados globais consistentes e constrói e examina a treliça de todos os estados globais consistentes. Esse algoritmo envolve muita complexidade computacional, mas é valioso entendê-lo; ele pode trazer algum benefício prático em sistemas reais, em que relativamente poucos eventos mudam o valor do predicado global. O algoritmo tem uma variante mais eficiente nos sistemas síncronos, em que os relógios podem ser sincronizados.

Exercícios

- 14.1 Por que a sincronização do relógio do computador é necessária? Descreva os requisitos de projeto de um sistema para sincronizar os relógios em um sistema distribuído. *página 596*
- 14.2 Um relógio está mostrando 10:27:54.0 (hr:min:seg), quando se descobre que ele está adiantado 4 segundos. Explique por que não é desejável configurá-lo com a hora certa nesse ponto e mostre (numericamente) como ele deve ser ajustado de modo a estar correto após terem decorrido 8 segundos. *página 600*
- 14.3 Um esquema para implementar envio de mensagens confiáveis *no máximo uma vez* usa relógios sincronizados para rejeitar mensagens duplicadas. Os processos colocam o valor de seus relógios locais (um “carimbo de tempo”) nas mensagens que enviam. Cada receptor mantém uma tabela fornecendo, para cada processo remetente, o maior carimbo de tempo de mensagem que observou. Suponha que os relógios estejam sincronizados dentro de 100 ms e que as mensagens podem chegar no máximo 50 ms após a transmissão.
- Quando um processo pode ignorar uma mensagem contendo o carimbo de tempo T , se tiver registrado a última mensagem recebida desse processo como tendo o carimbo de tempo T' ?
 - Quando um receptor pode remover de sua tabela o carimbo de tempo de 175.000 (ms)? (Dica: use o valor do relógio local do receptor.)
 - Os relógios devem ser sincronizados internamente ou externamente?
- página 601*
- 14.4 Um cliente tenta sincronizar com um servidor de tempo. Ele grava os tempos de viagem de ida e volta e os carimbos de tempo retornados pelo servidor na tabela a seguir.
- Quais desses tempos ele deve usar para configurar seu relógio? Com que tempo ele deve ser configurado? Faça uma estimativa da precisão da configuração com relação ao relógio do servidor. Se for conhecido que o tempo entre o envio e o recebimento de uma mensagem no sistema envolvido é de pelo menos 8 ms, suas respostas mudarão?

<i>Viagem de ida e volta (ms)</i>	<i>Tempo (hr:min:seg)</i>
22	10:54:23.674
25	10:54:25.450
20	10:54:28.342

página 601

- 14.5 No sistema do Exercício 14.4, exige-se que se mantenha a sincronização do relógio de um servidor de arquivos dentro de ± 1 milissegundo. Discuta isso em relação ao algoritmo de Cristian.

página 601

- 14.6 Quais reconfigurações você esperaria que ocorressem na sub-rede de sincronização NTP?

página 604

- 14.7 Um servidor NTP B recebe uma mensagem do servidor A às 16:34:23.480, contendo o carimbo de tempo 16:34:13.430, e dá uma resposta. O servidor A recebe a mensagem às 16:34:15.725, contendo o carimbo de tempo de B, 16:34:25.7. Faça uma estimativa da compensação de B e A e a sua precisão.

página 605

- 14.8 Discuta os fatores a serem levados em conta ao se decidir com qual servidor NTP um cliente deve sincronizar seu relógio.

página 606

- 14.9 Discuta como é possível compensar a deriva de relógio entre pontos de sincronização observando a taxa de deriva com o passar do tempo. Discuta as limitações de seu método.

página 607

- 14.10 Considerando um encadeamento de zero ou mais mensagens conectando os eventos e e e' , e usando indução, mostre que $e \rightarrow e' \Rightarrow L(e) < L(e')$.

página 608

- 14.11 Mostre que $V_j[i] \leq V_i[i]$.

página 609

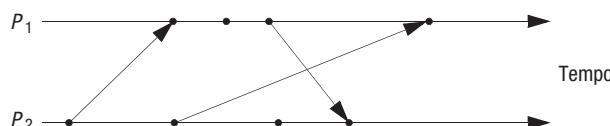
- 14.12 De maneira semelhante ao Exercício 14.10, mostre que $e \rightarrow e' \Rightarrow V(e) < V(e')$.

página 610

- 14.13 Usando o resultado do Exercício 14.11, mostre que, se os eventos e e e' são concorrentes, então nem $V(e) \leq V(e')$, nem $V(e') \leq V(e)$. Assim, mostre que, se $V(e) < V(e')$, então $e \rightarrow e'$.

página 610

- 14.14 Dois processos P e Q estão conectados em anel, usando dois canais, e constantemente fazem o rodízio de uma mensagem m . Em qualquer dado momento, existe apenas uma cópia de m no sistema. O estado de cada processo consiste no número de vezes que ele recebeu m , e P envia m primeiro. Em certo ponto, P tem a mensagem e seu estado é 101. Imediatamente após enviar m , P inicia o algoritmo do instantâneo. Explique o funcionamento do algoritmo neste caso, fornecendo o(s) estado(s) global(is) possível(is) relatado(s) por ele.

página 615

- 14.15 A figura anterior mostra eventos ocorrendo para cada um de dois processos, p_1 e p_2 . As setas entre os processos denotam transmissão de mensagens.

- Desenhe e rotule a treliça de estados consistentes (estado de p_1 , estado de p_2), começando com o estado inicial $(0,0)$.

página 622

- 14.16 Jones está executando um conjunto de processos p_1, p_2, \dots, p_N . Cada processo p_i contém uma variável v_i . Ela deseja determinar se todas as variáveis v_1, v_2, \dots, v_N já foram iguais no curso da execução.
- (i) Os processos de Jones são executados em um sistema síncrono. Ela usa um processo monitor para determinar se as variáveis já foram iguais. Quando os processos de aplicação devem se comunicar com o processo monitor e o que suas mensagens devem conter?
 - (ii) Explique a declaração *possivelmente* ($v_1 = v_2 = \dots = v_N$). Como Jones pode determinar se essa declaração é verdadeira para sua execução? página 623

15

Coordenação e Acordo

- 15.1 Introdução
- 15.2 Exclusão mútua distribuída
- 15.3 Eleições
- 15.4 Coordenação e acordo na comunicação em grupo
- 15.5 Consenso e problemas relacionados
- 15.6 Resumo

Neste capítulo, apresentaremos alguns tópicos e algoritmos relacionados ao problema de como os processos coordenam suas ações e entram em acordo sobre os valores compartilhados em sistemas distribuídos, a despeito das falhas. O capítulo começa com algoritmos para obter exclusão mútua dentre um conjunto de processos, de modo a coordenar seus acessos aos recursos compartilhados. Em seguida, ele examina como uma eleição pode ser implementada em um sistema distribuído. Isto é, descreve como um grupo de processos pode concordar sobre um novo coordenador para suas atividades, após o coordenador anterior ter falhado.

A segunda metade do capítulo examina os problemas relacionados com a comunicação em grupo, do consenso, do acordo bizantino e da consistência interativa. No contexto da comunicação em grupo, o problema é como entrar em acordo sobre questões como a ordem em que as mensagens devem ser enviadas. O consenso e os outros problemas são generalizados a partir deste: como um conjunto de processos pode concordar com algum valor, independentemente do domínio dos valores em questão? Encontramos um resultado fundamental na teoria dos sistemas distribuídos: que, sob certas condições – incluindo condições de falha surpreendentemente benignas – é impossível garantir que os processos cheguem a um consenso.

15.1 Introdução

Este capítulo apresenta um conjunto de algoritmos cujos objetivos variam, mas os quais compartilham uma meta fundamental em sistemas distribuídos: a de que um conjunto de processos coordene suas ações ou concorde com um ou mais valores. Por exemplo, no caso de um mecanismo complexo, como uma nave espacial, é fundamental que os computadores que a estão controlando concordem com condições como o fato de a missão da nave espacial estar prosseguindo ou ter sido cancelada. Além disso, os computadores precisam coordenar suas ações corretamente, com relação aos recursos compartilhados (os sensores e controladores da nave espacial). Os computadores devem ser capazes de fazer isso mesmo onde não haja nenhum relacionamento mestre-escravo fixo entre os componentes (o que tornaria a coordenação particularmente simples). O motivo para se evitar relacionamentos mestre-escravo fixos é que, frequentemente, exigimos que nossos sistemas continuem funcionando corretamente, mesmo que ocorram falhas; portanto, precisamos evitar pontos únicos de falha, como os mestres fixos.

Uma distinção importante para nós, como no Capítulo 14, será se o sistema distribuído sob estudo é assíncrono ou síncrono. Em um sistema assíncrono, não podemos fazer nenhuma suposição quanto à temporização. Em um sistema síncrono, devemos supor que existem limites para o atraso máximo na transmissão das mensagens, para o tempo que leva para executar cada etapa de um processo e para as taxas de deriva de relógio. As suposições síncronas nos permitem usar tempos limites para detectar falhas de processo.

Outro objetivo importante do capítulo, ao discutirmos os algoritmos, é considerar as falhas e como lidar com elas ao projetar algoritmos. A Seção 2.4.2 apresentou um modelo de falha, o qual usaremos neste capítulo. Suportar falhas é uma atividade sutil; portanto, começaremos considerando alguns algoritmos que não toleram falhas e passaremos para as falhas benignas, até considerarmos como fazemos para tolerar falhas arbitrárias. Encontramos um resultado fundamental na teoria dos sistemas distribuídos. Mesmo sob condições de falhas benignas, em um sistema assíncrono é impossível garantir que um conjunto de processos possa concordar com um valor compartilhado – por exemplo, que todos os processos de controle de uma nave espacial concordem com o prosseguimento da missão ou com o cancelamento da missão.

A Seção 15.2 examinará o problema da exclusão mútua distribuída. Trata-se da ampliação para os sistemas distribuídos do familiar problema de evitar condições de disputa nos núcleos e em aplicativos *multithreadeds*. Como grande parte do que ocorre nos sistemas distribuídos é o compartilhamento de recursos, esse é um problema importante a ser resolvido. Em seguida, a Seção 15.3 apresentará um problema relacionado, porém mais geral, de como “eleger” um processo a partir de um conjunto de processos, para desempenhar uma função especial. Por exemplo, no Capítulo 14, vimos como os processos sincronizavam seus relógios com um servidor de tempo designado. Se esse servidor falhar e vários servidores sobreviventes puderem desempenhar essa função, então, por causa da consistência, é necessário escolher apenas um servidor para assumir o comando.

A coordenação e o acordo relacionados à comunicação em grupo é o assunto da Seção 15.4. Conforme a Seção 4.4.1, a capacidade de fazer *multicast* de uma mensagem para um grupo é um paradigma de comunicação muito útil, com aplicações desde a localização de recursos até a coordenação das atualizações em dados replicados. A Seção 15.4 examinará a confiabilidade do *multicast* e a semântica da ordenação e fornecerá algoritmos para obter as variações. A entrega de mensagens baseada em *multicast* é basicamente um problema de acordo entre processos: os destinatários concordam com quais mensagens receberão e em que ordem as receberão. A Seção 15.5 discutirá o problema

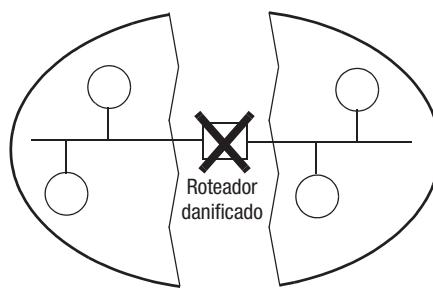


Figura 15.1 Um particionamento de rede.

do acordo de maneira mais geral, principalmente nas formas conhecidas como consenso e acordo bizantino.

O tratamento dado neste capítulo envolve a declaração das suposições e dos objetivos a serem atingidos e dar uma explicação informal sobre o motivo pelo qual os algoritmos apresentados estão corretos. Não há espaço suficiente para oferecer uma abordagem mais rigorosa. Para isso, recomendamos ao leitor um texto que forneça uma explicação completa sobre os algoritmos distribuídos, como Attiya e Welch [1998] e Lynch [1996].

Antes de apresentarmos os problemas e algoritmos, discutiremos as suposições de falha e a questão prática da detecção de falhas em sistemas distribuídos.

15.1.1 Suposições de falhas e detectores de falhas

Por simplicidade, este capítulo presume que cada par de processos está conectado por canais confiáveis. Isto é, embora os componentes de rede subjacentes possam falhar, os processos usam um protocolo de comunicação confiável que mascara essas falhas – por exemplo, retransmitindo mensagens perdidas ou corrompidas. Também por simplicidade, presumimos que nenhuma falha de processo implica em uma ameaça à capacidade dos outros processos de se comunicarem. Isso significa que nenhum dos processos depende de outro para encaminhar mensagens.

Note que um canal confiável acabará por enviar uma mensagem para o *buffer* de entrada do destinatário. Em um sistema síncrono, supomos que existe redundância de *hardware* onde é necessário, de modo que um canal confiável não apenas entregará cada mensagem, a despeito das falhas subjacentes, mas fará isso dentro de um limite de tempo especificado.

Em qualquer intervalo de tempo específico, a comunicação entre alguns processos pode acontecer, enquanto a comunicação entre outros é retardada. Por exemplo, a falha de um roteador entre duas redes pode significar que um conjunto de quatro processos é dividido em dois pares, de modo que a comunicação dentro do par é possível pelas suas respectivas redes; mas a comunicação entre pares não é possível enquanto o roteador estiver fora de operação. Isso é conhecido como *particionamento da rede* (Figura 15.1). Em uma rede ponto a ponto, como a Internet, topologias complexas e escolhas de roteamento independentes significam que a conectividade pode ser *assimétrica*: a comunicação é possível do processo *p* para o processo *q*, mas não *vice-versa*. A conectividade também pode ser *intransitiva*: a comunicação é possível de *p* para *q* e de *q* para *r*, mas *p* não pode se comunicar diretamente com *r*. Assim, nossa suposição de confiabilidade impõe que,

finalmente, qualquer enlace ou roteador defeituoso será reparado ou evitado. Contudo, nem todos os processos podem ser capazes de se comunicar ao mesmo tempo.

O capítulo presume, a não ser que declarado de outra forma, que os processos só falham por causa de defeitos – uma suposição suficientemente boa para muitos sistemas. Na Seção 15.5, consideraremos como fazer para tratar dos casos em que os processos têm falhas arbitrárias (bizantinas). Qualquer que seja o tipo de falha, um processo *correto* é aquele que não apresenta falha alguma em qualquer ponto na execução sob consideração. Note que a correção se aplica à execução inteira e não apenas a uma parte dela. Portanto, um processo que sofre uma falha por colapso era “não defeituoso” antes desse ponto e “não correto” depois dele.

Um dos problemas no projeto de algoritmos que podem superar falhas de processo é o de decidir quando um processo falhou. Um *detector de falha* [Chandra e Toueg 1996, Stelling et al. 1998] é um serviço que os processos consultam para saber se um processo em particular falhou. Frequentemente, ele é implementado por um objeto local para cada processo (no mesmo computador) que executa um algoritmo de detecção de falha, em conjunto com seus correlatos em outros processos. O objeto local de cada processo é chamado de *detector de falha local*. Em breve, descreveremos em linhas gerais como fazer para implementar detectores de falha, mas primeiro nos concentraremos em algumas de suas propriedades.

Um detector de falha não é necessariamente preciso. A maioria cai na categoria dos *detectores de falha não confiáveis*. Um detector de falha não confiável pode produzir um de dois valores, dada a identidade de um processo: *não suspeito* ou *suspeito*. Esses dois resultados são sugestões, que podem, ou não, refletir precisamente se o processo realmente falhou. Um resultado *não suspeito* significa que recentemente o detector recebeu uma evidência sugerindo que o processo não falhou; por exemplo, uma mensagem foi recebida recentemente dele. Entretanto, é claro que o processo pode ter falhado desde então. Um resultado *suspeito* significa que o detector de falha tem alguma indicação de que o processo pode ter falhado. Por exemplo, pode ser que nenhuma mensagem tenha sido recebida do processo por mais do que um período máximo nominal de silêncio (mesmo em um sistema assíncrono, limites superiores práticos podem ser usados como sugestões). A suspeita pode ser depositada em lugar errado: por exemplo, o processo poderia estar funcionando corretamente, mas no outro lado de uma partição de rede; ou poderia estar executando mais lentamente do que o esperado.

Um *detector de falha confiável* é aquele que é sempre preciso na detecção da falha de um processo. Ele responde às consultas dos processos indicando *Não suspeito* – o que, como antes, pode ser apenas uma sugestão – ou *Falho*. Um resultado *falso* significa que o detector determinou que o processo está em colapso. Lembre-se de que um processo em colapso permanece desse jeito, pois, por definição, um processo nunca dará outro passo, uma vez que tenha falhado.

É importante perceber que, embora estejamos falando de um detector de falha atuando para um conjunto de processos, a resposta dada por ele para um processo é apenas tão boa quanto à informação disponível nesse processo. Às vezes, um detector de falha pode dar diferentes respostas para diferentes processos, pois as condições de comunicação variam de um processo para outro.

Podemos implementar um detector de falha não confiável usando o algoritmo a seguir. Cada processo p envia uma mensagem “ p está aqui” para cada outro processo e faz isso a cada T segundos. O detector de falha usa uma estimativa do tempo máximo de transmissão da mensagem, de D segundos. Se o detector de falha local no processo q não receber uma mensagem “ p está aqui” dentro de $T + D$ segundos da última, então relatará para q que p é *suspeito*. Entretanto, se subsequentemente ele receber uma mensagem “ p está aqui”, então relatará para q que p está OK.

Em um sistema distribuído real, existem limites práticos para os tempos de transmissão da mensagem. Até os sistemas de *e-mail* desistem após alguns dias, pois é provável que os enlaces de comunicação e roteadores tenham sido reparados nesse tempo. Se escolhermos valores pequenos para T e D (de modo que eles totalizem, digamos, 0,1 segundo), então o detector de falha provavelmente suspeitará de processos não falhos muitas vezes e muita largura de banda será ocupada com mensagens “*p está aqui*”. Se escolhermos um valor grande para o tempo limite total (digamos, uma semana), os processos falhos serão frequentemente relatados como *Não suspeitos*.

Uma solução prática para esse problema é usar valores de tempo limite que reflitam as condições de atraso observadas na rede. Se um detector de falha local recebesse uma mensagem “*p está aqui*” em 20 segundos, em vez do máximo esperado de 10 segundos, ele poderia reconfigurar seu valor de tempo limite para p de acordo com isso. O detector de falha permanece não confiável e suas respostas às consultas ainda são apenas sugestões, mas a probabilidade de sua precisão aumenta.

Em um sistema síncrono, nosso detector de falha pode se tornar confiável. Podemos escolher D de modo que não seja uma estimativa, mas um limite absoluto para os tempos de transmissão da mensagem; a ausência de uma mensagem “*p está aqui*” dentro de $T + D$ segundos autoriza o detector de falha local a concluir que p falhou.

O leitor pode estar se perguntando se os detectores de falha têm algum uso prático. Os detectores de falha não confiáveis podem suspeitar de um processo que não falhou (eles podem ser *imprecisos*); e eles podem não suspeitar de um processo que, na realidade, falhou (eles podem ser *incompletos*). Por outro lado, os detectores de falha confiáveis exigem que o sistema seja síncrono (e alguns sistemas práticos o são).

Apresentamos os detectores de falha porque eles nos ajudam a pensar sobre a natureza das falhas em um sistema distribuído. E qualquer sistema prático projetado para suportar falhas deve detectá-las – ainda que imperfeitamente. Contudo, verifica-se que mesmo os detectores de falha não confiáveis com certas propriedades bem definidas podem nos ajudar a providenciar soluções práticas para o problema da coordenação de processos na presença de falhas. Voltaremos a esse assunto na Seção 15.5.

15.2 Exclusão mútua distribuída

Os processos distribuídos frequentemente precisam coordenar suas atividades. Se um conjunto de processos compartilha um recurso, ou uma coleção de recursos, então, frequentemente, a exclusão mútua é exigida para evitar interferência e garantir a consistência ao acessar esses recursos. Esse é o problema da *seção crítica*, familiar no domínio dos sistemas operacionais. Em um sistema distribuído, entretanto, nem as variáveis compartilhadas nem os recursos fornecidos por um único núcleo local podem ser usados para resolvê-lo, em geral. Precisamos de uma solução para a *exclusão mútua distribuída*: uma que seja baseada unicamente na passagem de mensagens.

Em alguns casos, os recursos compartilhados são gerenciados por servidores que também fornecem mecanismos de exclusão mútua. O Capítulo 16 descreve como alguns servidores sincronizam os acessos de cliente aos recursos. No entanto, em alguns casos práticos, é exigido um mecanismo de exclusão mútua separado.

Considere usuários que atualizam um arquivo de texto. Uma maneira simples de garantir que suas atualizações sejam consistentes é permitir que apenas um usuário por vez o acesse, exigindo que o editor bloquee o arquivo antes que as atualizações possam ser feitas. Os servidores de arquivo NFS, descritos no Capítulo 12, são projetados para

serem sem estado e, portanto, não suportam travamento (*lock*) de arquivo. Por isso, os sistemas UNIX fornecem um serviço de travamento de arquivo separado, implementado pelo daemon *lockd*, para tratar das requisições de travamento dos clientes.

Um exemplo particularmente interessante é aquele em que não há nenhum servidor e um conjunto de processos pares precisa coordenar seus acessos aos recursos compartilhados entre eles mesmos. Isso ocorre rotineiramente em redes como as Ethernet e em redes IEEE 802.11 sem fio no modo *ad hoc*, em que as interfaces de rede cooperam como pares para que apenas um nó por vez transmita no meio compartilhado. Considere também um sistema monitorando o número de vagas em um estacionamento, com um processo em cada entrada e saída, controlando o número de veículos que entram e que saem. Cada processo mantém uma contagem do número total de veículos dentro do estacionamento e mostra se ele está lotado ou não. Os processos devem atualizar consistentemente a contagem compartilhada do número de veículos. Existem várias maneiras de conseguir isso, mas seria conveniente que esses processos pudessem obter exclusão mútua unicamente se comunicando entre si, eliminando a necessidade de um servidor separado.

É útil ter um mecanismo genérico de exclusão mútua distribuída à nossa disposição, que seja independente do esquema de gerenciamento de recursos específico em questão. Examinaremos agora alguns algoritmos para isso.

15.2.1 Algoritmos de exclusão mútua

Consideramos um sistema de N processos p_i , $i = 1, 2, \dots, N$ que não compartilham variáveis. Os processos acessam recursos comuns, mas fazem isso em uma seção crítica. Por simplicidade, supomos que existe apenas uma seção crítica. É fácil estender os algoritmos que apresentaremos para mais de uma seção crítica.

Supomos que o sistema seja assíncrono, que os processos não falham e que o envio de mensagens é confiável, de modo que toda mensagem enviada será entregue intacta, exatamente uma vez.

O protocolo em nível de aplicativo para executar uma seção crítica é o seguinte:

```
enter()           // entra na seção crítica – bloqueia, se necessário  
resourceAccesses() // acessa recursos compartilhados na seção crítica  
exit()            // sai da seção crítica – outros processos podem entrar agora
```

Nossos requisitos básicos de exclusão mútua são os seguintes:

EM1: (segurança) No máximo um processo por vez pode ser executado na seção crítica (SC).

EM2: (subsistência) As requisições para entrar e sair da seção crítica têm sucesso.

A condição EM2 implica em independência de impasse e inanição. Um impasse envolveria dois ou mais processos travando indefinidamente, enquanto tentam entrar ou sair da seção crítica, devido à sua interdependência mútua. No entanto, mesmo sem um impasse, um algoritmo deficiente pode levar à *inanição*: o adiamento indefinido da entrada de um processo que a solicitou.

A ausência de inanição é uma condição de *imparcialidade*. Outro problema de imparcialidade é a ordem na qual os processos entram na seção crítica. Não é possível ordenar a entrada na seção crítica pelos tempos que os processos a solicitaram, devido à ausência de relógios globais, mas um requisito de imparcialidade útil que às vezes é esta-

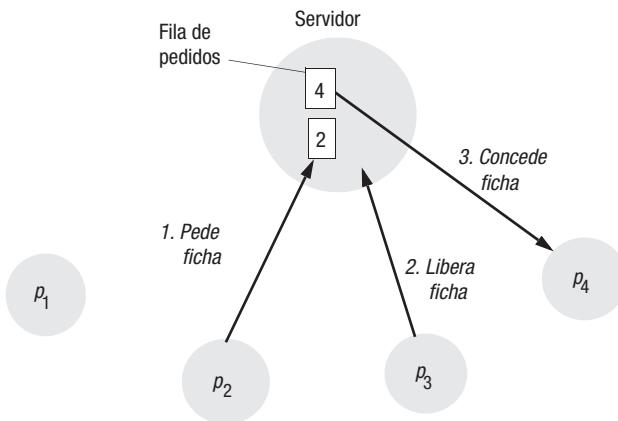


Figura 15.2 Servidor gerenciando uma ficha de exclusão mútua para um conjunto de processos.

beleido faz uso da ordem acontece antes (Seção 14.4) entre as mensagens que solicitam a entrada na seção crítica:

EM3: (ordenação →) Se uma requisição para entrar na SC aconteceu antes de outra, então a entrada na SC é garantida nessa ordem.

Se uma solução garante a entrada na seção crítica na ordem acontece antes, e se todas as requisições são relacionadas por essa ordem, então não é possível um processo entrar na seção crítica mais de uma vez, enquanto outro espera para entrar. Essa ordem também permite que os processos coordenem seus acessos à seção crítica. Um processo *multithreaded* pode continuar com outro processamento, enquanto uma de suas *threads* espera para ter a entrada garantida em uma seção crítica. Durante esse tempo, ele poderia enviar uma mensagem para outro processo, o qual, consequentemente, também tentaria entrar na seção crítica. EM3 especifica que o primeiro processo tem o acesso garantido antes do segundo.

Avaliamos o desempenho dos algoritmos de exclusão mútua de acordo com os seguintes critérios:

- a *largura de banda* consumida, que é proporcional ao número de mensagens enviadas em cada operação de *entrada* e *saída*;
- o *atraso do cliente* acarretado por um processo em cada operação de *entrada* e *saída*;
- o efeito do algoritmo sobre a taxa de rendimento (*throughput*) do sistema. Trata-se da velocidade com que o conjunto de processos como um todo pode acessar a seção crítica, dado que alguma comunicação é necessária entre processos sucessivos. Medimos o efeito usando o *atraso da sincronização* entre um processo saindo da seção crítica e o próximo processo entrando nela; a taxa de rendimento é maior quando o atraso da sincronização é mais curto.

Em nossas descrições, não levaremos em conta a implementação dos acessos aos recursos. Entretanto, vamos supor que os processos clientes tenham bom comportamento e gastem um tempo finito acessando recursos dentro de suas seções críticas.

O algoritmo do servidor central • O modo mais simples de obter exclusão mútua é empregar um servidor que conceda permissão para entrar na seção crítica. A Figura 15.2

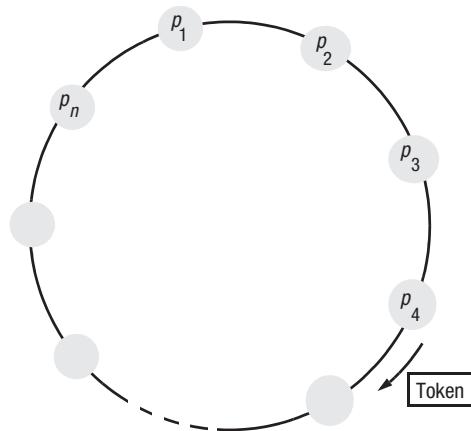


Figura 15.3 Um anel de processos transferindo um *token* de exclusão mútua.

mostra o uso desse servidor. Para entrar em uma seção crítica, um processo envia uma mensagem de requisição para o servidor e espera uma resposta. Conceitualmente, a resposta constitui uma ficha (*token*) significando permissão para entrar na seção crítica. Se nenhum outro processo tiver a ficha no momento da requisição, então o servidor responderá imediatamente, concedendo a ficha. Se a ficha estiver de posse de outro processo, então o servidor não responderá, mas enfileirará a requisição. Na saída da seção crítica, uma mensagem é enviada para o servidor, devolvendo a ficha a ele.

Se a fila de processos em espera não estiver vazia, o servidor escolherá a entrada mais antiga, a removerá e responderá para o processo correspondente. Então, o processo escolhido terá a posse da ficha. Na figura, mostramos uma situação em que a requisição de p_2 foi anexado na fila, a qual já continha a requisição de p_4 . Quando p_3 sai da seção crítica, o servidor remove a entrada de p_4 e concede a permissão para entrar na seção crítica enviando uma resposta a ele. Correntemente, o processo p_1 não solicita entrada na seção crítica.

Dada nossa suposição de que não ocorrem falhas, é fácil ver que as condições de segurança e subsistência são satisfeitas por esse algoritmo. Entretanto, o leitor deve verificar que o algoritmo não satisfaz a propriedade EM3.

Avaliaremos agora o desempenho desse algoritmo. A entrada na seção crítica – mesmo quando nenhum processo a ocupa correntemente – exige duas mensagens (uma *requisição*, seguida de uma *concessão*) e atrasa o processo de solicitação pelo tempo dessa viagem de ida e volta. A saída da seção crítica exige uma única mensagem de *liberação*. Supondo a passagem de mensagens assíncrona, isso não atrasa o processo de saída.

O servidor pode se tornar um gargalo de desempenho para o sistema como um todo. O atraso de sincronização é o tempo exigido para uma viagem de ida e volta: uma mensagem de *liberação* para o servidor, seguida de uma mensagem de *concessão* para o próximo processo a entrar na seção crítica.

Um algoritmo baseado em anel • Uma das maneiras mais simples de constituir a exclusão mútua entre os N processos, sem exigir um processo adicional, é organizá-los em um anel lógico. Isso exige apenas que cada processo p_i tenha um canal de comunicação com o processo seguinte no anel, $p_{(i+1)mod N}$. A ideia é que a exclusão seja concedida pela obtenção de uma ficha, na forma de uma mensagem passada de processo para processo, em uma única

Na inicialização

```
state := RELEASED;
```

Para entrar na seção

```
state := WANTED;
Envia a requisição por multicast para todos os processos;
T := carimbo de tempo da requisição;
Espera até (número de respostas recebidas = (N - 1));
state := HELD;
```

No recebimento de uma requisição $<T_i, p_j>$ em p_j ($i \neq j$)

```
if (state = HELD or (state = WANTED and (T, p_j) < (T_i, p_j)))
then
    enfileira requisição de  $p_j$  sem responder;
else
    responde imediatamente para  $p_j$ ;
end if
```

Para sair da seção crítica

```
state := RELEASED;
responde a todas as requisições enfileiradas;
```

Figura 15.4 Algoritmo de Ricart e Agrawala.

direção – digamos, no sentido horário – em torno do anel. A topologia em anel pode não estar relacionada com as interconexões físicas entre os computadores subjacentes.

Se, se ao receber a ficha, um processo não deseja entrar na seção crítica, encaminha imediatamente a ficha para seu vizinho. Um processo que solicite a ficha espera até recebê-lo, mas o mantém. Para sair da seção crítica, o processo envia a ficha para seu vizinho.

A organização dos processos aparece na Figura 15.3. É fácil verificar que as condições EM1 e EM2 são satisfeitas por esse algoritmo, mas que a ficha não é necessariamente obtida na ordem acontece antes. (Lembre-se de que os processos podem trocar mensagens independentemente da rotação da ficha.)

Esse algoritmo consome largura de banda de rede continuamente (exceto quando um processo está dentro da seção crítica): os processos enviam mensagens em torno do anel, mesmo quando nenhum processo solicita entrada na seção crítica. O atraso experimentado por um processo que esteja solicitando a entrada na seção crítica está entre 0 (quando ele acabou de receber a ficha) e N mensagens (quando ele acabou de passar a ficha). Sair da seção crítica exige apenas uma mensagem. O atraso de sincronização entre a saída de um processo da seção crítica e a entrada do processo seguinte está entre 1 e N transmissões de mensagens.

Um algoritmo usando multicast e relógios lógicos • Ricart e Agrawala [1981] desenvolveram um algoritmo para implementar exclusão mútua entre N processos pares, baseado em *multicast*. A ideia básica é que os processos que solicitam a entrada em uma seção crítica difundem seletivamente (*multicast*) uma mensagem de requisição e só podem entrar nela quando todos os outros processos tiverem respondido a essa mensagem. As condições sob as quais um processo responde a uma requisição são projetadas de forma a garantir que as condições EM1 a EM3 sejam satisfeitas.

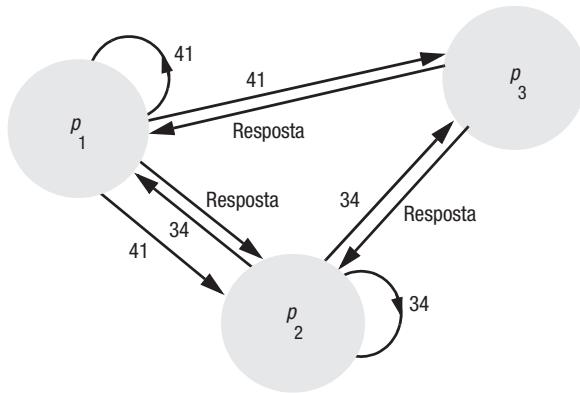


Figura 15.5 Sincronização por *multicast*.

Os processos p_1, p_2, \dots, p_N apresentam identificadores numéricos distintos. Presume-se que eles possuam canais de comunicação de um para o outro e que cada processo p_i mantenha um relógio de Lamport, atualizado de acordo com as regras RL1 e RL2 da Seção 14.4. As mensagens que solicitam entrada são da forma $\langle T, p_i \rangle$, onde T é o carimbo de tempo do remetente e p_i é o identificador do remetente.

Cada processo registra seu estado de estar fora da seção crítica (*RELEASED*), querendo entrar (*WANTED*) ou estar na seção crítica (*HELD*), em uma variável *state*. O protocolo aparece na Figura 15.4.

Se um processo solicita entrada, e o estado dos outros processos é *RELEASED*, então todos responderão imediatamente a requisição e o solicitante obterá a entrada. Se algum processo estiver no estado *HELD*, então esse processo não responderá as requisições até que tenha terminado com a seção crítica; portanto, o solicitante não poderá entrar nesse meio tempo. Se dois ou mais processos solicitam a entrada ao mesmo tempo, a requisição do processo que apresentar o carimbo de tempo mais baixo será o primeiro a coletar $N - 1$ respostas, garantindo a próxima entrada. Se as requisições apresentarem carimbo de tempo de Lamport iguais, serão ordenados de acordo com os identificadores correspondentes dos processos. Note que, quando um processo solicita a entrada, ele retarda o processamento dos pedidos de outros processos até que sua própria requisição tenha sido enviada e ele tenha gravado o carimbo de tempo T da requisição. Isso é assim para que os processos tomem decisões consistentes ao processar requisições.

Esse algoritmo tem a propriedade de segurança EM1. Se fosse possível dois processos, p_i e p_j ($i \neq j$), entrarem na seção crítica ao mesmo tempo, eles teriam que ter respondido um ao outro. No entanto, como os pares $\langle T_p, p_i \rangle$ são totalmente ordenados, isso é impossível. Deixamos para o leitor verificar que o algoritmo também atende aos requisitos EM2 e EM3.

Para ilustrar o algoritmo, considere uma situação envolvendo três processos, p_1, p_2 e p_3 , apresentada na Figura 15.5. Vamos supor que p_3 não esteja interessado em entrar na seção crítica e que p_1 e p_2 solicitam a entrada concorrentemente. O carimbo de tempo da requisição de p_1 é 41 e a de p_2 é 34. Quando p_3 recebe as requisições, responde imediatamente. Quando p_2 recebe a requisição de p_1 , verifica que sua própria requisição tem o carimbo de tempo mais baixo e, portanto, não responde, detendo p_1 . Entretanto, p_1 verifica que a requisição de p_2 tem um carimbo de tempo mais baixo do que à da sua própria

requisição e, portanto, responde imediatamente. Ao receber essa segunda resposta, p_2 pode entrar na seção crítica. Quando p_2 sair da seção crítica, responderá à requisição de p_1 e, portanto, garantirá sua entrada.

A obtenção da entrada exige $2(N - 1)$ mensagens nesse algoritmo: $N - 1$ para difundir a requisição por *multicast*, seguido de $N - 1$ respostas. Ou então, se houver suporte de *hardware* para *multicast*, apenas uma mensagem será exigida para a requisição; assim, o total será de N mensagens. Portanto, esse é um algoritmo mais dispendioso, em termos de consumo de largura de banda, do que os algoritmos que acabamos de descrever. Entretanto, o atraso do cliente na requisição de entrada é, novamente, o tempo de uma viagem de ida e volta (ignorando qualquer atraso acarretado no envio *multicast* da mensagem de requisição).

A vantagem desse algoritmo é que seu atraso de sincronização é apenas o tempo da transmissão de uma mensagem. Os dois algoritmos anteriores acarretavam um atraso de sincronização de uma viagem de ida e volta.

O desempenho do algoritmo pode ser melhorado. Primeiramente, note que o processo que entrou por último na seção crítica, e que não recebeu nenhuma outra requisição de entrada de outros processos, ainda passa pelo protocolo, conforme descrito, mesmo que pudesse simplesmente decidir realocá-lo para si mesmo de forma local. Segundo, Ricart e Agrawala refinaram esse protocolo de modo que ele exigisse N mensagens para obter a entrada no pior (e comum) caso, sem suporte de *hardware* para *multicast*. Isso está descrito em Raynal [1988].

Algoritmo de votação de Maekawa • Maekawa [1985] observou que, para um processo entrar em uma seção crítica, não é necessário que todos os seus pares concedam o acesso. Os processos só precisam obter permissão de *subconjuntos* de seus pares para entrar, desde que os subconjuntos usados por quaisquer dois processos se sobreponham. Podemos considerar os processos como votando um no outro para entrar na seção crítica. Um processo “candidato” deve reunir votos suficientes para entrar. Os processos na intersecção de dois conjuntos de votantes garantem a propriedade de segurança EM1, de que no máximo um processo pode entrar na seção crítica, depositando seus votos para apenas um candidato.

Maekawa associou um conjunto de votação V_i a cada processo p_i ($i = 1, 2, \dots, N$), onde $V_i \subseteq \{p_1, p_2, \dots, p_N\}$. Os conjuntos V_i são escolhidos de modo que, para todo $i, j = 1, 2, \dots, N$:

- $p_i \in V_i$;
- $V_i \cap V_j \neq \emptyset$: existe pelo menos um membro em comum de quaisquer dois conjuntos votantes;
- $|V_i| = K$: para ser imparcial, cada processo tem um conjunto votante de mesmo tamanho;
- Cada processo p_j está contido em M conjuntos votantes V_i .

Maekawa mostrou que a solução ótima, que minimiza K e permite que os processos obtenham exclusão mútua, tem $K \approx \sqrt{N}$ e $M = K$ (de modo que cada processo está em tantos conjuntos de votação quantos são os elementos em cada um desses conjuntos). Não é simples calcular os conjuntos ótimos R_i . Como uma aproximação, uma maneira simples de derivar conjuntos R_i tais que $|R_i| \approx \sqrt{2}$, é colocar os processos em uma matriz de \sqrt{N} por \sqrt{N} e deixar V_i ser a união da linha e coluna que contêm p_i .

O algoritmo de Maekawa aparece na Figura 15.6. Para obter a entrada na seção crítica, um processo p_i envia mensagens de *requisição* para todos os K membros de V_i

Na inicialização
state := RELEASED;
voted:= FALSE;

Para p_j entrar na seção crítica
state := WANTED;
Envia a requisição por multicast para todos os processos em V_j ;
Espera até (número de respostas recebidas = K);
state := HELD;

No recebimento de uma requisição de p_j em p_i
if (state = HELD ou voted = TRUE)
then
 enfileira a requisição de p_j sem responder;
else
 envia resposta para p_j ;
 voted := TRUE;
end if

Para p_j sair da seção crítica
state := RELEASED;
Envia a liberação via multicast para todos os processos em V_j ;

No recebimento de uma liberação de p_j em p_i
if (fila de requisições não estiver vazia)
then
 remove cabeça da fila – digamos, p_k ;
 envia resposta para p_k ;
 voted := TRUE;
else
 voted := FALSE;
end if

Figura 15.6 Algoritmo de Maekawa.

(incluindo ele mesmo). p_i não pode entrar na seção crítica até que tenha recebido todas as K mensagens de *resposta*. Quando um processo p_j em V_i recebe a mensagem de *requisição de p_i* , ele envia uma mensagem de *resposta* imediatamente, a não ser que seu estado seja *HELD* ou que já tenha respondido (“votado”) desde a última vez que recebeu uma mensagem de *liberação*. Caso contrário, ele enfileira a mensagem de requisição (na ordem de sua chegada), mas não responde ainda. Quando um processo recebe uma mensagem de *liberação*, ele remove o nodo cabeça de sua fila de requisições pendentes (se a fila não estiver vazia) e envia uma mensagem de *resposta* (um “voto”) em retorno a ela. Para sair da seção crítica, p_i envia mensagens de *liberação* para todos os K membros de V_i (incluindo ele mesmo).

Esse algoritmo obtém a propriedade da segurança EM1. Se fosse possível que dois processos p_i e p_j entrassem na seção crítica ao mesmo tempo, então os processos em $V_i \cap V_j \neq \emptyset$ teriam de ter votado em ambos. Contudo, o algoritmo permite que um processo deposite no máximo um voto entre sucessivos recebimentos de uma mensagem de *liberação* – portanto, essa situação é impossível.

Infelizmente, o algoritmo é propenso a impasses. Considere três processos p_1 , p_2 e p_3 com $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$ e $V_3 = \{p_3, p_1\}$. Se os três processos solicitam a entrada na seção crítica, então é possível que p_1 responda para si mesmo e detenha p_2 , que p_2 responda para si mesmo e detenha p_3 e que p_3 responda para si mesmo e detenha p_1 . Cada processo recebeu uma de duas respostas e nenhum pode prosseguir.

O algoritmo pode ser adaptado [Sanders 1987] de modo que se torne livre de impasses. No protocolo adaptado, os processos enfileiram as requisições pendentes na ordem acontece antes, de modo que o requisito EM3 também é satisfeito.

A utilização de largura de banda do algoritmo é de $2\sqrt{N}$ mensagens por entrada na seção crítica e de \sqrt{N} mensagens por saída (supondo que não exista nenhum recurso de hardware para *multicast*). O total de $3\sqrt{N}$ é superior às $2(N - 1)$ mensagens exigidas pelo algoritmo de Ricart e Agrawala, se $N > 4$. O atraso de cliente é o mesmo do algoritmo de Ricart e Agrawala, mas o atraso de sincronização é pior: um tempo de viagem de ida e volta, em vez de um único tempo de transmissão de mensagem.

Tolerância a falhas • Os principais pontos a considerar na avaliação dos algoritmos anteriores, com relação à tolerância a falhas, são:

- O que acontece quando mensagens são perdidas?
- O que acontece quando um processo falha?

Nenhum dos algoritmos que descrevemos toleraria a perda de mensagens, caso os canais fossem não confiáveis. O algoritmo baseado em anel não pode tolerar uma falha por colapso de um único processo. Como se vê, o algoritmo de Maekawa pode tolerar algumas falhas de processo por colapso: se um processo falho não estiver em um conjunto votante que seja exigido, então sua falha não afetará os outros processos. O algoritmo do servidor central pode tolerar a falha por colapso de um processo cliente que não contenha, nem tenha solicitado, a ficha. O algoritmo de Ricart e Agrawala, conforme o descrevemos, pode ser adaptado para tolerar a falha por colapso de tal processo, fazendo-o conceder todas as requisições implicitamente.

Convidamos o leitor a considerar como adaptar os algoritmos para tolerar falhas, supondo que um detector de falha confiável esteja disponível. Mesmo com um detector de falha confiável, é necessário cuidado para permitir a existência de falhas em qualquer ponto (inclusive durante um procedimento de recuperação) e para reconstruir o estado dos processos após uma falha ter sido detectada. Por exemplo, no algoritmo do servidor central, se o servidor falhar, deverá ser estabelecido quem detém a ficha, se o servidor ou um dos processos clientes.

Examinaremos o problema geral de como os processos devem coordenar suas ações na presença de falhas na Seção 15.5.

15.3 Eleições

Um algoritmo para escolher um único processo para desempenhar uma função em particular é chamado de *algoritmo de eleição*. Por exemplo, em uma variante de nosso algoritmo do servidor central para exclusão mútua, o servidor é escolhido dentre os processos p_i , ($i = 1, 2, \dots, N$) que precisam usar a seção crítica. Um algoritmo de eleição é necessário para escolher qual dos processos desempenhará a função de servidor. É fundamental que todos os processos concordem com a escolha. Depois disso, se o processo que desempenha a função de servidor quiser se retirar, outra eleição será exigida para escolher um substituto.

Dizemos que um processo *convoca a eleição* se ele faz uma ação que inicia uma execução em particular do algoritmo de eleição. Um processo individual não convoca mais do que uma eleição por vez, mas, em princípio, os N processos poderiam convocar N eleições concorrentes. A qualquer momento, um processo p_i é *participante* – significando que ele está envolvido em alguma execução do algoritmo de eleição – ou *não participante* – significando que correntemente ele não está envolvido em nenhuma eleição.

Um requisito importante é que a escolha do processo eleito seja única, mesmo que vários processos convoquem eleições concorrentemente. Por exemplo, dois processos poderiam decidir, independentemente, que um processo coordenador falhou, e ambos convocam eleições.

Sem perda de generalidade, exigimos que o processo eleito seja escolhido como aquele com o maior identificador. O identificador pode ser qualquer valor útil, desde que os identificadores sejam exclusivos e totalmente ordenados. Por exemplo, poderíamos eleger o processo com a menor carga computacional, fazendo cada processo usar $<1/carga, i>$ como identificador, onde $carga > 0$, e o índice de processo i seria usado para ordenar identificadores com a mesma carga.

Cada processo p_i ($i = 1, 2, \dots, N$) tem uma variável $eleito_i$, que conterá o identificador do processo eleito. Quando o processo se torna participante de uma eleição pela primeira vez, ele configura essa variável com o valor especial ‘ \perp ’, para denotar que ela ainda não está definida.

Nossos requisitos são que, durante qualquer execução em particular do algoritmo:

- | | |
|--------------------|---|
| E1: (segurança) | Um processo participante p_i tem $eleito_i = \perp$ ou $eleito_i = P$, onde P é escolhido como o processo não defeituoso com o maior identificador no final da execução. |
| E2: (subsistência) | Todos os processos p_i participam e configuram $eleito_i \neq \perp$ ou falham. |

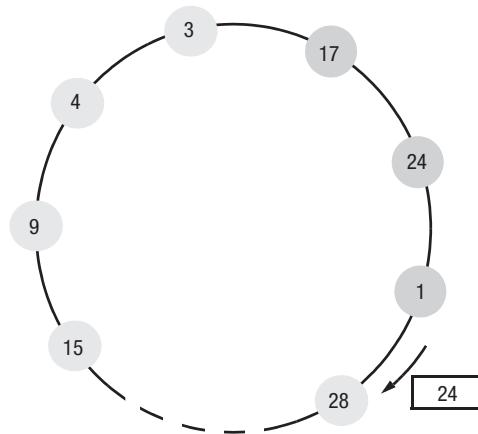
Note que podem existir processos p_i que ainda não sejam participantes, os quais registram em $eleito_i$ o identificador do processo eleito anterior.

Medimos o desempenho de um algoritmo de eleição por sua utilização de largura de banda de rede total (que é proporcional ao número total de mensagens enviadas) e pelo *tempo do ciclo de rotação* do algoritmo: o número de tempos de transmissão de mensagem dispostos em série, entre o início e o término de uma execução.

Um algoritmo de eleição baseado em anel • Fornecemos o algoritmo de Chang e Roberts [1979], que é conveniente para um conjunto de processos organizados em um anel lógico. Cada processo p_i tem um canal de comunicação para o processo seguinte no anel, $p_{(i+1)mod N}$, e todas as mensagens são enviadas no sentido horário em torno do anel. Supomos que não ocorrem falhas e que o sistema é assíncrono. O objetivo desse algoritmo é eleger um único processo, chamado de *coordenador*, que é aquele com o maior identificador.

Inicialmente, cada processo é marcado como *não participante* de uma eleição. Qualquer processo pode iniciar uma eleição. Ele prossegue marcando a si mesmo como *participante*, colocando seu identificador em uma mensagem de *eleição* e enviando-a para seu vizinho no sentido horário.

Quando um processo recebe uma mensagem de *eleição*, ele compara o identificador presente na mensagem com o seu próprio. Se o identificador que chegou é maior, ele encaminha a mensagem para seu vizinho. Se o identificador que chegou é menor e o receptor não é *participante*, ele substitui por seu próprio identificador na mensagem e a encaminha; mas não encaminha a mensagem se já for *participante*. Em qualquer caso,



Nota: A eleição foi iniciada pelo processo 17. O identificador de processo mais alto encontrado até aqui é 24. Os processos participantes aparecem com fundo cinza mais escuro.

Figura 15.7 Uma eleição baseada em anel (em andamento).

no encaminhamento de uma mensagem de *eleição*, o processo marca a si mesmo como *participante*.

Entretanto, se o identificador recebido for o do próprio receptor, então o identificador desse processo deve ser o maior e se tornará o coordenador. Mais uma vez, o coordenador marca a si mesmo como *não participante* e envia uma mensagem *eleito* para seu vizinho, anunciando sua eleição e incluindo sua identidade.

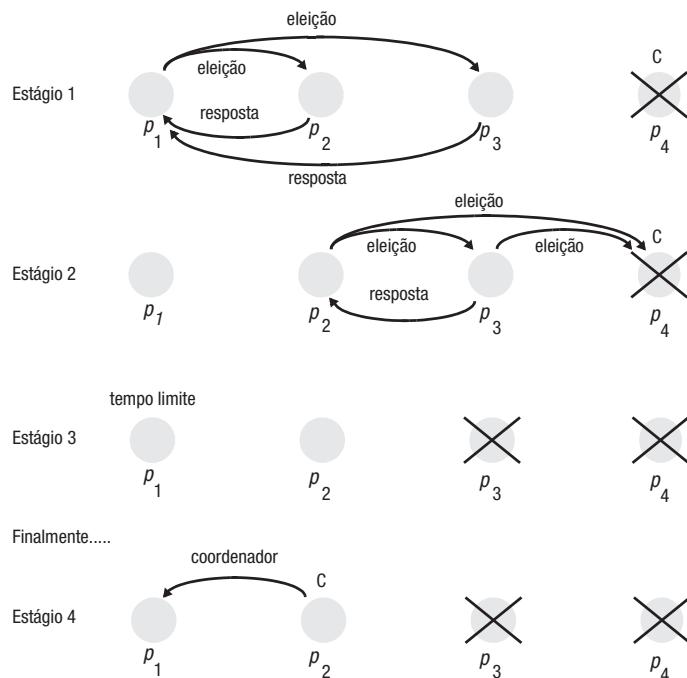
Quando um processo p_i recebe uma mensagem *eleito*, ele marca a si mesmo como *não participante*, configura sua variável $eleito_i$ com o identificador presente na mensagem e, a não ser que seja o novo coordenador, encaminha a mensagem para seu vizinho.

É fácil ver que a condição E1 é satisfeita. Todos os identificadores são comparados, pois um processo deve receber seu próprio identificador de volta, antes de enviar uma mensagem *eleito*. Para quaisquer dois processos, aquele com o identificador maior não passará o identificador do outro. Portanto, é impossível que ambos recebam seus próprios identificadores de volta.

A condição E2 resulta imediatamente da garantia das passagens de mensagem no anel (não existem falhas). Observe como os estados *não participante* e *participante* são usados para extinguir as mensagens que surgem quando outro processo inicia uma eleição ao mesmo tempo, assim que possível e sempre antes do resultado da eleição ter sido anunciado.

Se apenas um processo inicia uma eleição, então o caso de pior desempenho se dá quando seu vizinho no sentido anti-horário tem o identificador mais alto. Então, um total de $N - 1$ mensagens é exigido para chegar a esse vizinho, o qual não anunciará sua eleição até que seu identificador tenha completado outra volta, exigindo mais N mensagens. A mensagem *eleito* é, então, enviada N vezes, totalizando $3N - 1$ mensagens. O tempo do ciclo de rotação também é de $3N - 1$, pois essas mensagens são enviadas sequencialmente.

Um exemplo de eleição baseada em anel em andamento aparece na Figura 15.7. A mensagem de *eleição* contém correntemente 24, mas o processo 28 a substituirá pelo seu identificador, quando a mensagem chegar a ele.



A eleição do coordenador p_2 , após a falha de p_3 e, depois, de p_4 .

Figura 15.8 O algoritmo valenção.

Embora o algoritmo baseado em anel seja útil para se entender as propriedades dos algoritmos de eleição em geral, o fato de ele não tolerar falhas o torna limitado quanto ao seu valor prático. Entretanto, com um detector de falha confiável é possível, em princípio, reconstituir o anel quando um processo falha.

O algoritmo valenção (bully) • O algoritmo valenção [Garcia-Molina 1982] permite que os processos falhem durante uma eleição, embora presuma que a distribuição de mensagens entre os processos seja confiável. Ao contrário do algoritmo baseado em anel, este algoritmo presume que o sistema é síncrono: ele usa tempos limites para detectar uma falha de processo. Outra diferença é que o algoritmo baseado em anel presumia que os processos tinham *a priori* conhecimento mínimo uns dos outros: cada um sabia apenas como se comunicar com seu vizinho, e nenhum conhecia os identificadores dos outros processos. Por outro lado, o algoritmo valenção presume que cada processo sabe quais processos têm identificadores mais altos e que pode se comunicar com todos esses processos.

Existem três tipos de mensagem nesse algoritmo: uma mensagem de *eleição* enviada para convocar uma eleição; uma mensagem de *resposta* a esta; e, finalmente, uma mensagem de *coordenador*, para anunciar a identidade do processo eleito – o novo coordenador. Um processo inicia uma eleição quando observa, por meio dos tempos limites, que o coordenador falhou. Vários processos podem descobrir isso simultaneamente.

Como o sistema é síncrono, podemos construir um detector de falha confiável. Há um atraso máximo de transmissão de mensagem T_t e um atraso máximo T_p para processar uma mensagem. Portanto, podemos calcular um tempo $T = 2T_t + T_p$, que é um limite superior para o tempo total decorrido desde o envio de uma mensagem até o outro processo receber uma resposta. Se nenhuma resposta chegar dentro do tempo T , o detector de falha local poderá relatar que o destinatário da requisição falhou.

O processo que sabe que possui o identificador mais alto pode eleger a si mesmo como coordenador simplesmente enviando uma mensagem de *coordenador* para todos os processos com identificadores mais baixos. Por outro lado, um processo com um identificador mais baixo inicia uma eleição enviando uma mensagem de *eleição* para os processos que têm identificador mais alto e esperando uma mensagem de *resposta* em retorno. Se nenhuma resposta chegar dentro do tempo T , o processo se considerará o coordenador e enviará uma mensagem de *coordenador* para todos os processos com identificadores mais baixos, anunciando isso. Caso contrário, o processo esperará por mais um período T' que uma mensagem de *coordenador* chegue do novo coordenador. Se nenhuma resposta chegar, ele iniciará outra eleição.

Se um processo p_i recebe uma mensagem de *coordenador*, ele configura sua variável $eleito_i$ com o identificador do coordenador contido dentro dela e trata esse processo como coordenador.

Se um processo recebe uma mensagem de *eleição*, ele envia de volta uma mensagem de *resposta* e inicia outra eleição – a não ser que já tenha iniciado uma.

Quando um processo para substituir um processo falho é iniciado, ele inicia uma eleição. Se tiver o identificador de processo mais alto que os demais, decidirá que é o coordenador e anunciará isso para os outros processos. Assim, ele se tornará o coordenador, mesmo que o coordenador corrente esteja funcionando. É por isso que o algoritmo é chamado de “valentão”.

O funcionamento do algoritmo aparece na Figura 15.8. Existem quatro processos $p_1 - p_4$. O processo p_1 detecta a falha do coordenador p_4 e anuncia uma eleição (estágio 1 na figura). Ao receber uma mensagem de *eleição* de p_1 , os processos p_2 e p_3 enviam mensagens de *resposta* para p_1 e iniciam suas próprias eleições; p_3 envia uma mensagem de *resposta* para p_2 , mas p_3 não recebe nenhuma mensagem de *resposta* do processo falho p_4 (estágio 2). Portanto, ele decide que é o coordenador, mas antes que possa enviar a mensagem de *coordenador*, ele também falha (estágio 3). Quando o período do tempo limite T' de p_1 expira (o qual presumimos que ocorra antes que o tempo limite de p_2 expire), ele deduz a ausência de uma mensagem de *coordenador* e inicia outra eleição. Finalmente, p_2 é eleito coordenador (estágio 4).

Esse algoritmo satisfaz claramente a condição de subsistência E2, pela suposição do envio de mensagem confiável. E se nenhum processo for substituído, então o algoritmo satisfaz a condição E1. É impossível dois processos decidirem simultaneamente que são o coordenador, pois o processo com o identificador mais baixo descobrirá que o outro existe e o acatará.

Ainda assim, *não* é garantido que o algoritmo satisfaça a condição de segurança E1, caso processos que tenham falhado sejam substituídos por processos com o mesmo identificador. Um processo que substitui um processo falho p pode decidir que tem o identificador mais alto, assim como outro processo (que detectou a falha de p) decidiu que possui o identificador mais alto. Os dois processos se anunciarão como coordenadores, simultaneamente. Infelizmente, não há garantia da ordem do envio das mensagens, e os destinatários dessas mensagens, poderão chegar a diferentes conclusões sobre qual é o processo coordenador.

Além disso, a condição E1 pode ser violada, se os valores de tempo limite pressupostos se mostrarem imprecisos – isto é, se o detector de falha dos processos não for confiável.

Pegando o exemplo que acabamos de dar, suponha que p_3 não tivesse falhado, mas estava executando de forma extraordinariamente lenta (isto é, a suposição de que o sistema é síncrono está incorreta), ou que p_3 tivesse falhado, mas então foi substituído. Assim como p_2 envia sua mensagem de *coordenador*, p_3 (ou seu substituto) faz o mesmo. p_2 recebe a mensagem de *coordenador* de p_3 após ter enviado a sua própria e, portanto, configura $eleito_2 = p_3$. Devido aos atrasos de transmissão de mensagem variáveis, p_1 recebe a mensagem de *coordenador* de p_2 após a de p_3 e, assim, configura $eleito_1 = p_2$. A condição E1 foi violada.

Com relação ao desempenho do algoritmo, no melhor caso, o processo com o segundo identificador mais alto nota a falha do coordenador. Então, ele pode se eleger imediatamente e enviar $N - 2$ mensagens de coordenador. O tempo do ciclo de rotação é o de uma mensagem. O algoritmo valentão exige $O(N^2)$ mensagens, no pior caso – isto é, quando o processo com o menor identificador detecta primeiro a falha do coordenador. Nesse caso, $N - 1$ processos em conjunto iniciam eleições, cada um enviando mensagens para os processos com identificadores mais altos.

15.4 Coordenação e acordo na comunicação em grupo

Este capítulo examina os principais problemas de coordenação e acordo relacionados à comunicação em grupo – ou seja, como obter as propriedades da confiabilidade e da ordenação desejadas para todos os membros de um grupo. O Capítulo 6 apresentou a comunicação em grupo como um exemplo de técnica de comunicação indireta, por meio da qual os processos podem enviar mensagens para um grupo. Essa mensagem é propagada para todos os membros do grupo, com certas garantias em termos de confiabilidade e ordenação. Estamos buscando, particularmente, a confiabilidade em termos das propriedades da validade, integridade e acordo, e de ordenação FIFO, causal e total.

Neste capítulo, estudaremos a comunicação *multicast* para grupos de processos cuja participação como membro é conhecida. O Capítulo 18 expandirá nosso estudo para a comunicação de grupo completa, incluindo o gerenciamento de grupos que variam dinamicamente.

Modelo de sistema • O sistema contém um conjunto de processos, os quais podem se comunicar com confiabilidade por meio de canais um para um. Como antes, os processos podem falhar apenas por colapso.

Os processos são membros de grupos, os quais são os destinos das mensagens enviadas com a operação de *multicast*. Geralmente, é útil permitir que os processos sejam membros de vários grupos simultaneamente – por exemplo, para permitir que processos recebam informações de várias fontes, entrando em vários grupos. No entanto, para simplificar nossa discussão sobre as propriedades de ordenação, às vezes restringiremos os processos de modo a serem membros de no máximo um grupo por vez.

A operação $multicast(g, m)$ envia a mensagem m para todos os membros do grupo g (processos). Correspondentemente, existe uma operação $deliver(m)$ que distribui (entrega) uma mensagem recebida por *multicast* para o processo que a executa. Usamos o termo *distribuir* ou *entregar*, em vez de *receber*, para tornar claro que uma mensagem *multicast* nem sempre é passada para o aplicativo, assim que chega no nó do processo. Isso será mais bem explicado quando discutirmos a semântica da distribuição por *multicast*, em breve.

Toda mensagem m transporta o identificador exclusivo do processo $sender(m)$ que a enviou e o identificador de grupo de destino exclusivo $group(m)$. Supomos que os processos não mentem sobre a origem ou destinos das mensagens.

Alguns algoritmos presumem que os grupos são fechados (conforme definido no Capítulo 6).

15.4.1 Multicast básico

É interessante termos à nossa disposição uma primitiva de *multicast* básico que garanta, ao contrário do *multicast* IP, que um processo correto entregará a mensagem, desde que o difusor não falhe. Chamamos a primitiva de *B-multicast* e sua primitiva de entrega básica correspondente de *B-deliver*. Permitimos que os processos pertençam a vários grupos, e cada mensagem é destinada a algum grupo em particular.

Uma maneira simples de implementar *B-multicast* é usando uma operação *send* de um para um confiável, como segue:

Para $B\text{-}multicast}(g, m)$: para cada processo $p \in g$, $send(p, m)$;

Em $receive(m)$ em p : $B\text{-}deliver(m)$ em p .

A implementação pode usar *threads* para executar as operações *send* concorrentemente, em uma tentativa de reduzir o tempo total gasto para distribuir a mensagem. Infelizmente, tal implementação é propensa a sofrer a conhecida *explosão de confirmações*, caso o número de processos seja grande. Os sinais de confirmação, enviados como parte da operação *send* confiável, estão sujeitos a chegar de muitos processos quase ao mesmo tempo. Os *buffers* dos processos serão consumidos rapidamente e os sinais de confirmação podem ser perdidos. Portanto, ele retransmitirá a mensagem, acarretando ainda mais sinais de confirmações e mais desperdício de largura de banda de rede. Um serviço *multicast* básico mais prático pode ser construído, usando-se *multicast* IP, e convidamos o leitor a demonstrar isso no Exercício 15.10.

15.4.2 Multicast confiável

O Capítulo 6 discutiu o *multicast* confiável em termos de validade, integridade e acordo. Este capítulo complementa essa discussão informal, apresentando uma definição mais completa, a seguir.

Seguindo Hadzilacos e Toueg [1994] e Chandra e Toueg [1996], definiremos o *multicast confiável*, com as operações correspondentes *R-multicast* e *R-deliver* (o R é de “reliable”, que é “confiável” em inglês). Claramente, propriedades análogas à integridade e à validade são altamente desejáveis na distribuição por *multicast* confiável, mas acrescentamos outra: o requisito de que *todos* os processos corretos do grupo devem receber uma mensagem, caso *qualquer um* deles receba. É importante perceber que essa não é uma propriedade do algoritmo *B-multicast*, que é baseado em uma operação *send* de um para um confiável. O remetente pode falhar em qualquer ponto, enquanto *B-multicast* prossegue; portanto, alguns processos podem distribuir uma mensagem, enquanto outros não.

O *multicast* confiável é aquele que satisfaz as seguintes propriedades:

Integridade: um processo correto p entrega uma mensagem m no máximo uma vez. Além disso, $p \in group(m)$ e m foi fornecida para uma operação *multicast* por $sender(m)$. (Assim como acontece com a comunicação de um para um, as mensagens sempre podem ser diferenciadas por um número de sequência relativo aos seus remetentes.)

```

Na inicialização
Received := { };

Para o processo p enviar a mensagem m com R-multicast para o grupo g
B-multicast (g, m); // p ∈ g é incluído como destino

Em B-deliver(m) no processo q com g = group(m)
if (m ∈ Received)
then
    Received := Received ∪ {m};
    if (q ≠ p) then B-multicast (g, m); end if
    R-deliver m;
end if

```

Figura 15.9 Algoritmo de *multicast* confiável.

Validade: se um processo correto executa um *multicast* da mensagem m , então, ele distribuirá m .

Acordo: se um processo correto entrega a mensagem m , então todos os outros processos corretos em $group(m)$ distribuirão m .

A propriedade da integridade é análoga à da comunicação um para um confiável. A propriedade da validade garante a subsistência do remetente. Essa propriedade pode parecer incomum, pois é assimétrica (ela menciona apenas um processo em particular). No entanto, observe que, juntos, a validade e o acordo significam um requisito de subsistência global: se um processo (o remetente) distribuir uma mensagem m , então, como os processos corretos concordam com o conjunto de mensagens que distribuem, segue-se que m finalmente será entregue para todos os membros corretos do grupo.

A vantagem de expressar a condição de validade em termos de autoentrega é a simplicidade. O que necessitamos é que a mensagem seja entregue por *algum* membro correto do grupo.

A condição do acordo está relacionada à atomicidade, a propriedade do “tudo ou nada”, aplicada à entrega de mensagens para um grupo. Se um processo que envia por *multicast* uma mensagem falha antes de tê-la entregue, então é possível que a mensagem não seja entregue para nenhum processo do grupo; mas se ela for entregue para algum processo correto, então todos os outros processos corretos a entregaráão. Muitos artigos na literatura usam o termo “atômico” para incluir uma condição de ordenação total; definiremos isso em breve.

Implementação de multicast confiável por meio de B-multicast • A Figura 15.9 fornece um algoritmo de *multicast* confiável, com primitivas *R-multicast* e *R-deliver*, o qual permite aos processos pertencerem a vários grupos fechados simultaneamente. Para enviar uma mensagem com *R-multicast*, um processo a envia com *B-multicast* para os processos no grupo de destino (incluindo ele mesmo). Quando a mensagem é entregue com *B-deliver*, o destinatário, por sua vez, a entrega com *B-multicast* para o grupo (se ele não for o remetente original) e depois a entrega com *R-deliver*. Como uma mensagem pode chegar mais de uma vez, as duplicatas são detectadas e não enviadas.

Esse algoritmo claramente satisfaz a validade, pois um processo correto entregará a mensagem para si mesmo com *B-deliver*. Pela propriedade da integridade dos canais de comunicação subjacente usados em *B-multicast*, o algoritmo também satisfaz a propriedade da integridade.

O acordo resulta do fato de que todo processo correto envia a mensagem com *B-multicast* para os outros processos, após tê-la entregue com *B-deliver*. Se um processo correto não entregar a mensagem com *R-deliver*, então só pode ser porque ele nunca o fez com *B-deliver*. Isso, por sua vez, só pode ser porque nenhum outro processo correto a entregou com *B-deliver*; portanto, nenhum a enviará com *R-deliver*.

O algoritmo de *multicast* confiável que descrevemos é correto em um sistema assíncrono, pois não fizemos suposições de temporização. Contudo, o algoritmo é ineficiente para propósitos práticos. Cada mensagem é enviada $|g|$ vezes para cada processo.

Multicast confiável por meio de multicast IP • Uma realização alternativa de *R-multicast* é usar uma combinação *multicast IP*, confirmações “de carona” (isto é, confirmações anexadas em outras mensagens) e confirmações negativas. Esse protocolo *R-multicast* é baseado na observação de que a comunicação por *multicast IP* é frequentemente bem-sucedida. No protocolo, os processos não enviam mensagens de confirmação separadas; em vez disso, elas colocam as confirmações “de carona” das mensagens que enviam para o grupo. Os processos enviam uma mensagem de resposta separada apenas quando detectam que perderam uma mensagem. Uma resposta indicando a ausência de uma mensagem esperada é conhecida como *confirmação negativa*.

A descrição presume que os grupos são fechados. Cada processo p mantém um número de sequência S_g^p para cada grupo g ao qual pertence. O número de sequência inicialmente é zero. Cada processo também grava R_g^q , o número de sequência da última mensagem que recebeu do processo q enviada para o grupo g .

Para p enviar uma mensagem com *R-multicast* para o grupo g , ele coloca o valor S_g^p e a confirmação “de carona” na mensagem, na forma $\langle q, R_g^q \rangle$. Uma confirmação informa, para um remetente q , o número de sequência da mensagem mais recente de q , destinada a g , que p entregou desde que fez um *multicast*. Então, o emissor p envia a mensagem por *multicast IP* para g , com seus valores “de carona”, e incrementa S_g^p por um.

Os valores “de carona” em uma mensagem *multicast* permitem que os destinatários saibam sobre as mensagens que não receberam. Um processo entrega uma mensagem com *R-deliver*, destinada a g , contendo o número de sequência S de p , se e somente se $S = R_g^q + 1$, e incrementa R_g^q por um, imediatamente após a distribuição. Se uma mensagem recebida tem $S \leq R_g^q$, então r enviou a mensagem antes e a descarta. Se $S > R_g^q + 1$ ou se $R > R_g^q$ com uma confirmação incluída $\langle q, R \rangle$, então existe uma ou mais mensagens ainda não recebidas (e que provavelmente foram eliminadas, no primeiro caso). Ele mantém toda mensagem para a qual $S > R_g^q + 1$, em uma *fila de espera* (Figura 15.10) – tais filas são frequentemente usadas para satisfazer garantias de distribuição de mensagem. Ele solicita as mensagens ausentes enviando confirmações negativas para o remetente original, ou para um processo q a partir do qual recebeu uma confirmação $\langle q, R_g^q \rangle$, com R_g^q não menor do que o número de sequência exigido.

A fila de espera não é rigorosamente necessária para a confiabilidade, mas simplifica o protocolo, permitindo-nos usar números de sequência para representar conjuntos de mensagens enviadas. Ela também nos fornece uma garantia da ordem de envio (veja a Seção 15.4.3).

A propriedade da integridade resulta da detecção de duplicatas e das propriedades subjacentes do *multicast IP* (que usa somas de verificação para eliminar mensagens corrompidas). A propriedade da validade vale porque o *multicast IP* tem essa propriedade. Para o acordo, exigimos primeiro que um processo sempre possa detectar mensagens ausentes. Isso, por sua vez, significa que ele sempre receberá mais uma mensagem que permita detectar a omissão. Conforme o protocolo simplificado, garantimos a detecção

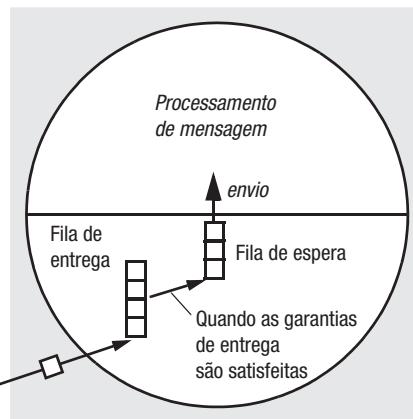


Figura 15.10 A fila de espera para mensagens *multicast* recebidas.

de mensagens ausentes apenas no caso em que processos corretos enviam mensagens por *multicast*, indefinidamente. Segundo, a propriedade do acordo exige que sempre haja uma cópia disponível de toda mensagem necessária para um processo que não a recebeu. Portanto, presumimos que os processos mantêm indefinidamente as cópias das mensagens que enviaram – nesse protocolo simplificado.

Nenhuma das suposições que fizemos para garantir o acordo é prática (veja o Exercício 15.15). Entretanto, o acordo é tratado praticamente em todos os protocolos dos quais o nosso é derivado: o protocolo Psync [Peterson *et al.* 1989], o protocolo Trans [Melliar-Smith *et al.* 1990] e o protocolo de *multicast* confiável escalável [Floyd *et al.* 1997]. Os protocolos Psync e Trans também fornecem outras garantias de ordenação de entrega.

Propriedades uniformes • A definição de acordo dada anteriormente se refere apenas ao comportamento de processos *corretos* – processos que nunca falham. Considere o que aconteceria no algoritmo da Figura 15.9 se um processo não fosse correto e falhasse após ter entregue uma mensagem com *R-deliver*. Como todo processo que entrega mensagem com *R-deliver* deve primeiro enviá-la com *B-multicast*, segue-se que todos os processos corretos terminarão por entregar a mensagem.

Toda propriedade que vale, sejam os processos corretos ou não, é chamada de propriedade *uniforme*. Definimos o acordo uniforme como segue:

Acordo uniforme: se um processo, correto ou falho, entregar uma mensagem m , então todos os processos corretos em $group(m)$ entreguerão.

O acordo uniforme permite que um processo falhe após ter enviado uma mensagem, enquanto ainda garante que todos os processos corretos entreguerão a mensagem. Argumentamos que o algoritmo da Figura 15.9 satisfaz essa propriedade, que é mais forte do que a propriedade do acordo não uniforme, definida anteriormente.

O acordo uniforme é útil em aplicações em que um processo pode executar uma ação que produz uma inconsistência observável antes de falhar. Por exemplo, considere que os processos são servidores gerenciando cópias de uma conta bancária, e que as atualizações na conta são enviadas para o grupo de servidores usando *multicast* confiável. Se o *multicast* não satisfizer o acordo uniforme, então um cliente que acesse um servidor imediatamente antes dele falhar poderá observar uma atualização que nenhum outro servidor processará.

É interessante notar que, se invertermos as linhas “*R-deliver m*” e “*if*($q \neq p$) *then B-multicast(g, m); end if*” na Figura 15.9, o algoritmo resultante não satisfará o acordo uniforme.

Assim como existe uma versão uniforme do acordo, também existem versões uniformes de qualquer propriedade de *multicast*, incluindo validade e integridade e as propriedades de ordenação que iremos definir.

15.4.3 Multicast ordenado

O algoritmo *multicast* básico da Seção 15.4.1 distribui as mensagens para processos em uma ordem arbitrária, devido aos atrasos aleatórios nas operações de envio de um para um subjacentes. Essa falta de garantia de ordem não é satisfatória para muitas aplicações. Por exemplo, em uma usina nuclear, pode ser importante que os eventos que signifiquem ameaças às condições de segurança e os eventos que signifiquem ações de unidades de controle sejam observados na mesma ordem por todos os processos do sistema.

Conforme discutido no Capítulo 6, os requisitos de ordenação comuns são: ordem total, ordem causal e ordem FIFO, junto a soluções mistas (em particular, causal-total e FIFO-total). Para simplificarmos nossa discussão, definiremos essas ordenações sob a suposição de que todo processo pertence no máximo a um grupo. Posteriormente, discutiremos as implicações resultantes de permitir que os grupos se sobreponham.

Ordem FIFO: se um processo correto executa $\text{multicast}(g, m)$ e depois $\text{multicast}(g, m')$, então todo processo correto que entregar m' entregará m antes de m' .

Ordem causal: se $\text{multicast}(g, m) \rightarrow \text{multicast}(g, m')$, onde \rightarrow é a relação acontece antes induzida apenas pelas mensagens enviadas entre os membros de g , então todo processo correto que entregar m' entregará m antes de m' .

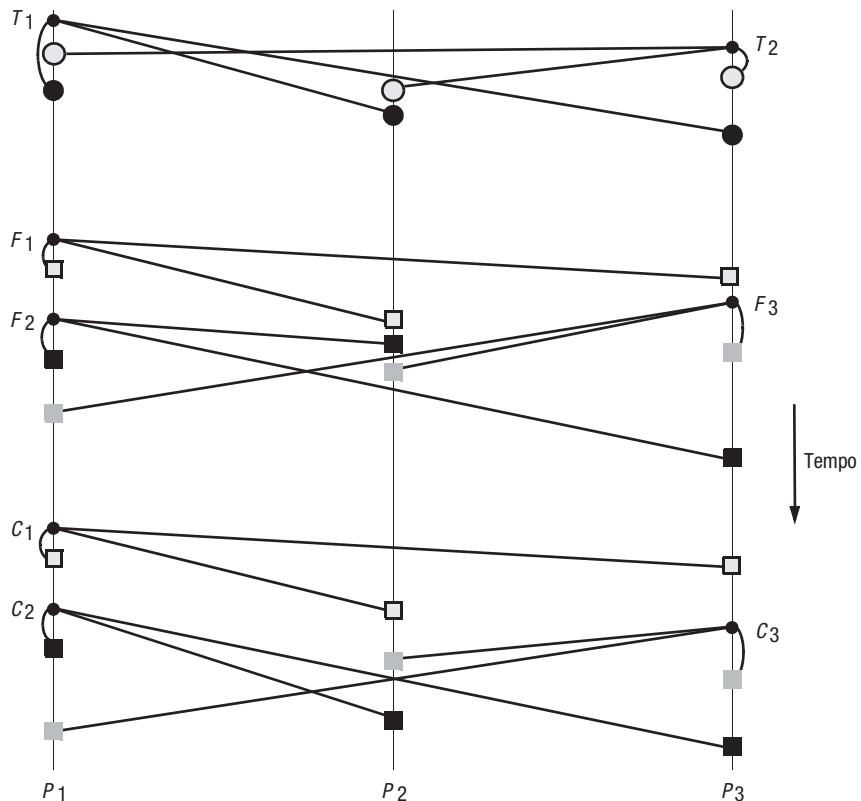
Ordem total: se um processo correto entregar a mensagem m antes de entregar m' , então qualquer outro processo correto que entregue m' entregará m antes de m' .

A ordem causal implica na ordem FIFO, pois quaisquer dois *multicasts* feitos pelo mesmo processo estão sujeitos à relação acontece antes. Note que a ordem FIFO e a ordem causal são ordenações apenas parciais: em geral, nem todas as mensagens são enviadas pelo mesmo processo; analogamente, alguns *multicasts* são concorrentes (não ordenados pela relação acontece antes).

A Figura 15.11 ilustra as ordenações para o caso de três processos. Uma inspeção detalhada da figura mostra que as mensagens totalmente ordenadas são enviadas na ordem oposta ao tempo físico em que foram enviadas. Na verdade, a definição de ordenação total permite que a distribuição de mensagens seja ordenada arbitrariamente, desde que a ordem seja a mesma em diferentes processos. Como a ordenação total não é necessariamente também uma ordenação FIFO, ou causal, definimos a mistura da ordenação FIFO-total como aquela para a qual a distribuição de mensagens obedece a ordem FIFO e a total; analogamente, sob a ordenação causal-total, a distribuição de mensagens obedece à ordenação causal e à total.

As definições de *multicast* ordenado não presumem nem implicam confiabilidade. Por exemplo, o leitor deve verificar que, na ordenação total, se o processo correto p entrega a mensagem m e depois m' , então um processo correto q poderá entregar m sem também entregar m' , ou qualquer outra mensagem ordenada após m .

Também podemos formar misturas dos protocolos ordenados e confiáveis. O *multicast* totalmente confiável é frequentemente referido na literatura como *multicast atômico*. Analogamente, podemos formar um *multicast* FIFO confiável, um *multicast* causal confiável e versões confiáveis de *multicasts* ordenados mistos.



Observe a ordem consistente das mensagens totalmente ordenadas T_1 e T_2 , das mensagens relacionadas com FIFO F_1 e F_2 e das mensagens relacionadas por causalidade C_1 e C_3 – e a ordem de distribuição arbitrária das mensagens.

Figura 15.11 Ordenação total, FIFO e causal de mensagens de *multicast*.

Ordenar a entrega de mensagens *multicast*, conforme veremos, pode ser dispendioso em termos de latência da distribuição e de consumo de largura de banda. A semântica da ordenação que descrevemos pode atrasar desnecessariamente a distribuição das mensagens. Isto é, no nível aplicativo, uma mensagem pode ser retardada por outra mensagem da qual, na verdade, não depende. Por isso, alguns têm proposto sistemas *multicast* que utilizam apenas a semântica de mensagem específica à própria aplicação para determinar a ordem de distribuição de mensagens [Cheriton e Skeen 1993, Pedone e Schiper 1999].

O exemplo de listas de discussão • Para tornar a semântica de distribuição por *multicast* mais concreta, considere uma aplicação na qual os usuários postam mensagens para listas de discussão. Cada usuário executa um processo aplicativo de lista de discussão. Cada tópico de discussão tem seu próprio grupo de processos. Quando um usuário posta uma mensagem em uma lista de discussão, o aplicativo envia em *multicast* a postagem do usuário para o grupo correspondente. O processo de cada usuário é um membro do grupo para o assunto no qual ele está interessado, de modo que o usuário receberá apenas as postagens relativas a esse tópico.

Lista de discussão: os.interesting		
Item	De	Assunto
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	desempenho do RPC
27	M.Walker	Re: Mach
fim		

Figura 15.12 Tela do programa de listas de discussão.

O *multicast* confiável é exigido se cada usuário precisa receberá cada postagem. Os usuários também têm requisitos de ordenação. A Figura 15.12 mostra as postagens conforme elas aparecem para um usuário em particular. No mínimo, a ordem FIFO é desejável, pois então cada postagem de determinado usuário – digamos, A.Hanlon – será recebida na mesma ordem, e os usuários poderão falar consistentemente sobre a segunda postagem de A.Hanlon.

Note que a mensagem cujos assuntos são “Re: Microkernels” (25) e “Re: Mach” (27) aparece após as mensagens às quais elas se referem. Um *multicast* ordenado por causalidade é necessário para garantir esse relacionamento. Caso contrário, atrasos de mensagem arbitrários poderiam significar, digamos, que uma mensagem “Re: Mach” poderia aparecer antes da mensagem original sobre Mach.

Se a distribuição por *multicast* fosse totalmente ordenada, então a numeração na coluna da esquerda seria consistente entre os usuários. Os usuários poderiam se referir sem ambiguidade, por exemplo, à “mensagem 24”.

Na prática, o sistema de listas de discussão USENET não implementa nem ordenação causal nem total. Os custos da comunicação para obter essas ordenações em larga escala superam suas vantagens.

Implementação da ordem FIFO • O *multicast* com ordem FIFO (com operações *FO-multicast* e *FO-deliver*) é obtida com números de sequência, como obteríamos para comunicação de um para um. Vamos considerar apenas grupos que não se sobrepõem. O leitor deve verificar que o protocolo de *multicast* confiável, que definimos sobre *multicast* IP, na Seção 15.4.2, também garante a ordem FIFO, mas mostraremos como se faz para construir um *multicast* com ordem FIFO sobre qualquer *multicast* básico dado. Usamos as variáveis S_g^p e R_g^q mantidas no processo p , do protocolo de *multicast* confiável da Seção 15.4.2: S_g^p é a contagem de quantas mensagens p enviou para g e, para cada q , S_g^p é o número de sequência da mensagem mais recente que p entregou do processo q , que foi enviada para o grupo g .

Para p enviar uma mensagem com *FO-multicast* para o grupo g , ele coloca o valor S_g^p “de carona” na mensagem, envia a mensagem com *B-multicast* para g e, em seguida, incrementa S_g^p por 1. Ao receber uma mensagem de q portando o número de sequência S , p verifica se $S = R_g^q + 1$. Se for, essa mensagem é a próxima esperada do remetente q e p a entrega com *FO-deliver*, configurando $R_g^q := S$. Se $S > R_g^q + 1$, ele coloca a mensagem na fila de espera até que as mensagens tenham sido entregues e $S = R_g^q + 1$.

Como todas as mensagens de determinado remetente são distribuídas na mesma sequência, e como a entrega de uma mensagem é retardada até que seu número de se-

quência tenha sido atingido, a condição para a ordem FIFO é claramente satisfeita. No entanto, isso só funciona sob a suposição de que os grupos não se sobrepõem.

Note que podemos usar qualquer implementação de *B-multicast* nesse protocolo. Além disso, se usarmos uma primitiva *R-multicast* confiável, em vez de *B-multicast*, obteremos um *multicast* FIFO confiável.

Implementação da ordem total • A estratégia básica para implementar a ordem total é atribuir identificadores totalmente ordenados às mensagens de *multicast* para que cada processo tome a mesma decisão de ordenação com base nesses identificadores. O algoritmo de distribuição é muito parecido com aquele que descrevemos para a ordem FIFO; a diferença é que os processos mantêm números de sequência específicos do grupo, em vez de números de sequência específicos do processo. Consideramos apenas como ordenar totalmente as mensagens enviadas para grupos que não se sobrepõem. Chamamos as operações de *multicast* de *TO-multicast* e *TO-deliver*.

Discutimos dois métodos principais para atribuir identificadores às mensagens. O primeiro deles usa um processo chamado *sequenciador* para essa atribuição (Figura 15.13). Um processo que queira enviar uma mensagem m com *TO-multicast* para o grupo g anexa nela um identificador exclusivo $id(m)$. As mensagens de g são enviadas para o sequenciador de g , $sequencer(g)$, assim como para os membros de g . (O sequenciador pode ser escolhido como um membro de g .) O processo $sequencer(g)$ mantém um número de sequência específico do grupo s_g , o qual utiliza para atribuir números de sequência cada vez maiores e consecutivos às mensagens que entrega com *B-deliver*. Ele anuncia os números de sequência por meio do envio de mensagens com *B-multicast* para g (veja os detalhes na Figura 15.13).

Uma mensagem permanecerá na fila de espera indefinidamente até que possa ser entregue com *TO-deliver*, de acordo com o número de sequência correspondente. Como os números de sequência são bem definidos (pelo sequenciador), o critério da ordem total é satisfeito. Além disso, se os processos usam uma variante de *B-multicast* com ordem FIFO, então o *multicast* totalmente ordenado também é ordenado por causalidade. Deixamos para o leitor demonstrar isso.

O problema óbvio de um esquema baseado em sequenciador é que ele pode se tornar um gargalo e um ponto único de falha. Existem algoritmos práticos que tratam do problema da falha. Chang e Maxemchuk [1984] foram os primeiros a sugerir um protocolo de *multicast* empregando um sequenciador (que chamaram de *site de ficha*). Kaashoek *et al.* [1989] desenvolveram um protocolo baseado em sequenciador para o sistema Amoeba. Esses protocolos garantem que uma mensagem esteja na fila de espera em $f + 1$ nós antes de ser entregue; assim, até f falhas podem ser toleradas. Assim como Chang e Maxemchuk, Birman *et al.* [1991] também empregaram um *site* contendo uma ficha que atua como sequenciador. A ficha pode ser passada de um processo para outro para que, por exemplo, se apenas um processo enviar *multicasts* totalmente ordenados, então esse processo poderá atuar como sequenciador, evitando comunicação.

O protocolo de Kaashoek *et al.* usa *multicast* baseado em *hardware* – disponível em uma rede Ethernet, por exemplo –, em vez da comunicação ponto a ponto confiável. Na variante mais simples desse protocolo, os processos enviam a mensagem *multicast* para o sequenciador, na base de um para um. O sequenciador faz um *multicast* para si mesmo, assim como o identificador e o número de sequência. Isso tem a vantagem de que os outros membros do grupo recebem apenas uma única mensagem *multicast*; sua desvantagem é a maior utilização de largura de banda. O protocolo está completamente descrito no endereço www.cdk5.net/coordination.

1. Algoritmo do membro do grupo p

Na inicialização: $r_g := 0;$

Para enviar a mensagem m com $T0$ -multicast para o grupo g

$B\text{-multicast}(g \cup \{\text{sequencer}(g)\}, \langle m, i \rangle);$

Em $B\text{-deliver}(\langle m, i \rangle)$ com $g = \text{group}(m)$

Coloca $\langle m, i \rangle$ na fila de espera;

Em $B\text{-deliver}(m_{ordem} = \langle "ordem", i, S \rangle)$ com $g = \text{group}(m_{ordem})$

espera até $\langle m, i \rangle$ na fila de espera e $S = r_g$;

$T0\text{-deliver } m;$ // (após excluí-la da fila de espera)

$r_g := S + 1;$

2. Algoritmo do sequenciador de g

Na inicialização: $s_g := 0;$

Em $B\text{-deliver}(\langle m, i \rangle)$ com $g = \text{group}(m)$

$B\text{-multicast}(g, \langle "ordem", i, s_g \rangle);$

$s_g := s_g + 1;$

Figura 15.13 Ordem total usando um sequenciador.

O segundo método que examinaremos para obter *multicast* totalmente ordenado é aquele no qual os processos concordam coletivamente sobre a atribuição de números de sequência às mensagens de maneira distribuída. Um algoritmo simples – semelhante àquele que foi originalmente desenvolvido para implementar distribuição por *multicast* totalmente ordenado para o *toolkit ISIS* [Birman e Joseph 1987a] – aparece na Figura 15.14. Mais uma vez, um processo envia sua mensagem com *B-multicast* para os membros do grupo. O grupo pode ser aberto ou fechado. Os processos receptores propõem números de sequência para as mensagens quando elas chegam e os retornam para o remetente, o qual os utiliza para gerar números de sequência acordados.

Cada processo q no grupo g mantém A_g^q , o maior número de sequência acordado observado até o momento para o grupo g , e p_g^q , seu próprio maior número de sequência proposto. O algoritmo do processo p para enviar por *multicast* uma mensagem m para o grupo g é:

1. p envia $\langle m, i \rangle$ para g com *B-multicast*, onde i é um identificador exclusivo para m .
2. Cada processo q responde ao remetente p com uma proposta para o número de sequência acordado da mensagem de $p_g^q := \text{Max}(A_g^q, P_g^q) + 1$. Na realidade, devemos incluir identificadores de processo nos valores p_g^q propostos, para garantir uma ordem total, pois de outro modo, diferentes processos poderiam propor o mesmo valor inteiro; mas, por simplicidade, não vamos tornar isso explícito aqui. Cada processo atribui provisoriamente o número de sequência proposto para a mensagem e a coloca em sua fila de espera, a qual é ordenada com o menor número de sequência na frente.
3. p coleta todos os números de sequência propostos e seleciona o maior a como o próximo número de sequência concordado. Então, ele envia $\langle i, a \rangle$ para g com

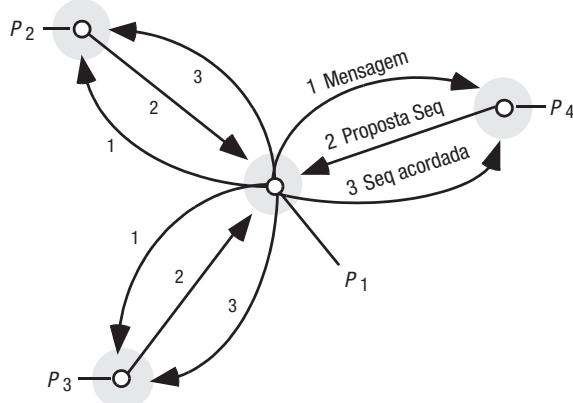


Figura 15.14 O algoritmo ISIS para ordem total.

B-multicast. Cada processo q em g configura $A_g^q := \text{Max}(A_g^q, a)$ e anexa a na mensagem (que é identificada por i). Ele reordena a mensagem na fila de espera, caso o número de sequência acordado seja diferente do proposto. Quando a mensagem que está na frente da fila de espera tiver recebido seu número de sequência acordado, ela será transferida para o fim da fila de entrega. Entretanto, as mensagens que receberam seu número de sequência acordado, mas que não estão na frente da fila de espera, ainda não são transferidas.

Se cada processo concordar com o mesmo conjunto de números de sequência e os entregar na ordem correspondente, então a ordem total será satisfeita. É claro que, em última análise, os processos corretos concordam com o mesmo conjunto de números de sequência, mas devemos mostrar que eles são monotonicamente cada vez maiores e que nenhum processo correto pode enviar uma mensagem prematuramente.

Suponha que uma mensagem m_1 tenha recebido um número de sequência acordado e tenha chegado na frente da fila de espera. Por construção, uma mensagem recebida após esse estágio será e deve ser entregue após m_1 : ela terá um número de sequência proposto maior e, portanto, um número de sequência acordado maior do que m_1 . Assim, seja m_2 qualquer outra mensagem que ainda não recebeu seu número de sequência acordado, mas que está na mesma fila. Temos que:

$$\text{SequênciaAcordada}(m_2) \geq \text{SequênciaProposta}(m_2)$$

pelo algoritmo que acabamos de ver. Como m_1 está na frente da fila:

$$\text{SequênciaProposta}(m_2) > \text{SequênciaAcordada}(m_1)$$

Portanto:

$$\text{SequênciaAcordada}(m_2) > \text{SequênciaAcordada}(m_1)$$

e a ordem total está garantida.

Esse algoritmo tem latência mais alta do que o *multicast* baseado em sequenciador: três mensagens são enviadas em série entre o remetente e o grupo, antes que uma mensagem possa ser entregue.

Algoritmo para membro de grupo p_i ($i = 1, 2, \dots, N$)

Na inicialização

$$V_i^g[j] := 0 \quad (j = 1, 2, \dots, N);$$

Para enviar a mensagem m com CO -multicast para o grupo g

$$V_i^g[i] := V_i^g[i] + 1;$$

$$B\text{-multicast } (g, < V_i^g, m>);$$

Em B -deliver ($< V_j^g, m>$) de p_j ($j \neq i$), com $g = group(m)$

coloca $< V_j^g, m>$ na fila de espera;

espera até que $V_j^g[j] = V_i^g[j] + 1$ e $V_j^g[k] \leq V_i^g[k]$ ($k \neq j$);

CO -deliver m ; // após removê-la da fila de espera

$$V_i^g[j] := V_i^g[j] + 1;$$

Figura 15.15 Ordem causal usando carimbos de tempo vetoriais.

Note que a ordem total escolhida por esse algoritmo não garante também a ordem por causalidade ou FIFO: quaisquer duas mensagens são enviadas em uma ordem total basicamente arbitrária, influenciada pelos atrasos da comunicação.

Para ver outras estratégias para implementar a ordem total, consulte Melliar-Smith *et al.* [1990], Garcia-Molina e Spauster [1991] e Hadzilacos e Toueg [1994].

Implementação da ordem causal • Fornecemos um algoritmo para grupos fechados que não se sobreponem, baseado naquele desenvolvido por Birman *et al.* [1991], mostrado na Figura 15.15, no qual as operações de *multicast* ordenados por causalidade são *CO-multicast* e *CO-deliver*. O algoritmo leva em conta o relacionamento acontece antes apenas quando é estabelecido por mensagens de *multicast*. Se os processos enviarem mensagens de um para um entre si, elas não serão consideradas.

Cada processo p_i ($i = 1, 2, \dots, N$) mantém seu próprio *carimbo de tempo vetorial* (veja a Seção 14.4). As entradas no vetor de carimbo de tempo contam o número de mensagens de *multicast* de cada processo que aconteceram antes da mensagem *multicast* seguinte ser enviada.

Para enviar uma mensagem com *CO-multicast* para o grupo g , o processo adiciona 1 em sua entrada de carimbo de tempo e envia a mensagem com *B-multicast*, junto a seu carimbo de tempo, para g .

Quando um processo p_i realiza *B-deliver* para uma mensagem enviada a partir de p_j , ele deve colocá-la na fila de espera, antes de poder entregá-la com *CO-deliver*: ou seja, até ter certeza de que entregou todas as mensagens que a precediam por causalidade. Para estabelecer isso, p_i espera até que (a) tenha entregue qualquer mensagem anterior enviada por p_j e (b) tenha entregue qualquer mensagem que p_j tenha distribuído no instante em que ele fez o *multicast* da mensagem. Essas duas condições podem ser detectadas examinando-se os carimbos de tempo vetoriais, como mostrado na Figura 15.15. Note que um processo pode entregar imediatamente para si mesmo, com *CO-deliver*, qualquer mensagem que tenha enviado com *CO-multicast*, embora isso não esteja descrito na Figura 15.15.

Cada processo atualiza seu carimbo de tempo vetorial ao entregar qualquer mensagem, para manter a contagem de mensagens de causalidade precedentes. Ele faz isso incrementando por um a j -ésima entrada de carimbo de tempo. Essa é uma otimização da operação *merge* que apareceu nas regras de atualização de relógios vetoriais na Seção 14.4. Podemos fazer a otimização em vista da condição de distribuição no algoritmo da Figura 15.15, a qual garante que apenas a j -ésima entrada aumentará.

Descrevemos, em linhas gerais, a prova da correção desse algoritmo, como segue. Suponha que $\text{multicast}(g, m) \rightarrow \text{multicast}(g, m')$. Sejam V e V' os carimbos de tempo vetoriais de m e m' , respectivamente. É fácil provar, por indução a partir do algoritmo, que $V < V'$. Em particular, se o processo p_k envia $\text{multicast } m$, então $V[k] \leq V'[k]$.

Considere o que acontece quando algum processo correto p_i entrega m' com *B-deliver* (em oposição a entregá-la com *CO-deliver*), sem primeiro entregar m com *CO-deliver*. De acordo com o algoritmo, $V_i[k]$ pode aumentar somente quando p_i entrega uma mensagem de p_k , quando aumenta por 1. No entanto, p_i não recebeu m e, portanto, $V_i[k]$ não pode aumentar além de $V[k] - 1$. Assim, não é possível que p_i entregue m' com *CO-deliver*, pois isso exigiria que $V_i[k] \geq V'[k]$ e, portanto, que $V_i[k] \geq V[k]$.

O leitor deve verificar que, se substituirmos a primitiva *R-multicast* confiável no lugar de *B-multicast*, obteremos um *multicast* que é tanto confiável como ordenado por causalidade.

Além disso, se combinarmos o protocolo de *multicast causal* com o protocolo baseado em sequenciador para distribuição totalmente ordenada, obteremos uma distribuição de mensagens total e causal. O sequenciador entrega mensagens de acordo com a ordem causal e faz o *multicast* dos números de sequência das mensagens na ordem em que os recebe. Os processos no grupo de destino não entregam uma mensagem até que tenham recebido uma mensagem de *order* do sequenciador e a mensagem seja a próxima na sequência de entrega.

Como o sequenciador envia a mensagem na ordem causal, e como todos os outros processos entregam mensagens na mesma ordem que o sequenciador, a ordenação é realmente tanto total como causal.

Sobreposição de grupos • Consideramos apenas os grupos que não se sobrepõem nas definições e algoritmos da semântica de ordenação FIFO, total e causal. Isso simplifica o problema, mas não é satisfatório, pois, em geral, os processos precisam ser membros de vários grupos sobrepostos. Por exemplo, um processo pode estar interessado nos eventos de várias fontes e, assim, entrar em um conjunto correspondente de grupos de distribuição de eventos.

Podemos estender as definições de ordenação para ordens globais [Hadzilacos e Toueg 1994], nas quais temos que considerar que, se a mensagem m é enviada por *multicast* para g e se a mensagem m' é enviada por *multicast* para g' , então as duas mensagens são endereçadas para os membros de $g \cap g'$.

Ordem FIFO global: se um processo correto executa $\text{multicast}(g, m)$ e então $\text{multicast}(g', m')$, então todo processo correto em $g \cap g'$ que entrega m' entregará m antes de m' .

Ordem causal global: se $\text{multicast}(g, m) \rightarrow \text{multicast}(g', m')$, onde \rightarrow é a relação que acontece antes induzida por um encadeamento de mensagens *multicast*, então todo processo correto em $g \cap g'$ que entrega m' entregará m antes de m' .

Ordem total com reconhecimento de pares: se um processo correto entrega a mensagem m enviada para g , antes de entregar m' enviada para g' , então qualquer outro processo correto em $g \cap g'$ que entregue m' entregará m antes de m' .

Ordem total global: seja ' $<$ ' a relação de ordem entre eventos de distribuição. Exigimos que ' $<$ ' obedeça à ordem total com reconhecimento de pares e que seja acíclica – sob a ordem total com reconhecimento de pares, ' $<$ ' não é acíclica por padrão.

Uma maneira de implementar essas ordens seria enviar por *multicast* cada mensagem m para o grupo de todos os processos no sistema. Cada processo descarta ou entrega a mensagem, de acordo com o fato de pertencer a $group(m)$. Essa seria uma implementação ineficiente e insatisfatória: um *multicast* deve envolver o mínimo de processos possível, além dos membros do grupo de destino. Alternativas são exploradas em Birman *et al.* [1991], Garcia-Molina e Spauster [1991], Hadzilacos e Toueg [1994], Kindberg [1995] e Rodrigues *et al.* [1998].

Multicast em sistemas síncronos e assíncronos • Nesta seção, descrevemos algoritmos para *multicast* desordenado confiável, *multicast* com ordem FIFO (confiável), *multicast* ordenado por causalidade (confiável) e *multicast* totalmente ordenado. Também indicamos como se faz para obter um *multicast* que é ordenado tanto totalmente como por causalidade. Deixamos o leitor projetar um algoritmo para uma primitiva de *multicast* que garanta ordenação tanto FIFO como total. Todos os algoritmos que descrevemos funcionam corretamente em sistemas assíncronos.

Entretanto, não apresentamos um algoritmo que garanta distribuição confiável e totalmente ordenada. Surpreendente como possa parecer, embora seja possível em um sistema síncrono, um protocolo com essas garantias é *impossível* em um sistema distribuído assíncrono – mesmo um que, na pior das hipóteses, tenha sofrido de uma única falha por colapso de processo. Voltaremos a este ponto na próxima seção.

15.5 Consenso e problemas relacionados

Esta seção apresenta o problema do consenso [Pease *et al.* 1980, Lamport *et al.* 1982] e os problemas relacionados dos generais bizantinos e da consistência interativa. Vamos nos referir a eles coletivamente como problemas de *acordo*. Grosso modo, o problema está relacionado ao fato de os processos concordarem com um valor após um, ou mais, dos processos terem proposto qual deve ser esse valor.

Por exemplo, no Capítulo 2, descrevemos uma situação na qual dois exércitos deviam decidir consistentemente sobre atacar ou recuar. Analogamente, podemos exigir que todos os computadores corretos controlando os motores de uma nave espacial precisam decidir se devem “prosseguir” ou se devem decidir “cancelar”, após cada um ter proposto uma ou outra ação. Em uma transação de transferência de fundos de uma conta para outra, os computadores envolvidos devem concordar consistentemente se realizarão o débito e o crédito respectivos. Na exclusão mútua, os processos concordam sobre qual processo pode entrar na seção crítica. Em uma eleição, os processos concordam sobre qual é o processo eleito. No *multicast* totalmente ordenado, os processos concordam sobre a ordem de distribuição da mensagem.

Existem protocolos personalizados para esses tipos individuais de acordo. Descrevemos alguns deles anteriormente, e os Capítulos 16 e 17 examinarão as transações. No entanto, é interessante considerarmos formas mais gerais de acordo, em busca de características e soluções comuns.

Esta seção define o consenso mais precisamente e o relaciona com três problemas de acordo conexos: generais bizantinos, consistência interativa e *multicast* totalmente ordenado. Examinaremos sob quais circunstâncias os problemas podem ser resolvidos e esboçaremos algumas soluções. Em particular, discutiremos o conhecido resultado da impossibilidade de Fischer *et al.* [1985], que diz que, em um sistema assíncrono, um conjunto de processos contendo apenas um processo falho não pode garantir um consenso. Finalmente, consideraremos como é que existem algoritmos práticos, a despeito do resultado da impossibilidade.

15.5.1 Modelo de sistema e definições do problema

Nosso modelo de sistema inclui um conjunto de processos p_i ($i = 1, 2, \dots, N$) comunicando-se por meio de passagem de mensagens. Um requisito importante que se aplica a muitas situações práticas é se chegar a um consenso mesmo na presença de falhas. Como antes, presumimos que a comunicação é confiável, mas que os processos possam falhar. Nesta seção, consideraremos falhas de processo bizantinas (arbitrárias), assim como as falhas por colapso. Às vezes, especificaremos a suposição de que até algum número f dos N processos são falhos – isto é, eles exibem alguns tipos de falha especificados; os outros processos são corretos.

Se puderem ocorrer falhas arbitrárias, então outro fator na especificação de nosso sistema é se os processos colocam uma assinatura digital nas mensagens que enviam (veja a Seção 11.4). Se os processos assinam suas mensagens, então um processo falho fica limitado quanto ao dano que pode causar. Especificamente, durante um algoritmo de acordo, ele não pode fazer uma reivindicação falsa sobre os valores enviados por um processo correto. A relevância da assinatura das mensagens se tornará mais clara quando discutirmos as soluções para o problema dos generais bizantinos. Por padrão, supomos que a assinatura não ocorre.

Definição do problema do consenso • Para chegar a um consenso, todo processo p_i começa no estado *indeciso* e propõe um único valor v_i , extraído de um conjunto D ($i = 1, 2, \dots, N$). Os processos se comunicam, trocando valores. Cada processo configura o valor de uma variável de decisão d_i . Ao fazer isso, ele entra no estado *decidido*, no qual não pode mais mudar d_i ($i = 1, 2, \dots, N$). A Figura 15.16 mostra três processos envolvidos em um algoritmo de consenso. Dois processos propõem “prosseguir” e um terceiro propõe “cancelar”, mas então falha. Cada um dos dois processos que permanecem corretos decide “prosseguir”.

Os requisitos de um algoritmo de consenso são que as seguintes condições devem valer para cada execução:

Término: cada processo correto acaba por configurar sua variável de decisão.

Acordo: o valor da decisão de todos os processos corretos é o mesmo: se p_i e p_j estão corretos e entraram no estado *decidido*, então $d_i = d_j$ ($i, j = 1, 2, \dots, N$).

Integridade: se todos os processos corretos propuseram o mesmo valor, então qualquer processo correto no estado *decidido* escolheu esse valor.

De acordo com a aplicação, variações na definição da integridade podem ser apropriadas. Por exemplo, um tipo de integridade mais fraca seria o valor da decisão ser igual a um valor que algum processo correto propôs – não necessariamente todos eles. Usaremos a definição declarada anteriormente.

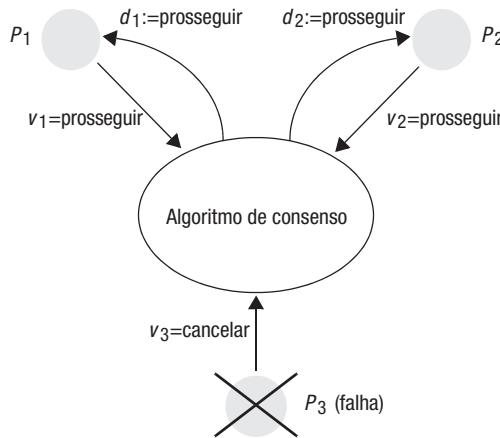


Figura 15.16 Consenso de três processos.

Para ajudar no entendimento de como a formulação do problema se transforma em um algoritmo, considere um sistema no qual os processos não podem falhar. Então, é simples resolver o consenso. Por exemplo, podemos reunir os processos em um grupo e fazer cada processo enviar em *multicast*, de modo confiável, seu valor proposto para os membros do grupo. Cada processo espera até que tenha reunido todos os N valores (incluindo o seu próprio). Então, ele avalia a função $majority(v_1, v_1, \dots, v_N)$, que retorna o valor que ocorre mais frequentemente entre seus argumentos, ou o valor especial $\perp \notin D$, caso não haja maioria. O término é garantido pela confiabilidade da operação de *multicast*. O acordo e a integridade são garantidos pela definição de *maioria* e pela propriedade da integridade de um *multicast* confiável. Todo processo recebe o mesmo conjunto de valores propostos e todo processo avalia a mesma função desses valores. Portanto, todos devem concordar e, se todo processo propôs o mesmo valor, então todos eles decidem por esse valor.

Note que *majority* (maioria) é a única função possível que os processos poderiam usar para concordar a respeito de um valor, a partir de valores candidatos. Por exemplo, se os valores fossem ordenados, as funções *minimum* e *maximum* (mínimo e máximo) poderiam ser apropriadas.

Se os processos podem falhar, isso introduz a complicação de detectar falhas, e não é imediatamente claro que uma execução do algoritmo de consenso possa terminar. Na verdade, se o sistema for assíncrono, poderá não terminar; voltaremos a esse ponto em breve.

Se os processos podem falhar de maneiras *arbitrarias* (bizantinas), os processos falhos podem, em princípio, comunicar valores aleatórios para os outros. Isso pode parecer improvável na prática, mas não está fora dos limites da possibilidade o fato de um processo com um erro falhar dessa maneira. Além disso, a falha pode não ser acidental, mas resultado de uma operação nociva ou mal-intencionada. Alguém poderia fazer um processo enviar deliberadamente valores diferentes para diferentes pares, em uma tentativa de frustrar os que estariam tentando chegar a um consenso. No caso de haver uma inconsistência, os processos corretos devem comparar os valores que receberam com os que os outros processos dizem ter recebido.

O problema dos generais bizantinos • Na declaração informal do *problema dos generais bizantinos* [Lamport *et al.* 1982], três ou mais generais devem concordar com um ataque ou com uma retirada. Um deles, o comandante, dá a ordem. Os outros, tenentes do comandante, devem decidir se vão atacar ou recuar. Contudo, um, ou mais, dos generais pode ser “traidor” – isto é, falho. Se o comandante é traidor, ele propõe atacar para um general e recuar para outro. Se um tenente é traidor, ele diz a um de seus pares que o comandante mandou atacar e para outro que eles devem recuar.

O problema dos generais bizantinos difere do consenso porque um processo distinto fornece um valor com o qual os outros devem concordar, em vez de cada um deles propor um valor. Os requisitos são:

Término: cada processo correto acaba por configurar sua variável de decisão.

Acordo: o valor de decisão de todos os processos corretos é o mesmo: se p_i e p_j estão corretos e entraram no estado *decidido*, então $d_i = d_j$ ($i, j = 1, 2, \dots, N$).

Integridade: se o comandante está correto, então todos os processos corretos decidem pelo valor proposto pelo comandante.

Note que, para o problema dos generais bizantinos, integridade implica em acordo quando o comandante está correto; mas o comandante não precisa estar correto.

Consistência interativa • O problema da consistência interativa é outra variante do consenso, na qual cada processo propõe um único valor. O objetivo do algoritmo é que os processos corretos concordem com um *vetor* de valores, um para cada processo. Chamaremos isso de “vetor de decisão”. Por exemplo, o objetivo poderia ser que cada processo de um conjunto de processos obtivesse a mesma informação sobre seus respectivos estados.

Os requisitos da consistência interativa são:

Término: cada processo correto acaba por configurar sua variável de decisão.

Acordo: o vetor de decisão de todos os processos corretos é o mesmo.

Integridade: se p_i está correto, então todos os processos corretos decidem por v_i como o i -ésimo componente de seus vetores.

Relação do consenso com outros problemas • Embora seja comum considerar o problema dos generais bizantinos com falhas de processo arbitrárias, na verdade cada um dos três problemas – consenso, generais bizantinos e consistência interativa – tem significado no contexto de falhas arbitrárias ou por colapso. Analogamente, cada um pode ser modelado supondo-se um sistema síncrono ou assíncrono.

Às vezes, é possível inferir uma solução para um problema usando a solução de outro. Essa é uma propriedade muito útil, tanto porque aumenta nosso entendimento dos problemas como porque, reutilizando soluções, podemos economizar trabalho de implementação e evitar complexidade.

Suponha que existam soluções para consenso (C), generais bizantinos (GB) e consistência interativa (CI), como segue:

$C_i(v_1, v_2, \dots, v_N)$ retorna o valor de decisão de p_i em uma execução da solução para o problema do consenso, onde v_1, v_2, \dots, v_N são os valores propostos pelos processos.

$GB_i(j, v)$ retorna o valor de decisão de p_j em uma execução da solução para o problema dos generais bizantinos, onde p_j , o comandante, propõe o valor v .

$CI_i(v_1, v_2, \dots, v_N)[j]$ retorna o j -ésimo valor no vetor de decisão de p_i em uma execução da solução para o problema da consistência interativa, onde v_1, v_2, \dots, v_N são os valores propostos pelos processos.

As definições de C_i , GB_i e CI_i presumem que um processo falho propõe um único valor especulativo, mesmo que possa ter fornecido diferentes valores propostos para cada um dos outros processos. Isso é apenas uma conveniência: as soluções não contarão com tal valor especulativo.

É possível construir soluções a partir das soluções para outros problemas. Fornece-mos três exemplos:

CI a partir de GB: construímos uma solução para CI a partir de GB, executando GB N vezes, uma vez com cada processo p_i ($i, j = 1, 2, \dots, N$) atuando como comandante:

$$CI_i(v_1, v_2, \dots, v_N)[j] = GB_i(j, v_j) \quad (i, j = 1, 2, \dots, N)$$

C a partir de CI: para o caso em que a maioria dos processos está correta, construímos uma solução para C a partir de CI executando CI para produzir um vetor de valores em cada processo, aplicando, então, uma função apropriada nos valores do vetor para derivar um único valor:

$$C_i(v_1, \dots, v_N) = majority(CI_i(v_1, \dots, v_N)[1], \dots, CI_i(v_1, \dots, v_N)[N])$$

($i = 1, 2, \dots, N$), onde *majority* (maioria) é conforme definido anteriormente.

GB a partir de C: construímos uma solução para GB a partir de C, como segue:

- O comandante p_j envia seu valor proposto v para si mesmo e para cada um dos processos restantes;
- Todos os processos executam C com os valores (v_1, v_2, \dots, v_N) que recebem (p_j pode ser falho);
- Eles derivam $GB_i(j, v) = C_i(v_1, v_2, \dots, v_N)$ ($i = 1, 2, \dots, N$).

O leitor deve verificar que as condições de término, acordo e integridade são preservadas em cada caso. Fischer [1983] relaciona os três problemas com mais detalhes.

Em sistemas com falhas por colapso, o consenso é equivalente a solucionar *multicast* confiável e totalmente ordenado: dada uma solução para um, podemos solucionar o outro. É fácil implementar o consenso com uma operação de *multicast* confiável e totalmente ordenado *RTO-multicast*. Reunimos todos os processos em um grupo g . Para chegar ao consenso, cada processo p_i executa $RTO\text{-}multicast(g, v_i)$. Então, cada processo p_i escolhe $d_i = m_i$, onde m_i é o primeiro valor entregue por p_i com *RTO-deliver*. A propriedade do término resulta da confiabilidade do *multicast*. As propriedades do acordo e da integridade resultam da confiabilidade e da ordenação total da distribuição por *multicast*. Chandra e Toueg [1996] demonstram como o *multicast* confiável e totalmente ordenado pode ser derivado a partir do consenso.

15.5.2 Consenso em um sistema síncrono

Esta seção descreve um algoritmo que usa apenas um protocolo de *multicast* básico para resolver o consenso em um sistema síncrono, embora seja baseado em uma forma modificada de exigência de integridade. O algoritmo presume que até f dos N processos apresentam falhas por colapso.

Algoritmo do processo $p_j \in g$; o algoritmo prossegue em $f + 1$ rodadas

Na inicialização

$$\text{Values}_j^1 := \{v_j\}; \text{Values}_j^0 := \{\};$$

Na rodada r ($1 \leq r \leq f + 1$)

$B\text{-multicast}(g, \text{Values}_j^r - \text{Values}_j^{r-1})$; // Envia apenas os valores que não foram enviados

$$\text{Values}_j^{r+1} := \text{Values}_j^r;$$

while (na rodada r)

{

Em B-deliver (V_j) de algum processo p_j

$$\text{Values}_j^{r+1} := \text{Values}_j^{r+1} \cup V_j;$$

}

Após ($f + 1$) rodadas

$$\text{Atribui } d_j = \min(\text{Values}_j^{f+1});$$

Figura 15.17 Consenso em um sistema síncrono.

Para chegar a um consenso, cada processo correto reúne os valores propostos pelos outros processos. O algoritmo prossegue em $f + 1$ rodadas, em cada uma das quais os processos corretos entregam os valores entre eles mesmos com $B\text{-multicast}$. Por suposição, no máximo f processos podem falhar. Na pior das hipóteses, todas as f falhas ocorreram durante as rodadas, mas o algoritmo garante que no final das rodadas todos os processos corretos que sobreviveram estão em condições de concordar.

O algoritmo, mostrado na Figura 15.17, é baseado no algoritmo de Dolev e Strong [1983] e em sua apresentação por Attiya e Welch [1998].

A variável Values_j^r contém o conjunto de valores propostos, conhecidos pelo processo p_j no início da rodada r . Cada processo envia por *multicast* o conjunto de valores que não enviou nas rodadas anteriores. Então, o processo recebe mensagens *multicast* de outros processos e registra todos os novos valores. Embora isso não apareça na Figura 15.17, a duração de uma rodada é limitada pela configuração de um tempo limite baseado no tempo máximo para um processo correto fazer o *multicast* de uma mensagem. Após $f + 1$ rodadas, cada processo escolhe como seu valor de decisão o valor mínimo que recebeu.

O término é óbvio, a partir do fato de que o sistema é síncrono. Para verificar a correção do algoritmo, devemos mostrar que cada processo chega ao mesmo conjunto de valores no final da última rodada. Então, resultam o acordo e a integridade, pois os processos aplicam a função *minimum* nesse conjunto.

Suponha, ao contrário, que dois processos difiram em seu conjunto de valores final. Sem perda de generalidade, um processo correto p_i possui um valor v que outro processo correto p_j ($i \neq j$) não possui. A única explicação para o fato de p_i possuir no final um valor proposto v que p_j não possui, é que um terceiro processo, digamos, p_k , que conseguiu enviar v para p_i , falhou antes que v pudesse ser enviado para p_j . Por sua vez, um processo enviando v na rodada anterior deve ter falhado, para explicar porque p_k possui v nessa rodada, mas p_j não o recebeu. Prosseguindo dessa maneira, temos que postular pelo menos uma falha em cada uma das rodadas anteriores. No entanto,

pressupomos que no máximo f falhas podem ocorrer e que existem $f + 1$ rodadas. Chegamos a uma contradição.

Verifica-se que qualquer algoritmo, para chegar a um consenso, a despeito de até f falhas por colapso, exige pelo menos $f + 1$ rodadas de trocas de mensagem, independentemente de como é construído [Dolev e Strong 1983]. Esse limite inferior também se aplica no caso das falhas bizantinas [Fischer e Lynch 1982].

15.5.3 O problema dos generais bizantinos em um sistema síncrono

Ao contrário do algoritmo de consenso descrito na seção anterior, supomos aqui que os processos podem apresentar falhas arbitrárias. Isto é, a qualquer momento, um processo falho pode enviar qualquer mensagem com qualquer valor; e ele pode omitir o envio de qualquer mensagem. Até f dos N processos podem ser falhos. Os processos corretos podem detectar a ausência de uma mensagem por meio de um tempo limite, mas não podem concluir que o remetente falhou, pois ele pode ficar em silêncio por algum tempo e, depois, enviar mensagens novamente.

Supomos que os canais de comunicação entre pares de processos são privados. Se um processo pudesse examinar todas as mensagens enviadas por outros processos, ele poderia detectar as inconsistências no que um processo falho envia para diferentes processos. Nossa suposição básica de confiabilidade do canal significa que nenhum processo falho pode injetar mensagens no canal de comunicação entre processos corretos.

Lamport *et al.* [1982] consideraram o caso de três processos enviando, um para o outro, mensagens não assinadas. Eles mostraram que não existe nenhuma solução que garanta o atendimento das condições do problema dos generais bizantinos, caso um processo possa falhar. Eles generalizaram esse resultado para mostrar que não existe nenhuma solução se $N \leq 3f$. Vamos demonstrar esses resultados em breve. Eles forneceram um algoritmo que resolve o problema dos generais bizantinos em um sistema síncrono se $N \geq 3f + 1$, para mensagens não assinadas (eles as chamam de mensagens “orais”).

Impossibilidade com três processos • A Figura 15.18 mostra dois cenários nos quais apenas um de três processos é falho. Na configuração da esquerda, um dos tenentes, p_3 , é falho; na da direita, o comandante, p_1 , é falho. Cada cenário da Figura 15.18 mostra duas rodadas de mensagens: os valores enviados pelo comandante e os valores que os tenentes enviam subsequentemente uns para os outros. Os prefixos numéricos servem para especificar as fontes das mensagens e para mostrar as diferentes rodadas. Leia o símbolo “:” nas mensagens como “fala”; por exemplo, “3:1:u” é a mensagem “3 fala 1 fala u ”.

No cenário da esquerda, o comandante envia corretamente o mesmo valor v para cada um dos outros dois processos, e p_2 ecoa isso corretamente para p_3 . Entretanto, p_3 envia um valor $u \neq v$ para p_2 . Nesse estágio, tudo que p_2 sabe é que recebeu valores diferentes; ele não pode identificar quais foram enviados pelo comandante.

No cenário da direita, o comandante é falho e envia valores diferentes para os tenentes. Após p_3 ter ecoado corretamente o valor x que recebeu, p_2 está na mesma situação em que estava quando p_3 era falho: ele recebeu dois valores diferentes.

Se existe uma solução, então o processo p_2 é obrigado a decidir pelo valor v quando o comandante estiver correto, de acordo com a condição da integridade. Se aceitarmos que nenhum algoritmo pode distinguir entre os dois cenários, p_2 também deve escolher o valor enviado pelo comandante no cenário da direita.

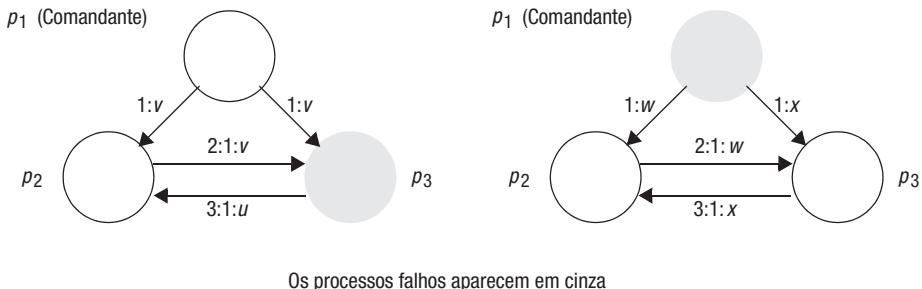


Figura 15.18 Três generais bizantinos.

Seguindo exatamente o mesmo raciocínio para p_3 , supondo que ele esteja correto, somos obrigados a concluir, por simetria, que p_3 também escolhe como seu valor de decisão o valor enviado pelo comandante. No entanto, isso contradiz a condição de acordo (o comandante envia valores diferentes, caso seja falho). Portanto, nenhuma solução é possível.

Note que esse argumento baseia-se em nossa intuição de que nada pode ser feito para aumentar o conhecimento do general correto, além do primeiro estágio, no qual ele não pode saber qual processo é falho. É possível provar a correção dessa intuição [Pease et al. 1980]. O acordo bizantino pode ser atingido para três generais, com um deles falho, caso os generais coloquem uma assinatura digital em suas mensagens.

Impossibilidade com $N \leq 3f$ • Pease et al. generalizaram o resultado básico da impossibilidade para três processos, para provar que nenhuma solução é possível se $N \leq 3f$. Em linhas gerais, o argumento é o seguinte: suponha que exista uma solução com $N \leq 3f$. Cada um dos três processos, p_1, p_2 e p_3 , usa a solução para simular o comportamento dos generais n_1, n_2 e n_3 , respectivamente, onde $n_1 + n_2 + n_3 = N$ e $n_1, n_2, n_3 \leq N/3$. Além disso, suponha que um dos três processos é falho. Os processos que simulam generais corretos realizam as interações de seus próprios generais internamente e enviam mensagens de seus generais para aqueles simulados pelos outros dois processos. O general simulado por um processo falho envia mensagens que podem ser espúrias, como parte da simulação para os outros dois processos. Como $N \leq 3f$ e $n_1, n_2, n_3 \leq N/3$, no máximo f generais simulados são falhos.

Como o algoritmo executado pelos processos é supostamente correto, a simulação termina. Os generais simulados corretos (nos dois processos corretos) concordam e satisfazem a propriedade da integridade. Contudo, agora, temos uma maneira de dois de três processos corretos chegarem a um consenso: cada um decide por um valor escolhido por todos os seus generais simulados. Isso contradiz nosso resultado da impossibilidade para três processos com um falho.

Solução com um único processo falho • Não há espaço suficiente para descrevermos completamente o algoritmo de Pease et al. que resolve o problema dos generais bizantinos em um sistema síncrono com $N \geq 3f+1$. Em vez disso, fornecemos o funcionamento do algoritmo para o caso $N \geq 4, f=1$ e o ilustramos para $N=4, f=1$.

Os generais corretos chegam a um acordo em duas rodadas de mensagens:

- Na primeira rodada, o comandante envia um valor para cada um dos tenentes.
- Na segunda rodada, cada um dos tenentes envia o valor que recebeu para seus pares.

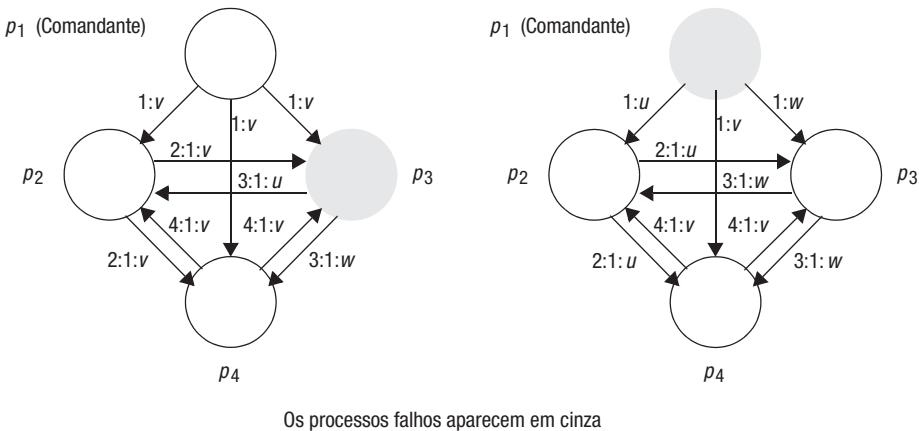


Figura 15.19 Quatro generais bizantinos.

Um tenente recebe um valor do comandante, mais $N - 2$ valores de seus pares. Se o comandante for falho, então todos os tenentes estão corretos e cada um terá reunido exatamente o conjunto de valores enviados pelo comandante. Caso contrário, um dos tenentes é falho; cada um de seus pares corretos recebe $N - 2$ cópias do valor enviado pelo comandante, mais um valor enviado pelo tenente falho.

Em qualquer caso, os tenentes corretos só precisam aplicar uma função de maioria simples no conjunto de valores que recebem. Como $N \geq 4$, $(N - 2) \geq 2$. Portanto, a função *majority* ignorará qualquer valor enviado por um tenente falho e produzirá o valor enviado pelo comandante, caso o comandante seja correto.

Agora, ilustraremos o algoritmo que acabamos de descrever em linhas gerais, para o caso de quatro generais. A Figura 15.19 mostra dois cenários semelhantes àqueles da Figura 15.18, mas, neste caso, existem quatro processos, sendo um deles falho. Como na Figura 15.18, na configuração da esquerda um dos tenentes, p_3 , é falho; na da direita, o comandante, p_1 , é falho.

No caso da esquerda, os dois processos tenentes corretos concordam, decidindo-se pelo valor do comandante:

$$p_2 \text{ decide-se por } \text{majority}(v, u, v) = v$$

$$p_4 \text{ decide-se por } \text{majority}(v, v, w) = v$$

No caso da direita, o comandante é falho, mas os três processos corretos concordam:

p_2, p_3 e p_4 decidem-se por $\text{majority}(u, v, w) = \perp$ (o valor especial \perp se aplica onde não existe nenhuma maioria de valores).

O algoritmo leva em conta o fato de que um processo falho pode omitir o envio de uma mensagem. Se um processo correto não recebe uma mensagem dentro de um limite de tempo conveniente (o sistema é síncrono), ele prossegue como se o processo falho tivesse enviado o valor \perp .

Discussão • Podemos medir a eficiência de uma solução para o problema dos generais bizantinos – ou qualquer outro problema de acordo – perguntando:

- Quantas rodadas de mensagem são necessárias? (Esse é um fator para o tempo que o algoritmo demora a terminar.)
- Quantas mensagens são enviadas e de que tamanho? (Isso mede a utilização de largura de banda total e tem um impacto sobre o tempo de execução.)

No caso geral ($f \geq 1$), o algoritmo de Lamport *et al.* para mensagens não assinadas funciona por $f + 1$ rodadas. Em cada rodada, um processo envia os valores que recebeu na rodada anterior para um subconjunto dos outros processos. O algoritmo é muito dispensioso: ele envolve o envio de $O(N^{f+1})$ mensagens.

Fischer e Lynch [1982] provaram que qualquer solução determinista para o consenso, supondo falhas bizantinas (e, portanto, para o problema dos generais bizantinos, conforme mostrou a Seção 15.5.1), exigirá pelo menos $f + 1$ rodadas de mensagem. Portanto, nesse aspecto, nenhum algoritmo pode funcionar mais rapidamente do que o de Lamport *et al.* Contudo, houve melhorias na complexidade da mensagem; veja, por exemplo, Garay e Moses [1993].

Vários algoritmos, como o de Dolev e Strong [1983], tiram proveito das mensagens assinadas. O algoritmo de Dolev e Strong, novamente, exige $f + 1$ rodadas, mas o número de mensagens enviadas é de apenas $O(N^2)$.

A complexidade e o custo das soluções sugerem que elas são aplicáveis apenas em situações em que a ameaça é grande. As soluções baseadas no conhecimento mais detalhado do modelo de falha podem ser mais eficientes [Barborak *et al.* 1993]. Se usuários mal-intencionados forem a fonte da ameaça, então um sistema, para opor-se a eles, provavelmente usará assinaturas digitais; uma solução sem assinaturas é impraticável.

15.5.4 Impossibilidade em sistemas assíncronos

Fornecemos soluções para o problema do consenso e dos generais bizantinos (e, daí, por derivação, para a consistência interativa). Entretanto, todas essas soluções contavam com o fato de o sistema ser síncrono. Os algoritmos presumem que as trocas de mensagem ocorrem em rodadas, e que os processos recebem tempos limites; além disso, presumem que um processo falho não enviou a eles uma mensagem dentro da rodada porque o atraso máximo foi excedido.

Fischer *et al.* [1985] provaram que nenhum algoritmo pode garantir um consenso em um sistema assíncrono, mesmo com a falha por colapso de um único processo. Em um sistema assíncrono, os processos podem responder às mensagens em tempos arbitrários; portanto, um processo falho não pode ser distinguido de um lento. A prova deles, que está fora dos objetivos deste livro, envolve mostrar que sempre existe alguma continuação da execução dos processos que evita que se chegue a um consenso.

A partir do resultado de Fischer *et al.*, sabemos imediatamente que, em um sistema assíncrono, não há nenhuma solução garantida para o problema dos generais bizantinos, da consistência interativa ou do *multicast* totalmente ordenado e confiável. Se houvesse tal solução, então, pelos resultados da Seção 15.5.1, teríamos uma solução para o consenso – contradizendo o resultado da impossibilidade.

Observe a palavra “garantia” na declaração do resultado da impossibilidade. O resultado não significa que os processos *nunca* podem chegar a um consenso distribuído em um sistema assíncrono, caso um deles seja falho. Ele permite que o consenso possa ser obtido com alguma probabilidade maior do que zero, confirmando o que sabemos na prática. Por exemplo, apesar do fato de nossos sistemas frequentemente

serem efetivamente assíncronos, os sistemas de transação têm chegado regularmente a um consenso há muitos anos.

Uma estratégia para contornar o resultado da impossibilidade é considerar sistemas *parcialmente síncronos*, os quais são suficientemente mais fracos do que os sistemas síncronos para serem úteis como modelos de sistemas práticos e suficientemente mais fortes do que os sistemas assíncronos para que o consenso possa ser resolvido neles [Dwork *et al.* 1988]. Essa estratégia está fora dos objetivos deste livro. Entretanto, três outras técnicas para contornar o resultado da impossibilidade, que vamos descrever agora, são o mascaramento de falhas, o consenso atingido por meio da exploração de detectores de falha e o consenso pela aleatoriedade dos aspectos do comportamento dos processos.

Mascaramento de falhas • A primeira técnica é evitar completamente o resultado da impossibilidade, mascarando todas as falhas de processo que ocorrerem (veja a Seção 2.4.2 para uma introdução ao mascaramento de falhas). Por exemplo, os sistemas de transação empregam armazenamento persistente, o qual sobrevive às falhas por colapso. Se um processo falha, ele é reiniciado (automaticamente ou manualmente, por um administrador). O processo armazena em um meio permanente, informações suficientes de pontos críticos para que, se falhar e for reiniciado, encontre dados suficientes para ser capaz de continuar corretamente sua tarefa interrompida. Em outras palavras, ele se comportará como um processo correto, mas que às vezes leva um longo tempo para executar uma etapa do processamento.

O mascaramento, geralmente, é aplicável ao projeto de sistemas. O Capítulo 16 discutirá como os sistemas transacionais tiram proveito do armazenamento persistente. O Capítulo 18 descreverá como as falhas de processo também podem ser mascaradas pela replicação de componentes de *software*.

Consenso usando detectores de falha • Outro método para contornar o resultado da impossibilidade emprega detectores de falha. Alguns sistemas práticos empregam detectores de falha “de projeto perfeito” para chegar ao consenso. Nenhum detector de falha em um sistema assíncrono que funcione unicamente pela passagem de mensagens pode ser realmente perfeito. Entretanto, os processos podem concordar em *julgar* falho um processo que não respondeu por mais do que um tempo limitado. Um processo que não responde pode não ter falhado realmente, mas os processos restantes agem como se ele tivesse falhado. Eles tornam a falha “silenciosa”, descartando as mensagens subsequentes que recebem de um processo considerado “falho”. Em outras palavras, transformamos efetivamente um sistema assíncrono em síncrono. Essa técnica é usada no sistema ISIS [Birman 1993].

Esse método conta com o fato de o detector de falha normalmente ser preciso. Quando ele é impreciso, o sistema tem de prosseguir sem um membro do grupo que, de outro modo, poderia ter contribuído para sua eficácia. Infelizmente, tornar o detector de falha razoavelmente preciso envolve o uso de valores de tempo limite longos, obrigando os processos a esperarem por um tempo relativamente grande (e a não realizar trabalho útil) antes de concluírem que um processo falhou. Outro problema que surge com essa estratégia é o particionamento da rede, que discutiremos no Capítulo 18.

Uma estratégia bastante diferente é usar detectores de falha imperfeitos e chegar ao consenso enquanto se permite que os processos suspeitos se comportem corretamente, em vez de excluí-los. Chandra e Toueg [1996] analisaram as propriedades que um detector de falha deve ter para resolver o problema do consenso em um sistema assíncrono.

Eles mostraram que o consenso pode ser resolvido em um sistema assíncrono, mesmo com um detector de falha não confiável, se menos de $N/2$ processos falharem e se a comunicação for confiável. O tipo de detector de falha para o qual isso vale é chamado de *detector de falha finalmente fraco*. Ele é:

Finalmente pouco completo: cada processo falho é finalmente tido como suspeito permanentemente por algum processo correto.

Finalmente pouco preciso: após algum ponto no tempo, pelo menos um processo correto nunca é tido como suspeito por qualquer processo correto.

Chandra e Toueg mostram que não podemos implementar um detector de falha finalmente fraco em um sistema assíncrono apenas pela passagem de mensagens. Entretanto, descrevemos um detector de falha baseado em mensagem, na Seção 15.1, que se adapta aos seus valores de tempo limite de acordo com os tempos de resposta observados. Se um processo (ou a conexão com ele) estiver muito lento, então o valor do tempo limite crescerá para que os casos de um processo falsamente suspeito se tornem raros. No caso de muitos sistemas reais, esse algoritmo se comporta de forma suficientemente parecida com um detector de falha finalmente fraco para propósitos práticos.

O algoritmo de consenso de Chandra e Toueg permite que processos falsamente suspeitos continuem com suas operações normais, e que os processos que suspeitaram deles recebam suas mensagens e processem essas mensagens normalmente. Isso complica a vida do programador de aplicativos, mas tem a vantagem de que os processos corretos não são desperdiçados pelo fato de serem falsamente excluídos. Além disso, os tempos limites para detecção de falhas podem ser configurados de forma menos conservadora do que com a estratégia do ISIS.

Consenso usando aleatoriedade • O resultado de Fischer *et al.* depende do que podemos considerar como “adversário”. Trata-se de um “personagem” (na verdade, apenas um conjunto de eventos aleatórios) que pode explorar os fenômenos dos sistemas assíncronos para frustrar as tentativas dos processos de chegarem ao consenso. O adversário manipula a rede para atrasar as mensagens, de modo que elas cheguem justamente no momento errado e, analogamente, desacelera ou acelera os processos o suficiente para que eles estejam no estado “errado” ao receberem uma mensagem.

A terceira técnica que trata do resultado da impossibilidade é a introdução de um elemento de chance no comportamento dos processos, de modo que o adversário não possa exercer sua estratégia de impedimento eficientemente. O consenso pode, ainda, não ter sido atingido, em alguns casos, mas esse método permite que os processos cheguem a um consenso em um tempo finito *esperado*. Um algoritmo probabilístico que resolve o consenso, mesmo com falhas bizantinas, pode ser encontrado em Canetti e Rabin [1993].

15.6 Resumo

Este capítulo começou discutindo a necessidade dos processos de acessar recursos compartilhados sob condições de exclusão mútua. As travas (*locks*) nem sempre são implementadas pelos servidores que gerenciam os recursos compartilhados e, então, exige-se um serviço separado de exclusão mútua distribuída. Foram considerados três algoritmos que obtêm exclusão mútua: um empregando um servidor central, um baseado em anel e

um algoritmo baseado em *multicast*, usando relógios lógicos. Nenhum desses mecanismos, conforme os descrevemos, pode suportar falhas, embora possam ser modificados para tolerar algumas delas.

Em seguida, este capítulo considerou um algoritmo baseado em anel e o algoritmo valentão, cujo objetivo comum é eleger um processo exclusivamente a partir de um dado conjunto – mesmo que várias eleições ocorram concomitantemente. O algoritmo valentão poderia ser usado, por exemplo, para eleger um novo servidor de tempo mestre ou um novo servidor de travas (*lock server*), quando o anterior falhar.

Este capítulo descreveu a coordenação e o acordo na comunicação em grupo. Ele discutiu o *multicast* confiável, na qual os processos corretos concordam com o conjunto de mensagens a serem enviadas, e o *multicast* com ordenação de entrega FIFO, causal e total. Fornecemos algoritmos para *multicast* confiável e para todos os três tipos de ordem de entrega.

Finalmente, descrevemos os problemas de consenso, generais bizantinos e consistência interativa. Definimos as condições para sua solução e mostramos os relacionamentos entre esses problemas – incluindo o relacionamento entre consenso e *multicast* totalmente ordenado confiável.

Existem soluções em um sistema síncrono, e descrevemos algumas delas. Na verdade, existem soluções mesmo quando são possíveis falhas arbitrárias. Descrevemos, em linhas gerais, parte da solução para o problema dos generais bizantinos de Lamport *et al.* Algoritmos mais recentes têm menor complexidade, mas, em princípio, nenhum pode ser melhor do que as $f+1$ rodadas exigidas por esse algoritmo, a não ser que as mensagens tenham assinatura digital.

O capítulo terminou descrevendo o resultado fundamental de Fischer *et al.* a respeito da impossibilidade de garantir o consenso em um sistema assíncrono. Discutimos como, contudo, os sistemas chegam regularmente a um acordo em sistemas assíncronos.

Exercícios

- 15.1 É possível implementar um (processo) detector de falha confiável, ou um não confiável, usando um canal de comunicação não confiável? *página 632*
- 15.2 Se todos os processos clientes são *single-threaded*, a condição de exclusão mútua ME3, que especifica entrada na ordem acontece antes, é relevante? *página 635*
- 15.3 Forneça uma fórmula para o rendimento máximo (*throughput*) de um sistema de exclusão mútua em termos do atraso de sincronização. *página 635*
- 15.4 No algoritmo do servidor central para exclusão mútua, descreva uma situação na qual duas requisições não são processadas na ordem acontece antes. *página 636*
- 15.5 Adapte o algoritmo do servidor central para exclusão mútua para tratar da falha por colapso de qualquer cliente (em qualquer estado), supondo que o servidor seja correto e dado um detector de falha confiável. Comente se o sistema resultante é tolerante a falhas. O que aconteceria se um cliente que possuísse a ficha fosse erroneamente suspeito de ter falhado? *página 636*
- 15.6 Dê um exemplo de execução do algoritmo baseado em anel para mostrar que os processos não têm necessariamente a entrada garantida na seção crítica na ordem acontece antes. *página 637*

- 15.7 Em determinado sistema, cada processo normalmente usa uma seção crítica muitas vezes, antes que outro processo a solicite. Explique por que o algoritmo de exclusão mútua baseado em *multicast* de Ricart e Agrawala é inefficiente para esse caso e descreva como fazer para melhorar seu desempenho. Sua adaptação satisfaz a condição de subsistência ME2?
página 639
- 15.8 No algoritmo valentão, um processo de recuperação inicia uma eleição e se tornará o novo coordenador, caso tenha um identificador mais alto do que o encarregado atual. Essa é uma característica necessária do algoritmo?
página 644
- 15.9 Sugira como fazer para adaptar o algoritmo valentão para tratar com particionamentos temporários de rede (comunicação lenta) e processos lentos.
página 646
- 15.10 Projete um protocolo para *multicast* básico sobre *multicast IP*.
página 647
- 15.11 Como, se houver possibilidade, as definições de integridade, acordo e validade do *multicast* confiável devem mudar para o caso de grupos abertos?
página 647
- 15.12 Explique por que inverter a ordem das linhas ‘*R-deliver m*’ e ‘*if (q ≠ p) then B-multicast(g, m); end if*’, na Figura 15.9, faz com que o algoritmo não satisfaça mais o acordo uniforme. O algoritmo de *multicast* confiável baseado no *multicast IP* satisfaz o acordo uniforme?
página 648
- 15.13 Explique se o algoritmo de *multicast* confiável sobre *multicast IP* funciona para grupos abertos, assim como para grupos fechados. Dado qualquer algoritmo para grupos fechados, como podemos simplesmente derivar um algoritmo para grupos abertos?
página 649
- 15.14 Explique como fazer para adaptar o algoritmo de *multicast* confiável sobre *multicast IP* para eliminar a fila de espera – para que uma mensagem recebida que não seja uma duplicata possa ser entregue imediatamente, mas sem quaisquer garantias de ordenação. Dica: use conjuntos de números em sequência para representar as mensagens entregues até o momento.
página 649
- 15.15 Considere como fazer para tratar das suposições impraticáveis que fizemos para atender às propriedades de validade e acordo para o protocolo de *multicast* confiável baseado no *multicast IP*. Dica: acrescente uma regra para excluir as mensagens retidas, quando elas tiverem sido entregues para todas as partes; e considere a inclusão de uma mensagem de “pulsão”, que nunca é entregue para o aplicativo, mas que o protocolo envia se o aplicativo não tiver nenhuma mensagem para enviar.
página 649
- 15.16 Mostre que o algoritmo de *multicast* de ordem FIFO não funciona para grupos sobrepostos, considerando duas mensagens enviadas da mesma fonte para dois grupos sobrepostos e considerando um processo na interseção desses grupos. Adapte o protocolo para funcionar para esse caso. Dica: os processos devem incluir com suas mensagens os números de sequência mais recentes das mensagens enviadas para *todos* os grupos.
página 649
- 15.17 Mostre que, se o *multicast* básico que usamos no algoritmo da Figura 15.13 também tiver ordem FIFO, então o *multicast* totalmente ordenado resultante também será ordenado por causalidade. É verdade que qualquer *multicast* ordenado com FIFO e totalmente ordenado é, por isso, ordenado por causalidade?
página 655
- 15.18 Sugira como fazer para adaptar o protocolo de *multicast* ordenado por causalidade para manipular grupos sobrepostos.
página 657

-
- 15.19 Na discussão do algoritmo de exclusão mútua de Maekawa, demos um exemplo de três subconjuntos de um conjunto de três processos que poderiam levar a um impasse. Use esses subconjuntos como grupos de *multicast* para mostrar como uma ordenação total com reconhecimento de pares não é necessariamente acíclica. *página 658*
- 15.20 Construa uma solução para o *multicast* totalmente ordenado e confiável em um sistema síncrono, usando um *multicast* confiável e uma solução para o problema do consenso. *página 659*
- 15.21 Fornecemos uma solução para o consenso a partir de uma solução para o *multicast* totalmente ordenado e confiável, que envolvia selecionar o primeiro valor a ser enviado. Explique, a partir dos primeiros princípios, por que em um sistema assíncrono não poderíamos, em vez disso, derivar uma solução usando um serviço de *multicast* confiável, mas não totalmente ordenado, e a função de “maioria”. (Note que, se pudéssemos, isso contradiria o resultado da impossibilidade de Fischer *et al.* [1985]!) Dica: considere processos lentos/falhos. *página 663*
- 15.22 Considere o algoritmo dado, na Figura 15.17, para consenso em um sistema síncrono, o qual usa a seguinte definição de integridade: se todos os processos, sejam corretos ou não, propusessem o mesmo valor, então qualquer processo correto no estado decidido escolheria esse valor. Agora, considere uma aplicação na qual os processos corretos podem propor resultados diferentes; por exemplo, executando diferentes algoritmos para decidir a ação a ser executada na operação de um sistema de controle. Sugira uma modificação apropriada na definição de integridade e, portanto, no algoritmo. *página 664*
- 15.23 Mostre que o acordo bizantino pode ser conseguido por três generais, com um deles falho, caso os generais coloquem uma assinatura digital em suas mensagens. *página 665*

16

Transações e Controle de Concorrência

- 16.1 Introdução
- 16.2 Transações
- 16.3 Transações aninhadas
- 16.4 Travas
- 16.5 Controle de concorrência otimista
- 16.6 Ordenação da indicação de tempo
- 16.7 Comparação dos métodos de controle de concorrência
- 16.8 Resumo

Este capítulo discute a aplicação de transações e controle de concorrência em objetos compartilhados gerenciados por servidores.

Uma transação define uma sequência de operações no servidor que garante que elas sejam atômicas na presença de várias falhas de clientes e de servidor. As transações aninhadas são estruturadas a partir de conjuntos de outras transações. Elas são particularmente úteis nos sistemas distribuídos porque permitem uma maior concorrência.

Todos os protocolos de controle de concorrência são baseados no critério da equivalência serial e são derivados das regras de conflitos entre operações. Serão descritos três métodos:

- Travas (*locks*) são usadas para ordenar transações que acessam os mesmos objetos, de acordo com a ordem de chegada de suas operações nos objetos.
- O controle de concorrência otimista permite que as transações prossigam até estarem prontas para serem confirmadas (*commit*); após isso, é feita uma verificação para ver se elas efetuaram operações conflitantes nos objetos.
- A ordenação por carimbo de tempo (*timestamp*) ordena as transações que acessam os mesmos objetos de acordo com seus tempos iniciais.

16.1 Introdução

O objetivo das transações é garantir que todos os objetos gerenciados por um servidor permaneçam em um estado consistente ao serem acessados por várias transações e na presença de falhas do servidor. O Capítulo 2 apresentou um modelo de falha para sistemas distribuídos. As transações tratam falhas por colapso de processos e falhas por omissão na comunicação, mas não com qualquer tipo de comportamento arbitrário (ou bizantino). O modelo de falha para transações será apresentado na Seção 16.1.2.

Os objetos que podem ser recuperados depois que seu servidor falha são chamados de *objetos recuperáveis*. Em geral, os objetos gerenciados por um servidor podem ser armazenados em memória volátil (por exemplo, a RAM) ou em memória persistente (por exemplo, o disco). Mesmo que os objetos sejam armazenados em memória volátil, o servidor pode usar memória persistente para armazenar informações suficientes para que, no caso de falhas do processo servidor, o estado dos objetos seja recuperado. Isso permite que os servidores tornem os objetos recuperáveis. Uma transação é especificada por um cliente como um conjunto de operações sobre os objetos a serem executadas como uma unidade indivisível pelos servidores que estão gerenciando esses objetos. Os servidores devem garantir que a transação inteira seja executada e que os resultados sejam gravados no armazenamento permanente ou, no caso de um ou mais deles falhar, que seus efeitos sejam completamente desconsiderados. O próximo capítulo discutirá problemas relacionados às transações que envolvem vários servidores, em particular, como eles decidem a respeito do resultado de uma transação distribuída. Este capítulo trata dos problemas de uma transação em um único servidor. A transação de um cliente também é considerada indivisível do ponto de vista das transações dos outros clientes, no sentido de que as operações de uma transação não podem observar os efeitos parciais das operações de outra. A Seção 16.1.1 discutirá a sincronização simples de acesso aos objetos, e a Seção 16.2 apresentará as transações, as quais exigem técnicas mais avançadas para evitar interferência entre os clientes. A Seção 16.3 discutirá as transações aninhadas. As Seções 16.4 a 16.6 discutirão três métodos de controle de concorrência para transações cujas operações são todas endereçadas a um único servidor (travamento, controle de concorrência otimista e ordenação por carimbo de tempo). O Capítulo 17 discutirá como esses métodos são ampliados para uso com transações cujas operações são endereçadas a vários servidores.

Para explicar alguns dos assuntos deste capítulo, usaremos um exemplo de transações bancárias, mostradas na Figura 16.1. Cada conta é representada por um objeto remoto cuja interface *Account* fornece operações para fazer depósitos e saques e para solicitar e determinar o saldo. Cada agência do banco é representada por um objeto remoto cuja interface *Branch* fornece operações para criar uma nova conta, pesquisar uma conta pelo nome e solicitar os fundos totais nessa agência.

16.1.1 Sincronização simples (sem transações)

Um dos principais problemas deste capítulo é que, a não ser que um servidor seja projetado cuidadosamente, as operações executadas em nome de diferentes clientes às vezes podem interferir umas nas outras. Tal interferência pode resultar em valores incorretos nos objetos. Nesta seção, discutiremos como as operações de clientes podem ser sincronizadas sem apelar para as transações.

Operações atômicas no servidor • Vimos nos capítulos anteriores que o uso de várias *threads* é vantajoso para o desempenho em muitos servidores. Também observamos que o uso de *threads* permite que operações de vários clientes sejam efetuadas concorrentemente e possi-

Operações da interface *Account*

<i>deposit(amount)</i>	deposita <i>amount</i> na conta
<i>withdraw(amount)</i>	saca <i>amount</i> da conta
<i>getBalance() → amount</i>	retorna o saldo da conta
<i>setBalance(amount)</i>	configura o saldo da conta como <i>amount</i>

Operações da interface *Branch*

<i>create(name) → account</i>	cria uma nova conta com um nome informado
<i>lookUp(name) → account</i>	retorna uma referência para a conta com o nome informado
<i>branchTotal() → amount</i>	retorna o total de todos os saldos da agência

Figura 16.1

velmente acessam os mesmos objetos. Portanto, os métodos dos objetos devem ser projetados para uso em um contexto *multithreaded*. Por exemplo, se os métodos *deposit* e *withdraw* (depósito e saque) não forem projetados para serem usados em um programa *multithreaded*, é possível que as ações de duas ou mais execuções concorrentes do método sejam interpostas arbitrariamente e tenham efeitos estranhos nas variáveis de instância dos objetos *conta*.

O Capítulo 7 explicou o uso da palavra-chave *synchronized*, que pode ser aplicada em métodos na linguagem Java para garantir que apenas uma *thread* por vez possa acessar um objeto. Em nosso exemplo, a classe que implementa a interface *Account* poderá declarar os métodos como *synchronized*. Por exemplo:

```
public synchronized void deposit(int amount) throws RemoteException{
    // adiciona amount ao saldo da conta
}
```

Se uma *thread* invoca um método sincronizado em um objeto, então o acesso a esse objeto é travado (*locked*); assim, se outra *thread* invocar um de seus métodos sincronizados, será bloqueada até que a trava (*lock*) seja liberada. Essa forma de sincronização obriga a execução das *threads* a ser separada no tempo e garante que as variáveis de instância de um único objeto sejam acessadas de maneira consistente. Sem a sincronização, duas invocações distintas de *deposit* poderiam ler o saldo antes de uma delas o ter incrementado, resultando em um valor incorreto. Qualquer método que acesse uma variável de instância que possa variar deve ser sincronizado.

As operações que estão livres de interferência de operações concorrentes sendo executadas por outras *threads* são chamadas de *operações atômicas*. O uso de métodos sincronizados em Java é uma maneira de obter operações atômicas. No entanto, em outros am-

bientes de programação de servidores *multithreaded*, as operações sobre os objetos ainda precisam ter operações atômicas para manter seus objetos consistentes. Isso pode ser conseguido com o uso de qualquer mecanismo de exclusão mútua disponível, como um *mutex*.

Melhorando a cooperação dos clientes por meio da sincronização das operações do servidor • Os clientes podem usar um servidor como uma maneira de compartilhar recursos. Isso é conseguido com alguns clientes usando operações para atualizar os objetos do servidor e outros clientes usando operações para acessá-los. O esquema de acesso sincronizado aos objetos, mencionado anteriormente, fornece tudo que é necessário para muitas aplicações – ele impede que as *threads* interfiram umas nas outras. Entretanto, algumas aplicações exigem uma maneira das *threads* se comunicarem entre si.

Por exemplo, pode surgir uma situação na qual a operação solicitada por um cliente não possa ser concluída até que uma operação solicitada por outro cliente tenha sido efetuada. Isso pode acontecer quando alguns clientes são produtores e outros são consumidores – talvez os consumidores tenham que esperar até que um produtor tenha fornecido mais algumas das mercadorias em questão. Isso também pode ocorrer quando os clientes estão compartilhando um recurso – os clientes que precisam do recurso talvez tenham que esperar que outros clientes o liberem. Veremos, posteriormente neste capítulo, que uma situação semelhante surge quando travamentos ou carimbos de tempo (*timestamps*) são usados para controle de concorrência em transações.

Os métodos Java *wait* e *notify*, apresentados no Capítulo 7, permitem que as *threads* se comuniquem de uma maneira que resolve os problemas mencionados anteriormente. Eles devem ser usados dentro dos métodos sincronizados de um objeto. Uma *thread* chama *wait* em um objeto, para ficar suspensa e permitir que outra *thread* execute um método desse objeto. Uma *thread* chama *notify* para informar qualquer outra *thread* que esteja esperando nesse objeto que ela alterou alguns de seus dados. O acesso a um objeto ainda é atômico quando as *threads* esperam umas pelas outras: uma *thread* que chama *wait* libera sua trava (*lock*) e fica suspensa como uma única ação atômica; quando for reiniciada, após ser notificada, ela deve adquirir uma nova trava sobre o objeto e retoma a execução após a espera. Uma *thread* que chama *notify* (dentro de um método sincronizado) conclui a execução desse método antes de liberar a trava sobre o objeto.

Considere a implementação de um objeto compartilhado *Queue*, com dois métodos: *first*, que remove e retorna o primeiro objeto da fila; e *append*, que adiciona um objeto dado no final da fila. O método *first* testará se a fila está vazia, no caso em que chamará *wait* na fila. Se um cliente invocar *first* quando a fila estiver vazia, ele não obterá uma resposta até que outro cliente tenha adicionado algo na fila – a operação *append* chamará *notify* quando tiver adicionado um objeto na fila. Isso permite que uma das *threads* que esteja esperando no objeto fila retome a operação e retorne o primeiro objeto da fila para seu cliente. Quando as *threads* sincronizam suas ações em um objeto por meio de *wait* e *notify*, o servidor “segura” as requisições que não podem ser atendidas imediatamente, e o cliente espera por uma resposta até que outro cliente tenha produzido o que ele precisa.

Na seção posterior, sobre travamentos de transações, discutiremos a implementação de travas (*locks*) como um objeto com operações sincronizadas. Quando os clientes tentam adquirir uma trava, talvez tenham que esperar até que a trava seja liberada por outros clientes.

Sem a capacidade de sincronizar *threads* dessa maneira, um cliente que não pode ser atendido imediatamente (por exemplo, um cliente que invoca o método *first* em uma fila vazia), é orientado a tentar novamente em um momento posterior. Isso é insatisfatório, pois envolverá o cliente na consulta sequencial ao servidor, e o servidor na execução de requisições extras. Isso também é potencialmente injusto, pois outros clientes podem fazer suas requisições antes de esperar que o cliente tente novamente.

16.1.2 Modelo de falha para transações

Lamson [1981] propôs um modelo de falhas para transações distribuídas que considera falhas de discos, servidores e comunicação. Nesse modelo, a alegação é de que os algoritmos funcionam corretamente na presença de falhas previsíveis, mas nenhuma alegação é feita a respeito de seu comportamento quando ocorre um desastre. Embora possam ocorrer erros, eles podem ser detectados e tratados, antes que qualquer comportamento incorreto ocorra. O modelo diz o seguinte:

- As escritas em armazenamento permanente podem falhar, ou não gravando nada ou gravando um valor errado – por exemplo, escrever no bloco errado é um desastre. O armazenamento de arquivos também pode deteriorar. As leituras feitas no armazenamento permanente podem detectar (por meio de uma soma de verificação) quando um bloco de dados está danificado.
- Os servidores podem falhar ocasionalmente. Quando um servidor danificado é substituído por um novo processo, primeiramente sua memória volátil é colocada em um estado no qual não conhece nenhum dos valores (por exemplo, de objetos) anteriores à falha. Depois disso, ele executa um procedimento de recuperação, usando informações do meio de armazenamento permanente e aquelas obtidas a partir de outros processos, para configurar os valores dos objetos, incluindo aqueles relacionados ao protocolo de confirmação (*commit*) de duas fases (veja a Seção 17.6). Quando um processo falha, ele é obrigado a parar para que seja impedido de enviar mensagens errôneas e de escrever valores errados no meio de armazenamento permanente – isto é, ele não pode produzir falhas arbitrárias. Falhas podem ocorrer a qualquer momento; em particular, elas podem ocorrer durante a recuperação.
- Pode haver um atraso arbitrário antes da chegada de uma mensagem. Uma mensagem pode ser perdida, duplicada ou corrompida. O destinatário pode detectar mensagens corrompidas (por meio de uma soma de verificação). Tanto as mensagens falsificadas como as mensagens corrompidas não detectadas são consideradas desastres.

O modelo de falha para armazenamento permanente, processadores e comunicações foi usado para projetar um sistema estável, cujos componentes podem sobreviver a qualquer falha e que apresenta um modelo de falha simples. Em particular, o *armazenamento estável* fornece uma operação de *escrita atômica* na presença de uma falha da operação de *escrita* ou de uma falha por colapso do processo. Isso foi obtido por meio da replicação de cada bloco em dois blocos de disco. Uma operação de *escrita* é aplicada no par de blocos do disco – e no caso de uma única falha, um bloco sem defeito está sempre disponível. Um *processador estável* usa armazenamento estável para permitir a recuperação de seus objetos após uma falha. Os erros de comunicação são mascarados pelo uso de um mecanismo de chamada de procedimento remoto confiável.

16.2 Transações

Em algumas situações, os clientes exigem que uma sequência de requisições separadas para um servidor seja atômica, no sentido de que:

1. estejam livres da interferência de operações sendo efetuadas em nome de outros clientes concorrentes; e
2. todas as operações sejam concluídas com êxito, ou não tenham nenhum efeito na presença de falhas do servidor.

Transaction T:

```
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);
```

Figura 16.2 Transação bancária de um cliente.

Voltemos ao nosso exemplo de transações bancárias para ilustrar as transações. Um cliente que executa uma sequência de operações sobre uma conta bancária em particular, em nome de um usuário, primeiramente pesquisará (com *lookUp*) a conta pelo nome e depois aplicará as operações *deposit*, *withdraw* e *getBalance* (depósito, saque e obter saldo) diretamente na conta relevante. Em nossos exemplos, usamos contas com nomes *A*, *B* e *C*. O cliente as pesquisa e armazena referências para elas nas variáveis *a*, *b* e *c* de tipo *Account*. Os detalhes da pesquisa das contas pelo nome e as declarações das variáveis foram omitidos dos exemplos.

A Figura 16.2 mostra um exemplo de transação simples de cliente, especificando uma série de ações relacionadas, envolvendo as contas bancárias *A*, *B* e *C*. As duas primeiras ações transferem \$100 de *A* para *B* e as duas outras transferem \$200 de *C* para *B*. Um cliente obtém uma operação de transferência fazendo um saque, seguido de um depósito.

As transações são originárias dos sistemas de gerenciamento de banco de dados. Naquele contexto, uma transação é a execução de um programa que acessa um banco de dados. As transações foram introduzidas nos sistemas distribuídos na forma de servidores de arquivo transacionais, como o XDFS [Mitchell e Dion 1982]. No contexto de um servidor de arquivo transacional, uma transação é a execução de uma sequência de requisições de cliente para operações de arquivo. Foram fornecidas transações sobre objetos distribuídos em vários sistemas de pesquisa, incluindo o Argus [Liskov 1988] e o Arjuna [Shrivastava *et al.* 1991]. Neste último contexto, uma transação consiste na execução de uma sequência de requisições de cliente como, por exemplo, pode-se ver na Figura 16.2. Do ponto de vista do cliente, uma transação é uma sequência de operações que formam um único passo, transformando os dados do servidor de um estado consistente para outro.

As transações podem ser fornecidas como parte do *middleware*. Por exemplo, o CORBA fornece a especificação de um *Object Transaction Service* (*Serviço de Transação de Objeto*) [OMG 2003] com interfaces IDL que permitem às transações dos clientes incluírem diversos objetos em vários servidores. São fornecidas operações para o cliente especificar o início e o fim de uma transação. O cliente ORB mantém um contexto para cada transação, o qual propaga com cada operação nessa transação. No CORBA, os objetos transacionais são invocados dentro do escopo de uma transação e geralmente têm algum meio de armazenamento persistente associado.

Em todos esses contextos, uma transação se aplica aos objetos recuperáveis e se destina a ser atômica. Frequentemente, ela é chamada de *transação atômica*. Existem dois aspectos na atomicidade:

Tudo ou nada: ou uma transação termina com êxito e os efeitos de todas as suas operações são gravados nos objetos ou (se ela falhar ou for deliberadamente cancelada) não há efeito nenhum. Essa propriedade de tudo ou nada tem mais dois aspectos próprios:

Atomicidade da falha: os efeitos são atômicos mesmo quando o servidor falha.

Durabilidade: após uma transação ter sido concluída com êxito, todos os seus efeitos são salvos no armazenamento permanente. Usamos o termo armazenamento permanente para nos referirmos aos arquivos mantidos no disco ou em outro meio permanente. Os dados escritos em um arquivo sobreviverão, caso o processo servidor falhe.

Isolamento: cada transação deve ser realizada sem interferência de outras transações; em outras palavras, os efeitos intermediários de uma transação não devem ser visíveis por outras transações. O quadro a seguir apresenta um mnemônico – ACID – para lembrar as propriedades das transações atômicas.

Para suportar os requisitos de atomicidade da falha e de durabilidade, os objetos devem ser *recuperáveis*; isto é, quando um processo servidor falha inesperadamente, devido a uma falha de *hardware* ou a um erro de *software*, as alterações atribuídas a todas as transações concluídas devem estar disponíveis no armazenamento permanente para que, quando o servidor for substituído por um novo processo, ele possa recuperar os objetos para refletir o efeito do tudo ou nada. Quando um servidor reconhece o término da transação de um cliente, todas as alterações da transação feitas nos objetos devem ter sido gravadas no armazenamento permanente.

Um servidor que suporta transações deve sincronizar as operações o suficiente para garantir que o requisito do isolamento seja satisfeito. Uma maneira de fazer isso é executar as transações em série – uma por vez, em alguma ordem arbitrária. Infelizmente, essa solução geralmente seria inaceitável para servidores cujos recursos são compartilhados por vários usuários interativos. Em nosso exemplo de transações bancárias, é desejável permitir que vários funcionários do banco realizem transações bancárias *online* ao mesmo tempo que os outros.

O objetivo de qualquer servidor que suporte transações é maximizar a concorrência. Portanto, as transações podem ser executadas concomitantemente, caso tenham o mesmo efeito que uma execução serial – isto é, elas são *serialmente equivalentes* ou *serializáveis*.

Propriedades ACID

Härder e Reuter [1983] sugeriram o mnemônico ACID para lembrar das propriedades das transações:

Atomicidade: uma transação deve ser do tipo tudo ou nada;

Consistência: uma transação leva o sistema de um estado consistente para outro estado consistente;

Isolamento;

Durabilidade.

Não incluímos a consistência em nossa lista de propriedades das transações porque geralmente os programadores de servidores e clientes são responsáveis por garantir que as transações deixem o banco de dados consistente.

Como exemplo de consistência, suponha que no exemplo de transações bancárias um objeto contenha a soma dos saldos de todas as contas e seu valor seja usado como resultado de *branchTotal*. Os clientes podem obter a soma dos saldos de todas as contas usando *branchTotal* ou chamando *getBalance* em cada uma das contas. Por consistência, eles devem obter o mesmo resultado a partir de ambos os métodos. Para manter essa consistência, as operações *deposit* e *withdraw* devem atualizar o objeto que contém a soma dos saldos de todas as contas.

openTransaction() → trans;

inicia uma nova transação e gera um TID *trans* exclusivo. Esse identificador será usado nas outras operações da transação.

closeTransaction(trans) → (commit, abort);

termina uma transação: um valor de retorno *commit* indica que a transação foi confirmada; um valor de retorno *abort* indica que ela foi cancelada.

abortTransaction(trans);

cancela a transação.

Figura 16.3 Operações na interface *Coordinator*.

Recursos para transação podem ser adicionados aos servidores de objetos recuperáveis. Cada transação é criada e gerenciada por um coordenador, o qual implementa a interface *Coordinator*, mostrada na Figura 16.3. O coordenador fornece, a cada transação, um identificador ou TID. O cliente invoca o método *openTransaction* do coordenador para introduzir uma nova transação – um identificador de transação, ou TID, é alocado e retornado. No final de uma transação, o cliente invoca o método *closeTransaction* para indicar seu final – todos os objetos recuperáveis acessados pela transação devem ser salvos. Se, por algum motivo, o cliente quiser cancelar uma transação, ele invoca o método *abortTransaction* – todos os seus efeitos devem desaparecer.

Uma transação é obtida pela cooperação entre um programa cliente, alguns objetos recuperáveis e um coordenador. O cliente especifica a sequência de invocações nos objetos recuperáveis que devem compreender uma transação. Para obter isso, o cliente envia com cada invocação o identificador de transação retornado por *openTransaction*. Uma maneira de tornar isso possível é incluir um argumento extra em cada operação de um objeto recuperável para transportar o TID. Por exemplo, no serviço de transação bancária, a operação *deposit* poderia ser definida como:

deposit(trans, amount)

deposita *amount* na conta para a transação com TID *trans*

Quando as transações são fornecidas como *middleware*, o TID pode ser passado implicitamente com todas as invocações remotas entre *openTransaction* e *closeTransaction* ou *abortTransaction*. É isso que o serviço de transação do CORBA faz. Não vamos mostrar TIDs em nossos exemplos.

Normalmente, uma transação termina quando o cliente faz uma requisição de *closeTransaction*. Se a transação progrediu normalmente, a resposta diz que ela foi *confirmada* – isso constitui um compromisso para o cliente de que todas as alterações solicitadas na transação foram gravadas permanentemente, e que as transações futuras, que acessem os mesmos dados, verão os resultados de todas as alterações feitas durante a transação.

Como alternativa, talvez a transação tenha que ser cancelada por um de vários motivos relacionados à natureza da transação em si, a conflitos com outra transação ou à falha de um processo ou de um computador. Quando uma transação é cancelada, as partes envolvidas (os objetos recuperáveis e o coordenador) devem garantir que nenhum de seus efeitos seja visível para as futuras transações, ou nos objetos ou em suas cópias no meio de armazenamento permanente.

Uma transação é bem-sucedida, ou é cancelada, de uma de duas maneiras – o cliente a cancela (usando uma chamada de *abortTransaction* para o servidor) ou o servidor a

<i>Bem-sucedida</i>	<i>Cancelada pelo cliente</i>	<i>Cancelada pelo servidor</i>
<i>openTransaction</i>	<i>openTransaction</i>	<i>openTransaction</i>
<i>operação</i>	<i>operação</i>	<i>operação</i>
<i>operação</i>	<i>operação</i>	<i>operação</i>
•	•	o servidor cancela
•	•	<i>transação →</i>
<i>operação</i>	<i>operação</i>	<i>ERRO de operação</i> <i>relatado ao cliente</i>
<i>closeTransaction</i>	<i>abortTransaction</i>	

Figura 16.4 Alternativas para realização de transações.

cancela. A Figura 16.4 mostra essas três possibilidades alternativas das transações. Nesses dois casos, nos referimos a uma transação como *fallha*.

Ações de serviço relacionadas a falhas de processo • Se um processo servidor falha inesperadamente, ele acaba por ser substituído. O novo processo servidor cancela todas as transações não confirmadas e usa um procedimento de recuperação para restaurar os valores dos objetos com os valores produzidos pela transação confirmada mais recentemente. Para lidar com um cliente que falhou inesperadamente durante uma transação, os servidores podem fornecer um tempo de expiração a cada transação e cancelar toda transação que não tiver terminado antes de seu tempo de expiração.

Ações de cliente relacionadas a falhas do processo servidor • Se um servidor falhar enquanto uma transação está em andamento, o cliente saberá disso quando uma das operações retornar uma exceção, após um tempo limite. Se um servidor falhar e, depois, durante o andamento de uma transação, for substituído, a transação não será mais válida, e o cliente deverá ser informado como uma exceção para a próxima operação. Em qualquer caso, o cliente deverá, então, formular um plano, possivelmente consultando o usuário humano, para a conclusão ou cancelamento da tarefa da qual a transação fazia parte.

16.2.1 Controle de concorrência

Esta seção ilustra dois conhecidos problemas de transações concorrentes no contexto do exemplo de transações bancárias – o problema da *atualização perdida* e o problema das *recuperações inconsistentes*. Esta seção mostrará como esses dois problemas podem ser evitados com o uso de execuções equivalentes em série das transações. Supomos em toda parte que cada uma das operações *deposit*, *withdraw*, *getBalance* e *setBalance* é sincronizada – isto é, seus efeitos sobre a variável de instância que altera o saldo de uma conta são atômicos.

O problema da atualização perdida • O problema da atualização perdida é ilustrado pelas duas transações a seguir, nas contas bancárias *A*, *B* e *C*, cujos saldos iniciais são de \$100, \$200 e \$300 respectivamente. A transação *T* transfere um valor da conta *A* para a conta *B*. A transação *U* transfere um valor da conta *C* para a conta *B*. Nos dois casos, o valor

Transação <i>T</i> :	Transação <i>U</i> :	
<i>balance = b.getBalance();</i>	<i>balance = b.getBalance();</i>	
<i>b.setBalance(balance * 1.1);</i>	<i>b.setBalance(balance * 1.1);</i>	
<i>a.withdraw(balance / 10)</i>	<i>c.withdraw(balance / 10)</i>	
 <i>balance = b.getBalance();</i>	 <i>balance = b.getBalance();</i>	\$200
		\$200
<i>b.setBalance(balance * 1.1);</i>	<i>b.setBalance(balance * 1.1);</i>	\$220
<i>a.withdraw(balance / 10)</i>	<i>c.withdraw(balance / 10)</i>	\$80
		\$280

Figura 16.5 O problema da atualização perdida.

transferido é calculado de forma a aumentar o saldo de *B* em 10%. O efeito líquido da execução das transações *T* e *U* sobre a conta *B* deve ser o aumento do seu saldo em 10%, duas vezes; portanto, seu valor final será de \$242.

Agora, considere os efeitos de permitir que as transações *T* e *U* sejam executadas concorrentemente, como na Figura 16.5. As duas transações obtêm o saldo de *B* como \$200 e depois depositam \$20. O resultado é incorreto, aumentando o saldo da conta *B* por \$20, em vez de \$42. Essa é uma ilustração do problema da atualização perdida. A atualização de *U* é perdida porque *T* a sobrescreve sem vê-la. As duas transações leem o valor antigo, antes que qualquer uma escreva o novo valor.

Da Figura 16.5 em diante, mostramos as operações que afetam o saldo de uma conta em linhas sucessivas, e o leitor deve supor que uma operação em uma linha específica é executada depois da que está na linha acima dela.

Recuperações inconsistentes • A Figura 16.6 mostra outro exemplo relacionado a uma conta bancária, no qual a transação *V* transfere uma quantia da conta *A* para *B* e a transação *W* ativa o método *branchTotal* para obter a soma dos saldos de todas as contas do

Transação <i>V</i> :	Transação <i>W</i> :	
<i>a.withdraw(100)</i>	<i>aBranch.branchTotal()</i>	
<i>b.deposit(100)</i>		
 <i>a.withdraw(100);</i>	 <i>total = a.getBalance()</i>	\$100
		\$100
<i>b.deposit(100)</i>	<i>total = total + b.getBalance()</i>	\$300
	<i>total = total + c.getBalance()</i>	
	•	
	•	

Figura 16.6 O problema das recuperações inconsistentes.

Transação <i>T</i> :	Transação <i>U</i> :	
<i>b.balance = b.getBalance()</i>		<i>b.balance = b.getBalance()</i>
<i>b.setBalance(balance * 1.1)</i>		<i>b.setBalance(balance * 1.1)</i>
<i>a.withdraw(balance / 10)</i>		<i>c.withdraw(balance / 10)</i>
 	<i>balance = b.getBalance()</i> \$200	
 <i>b.setBalance(balance * 1.1)</i> \$220		 <i>balance = b.getBalance()</i> \$220
 <i>a.withdraw(balance / 10)</i> \$80		 <i>b.setBalance(balance * 1.1)</i> \$242
		<i>c.withdraw(balance / 10)</i> \$278

Figura 16.7 Uma interposição equivalente em série de *T* e *U*.

banco. Os saldos das duas contas bancárias, *A* e *B*, são inicialmente iguais a \$200. O resultado de *branchTotal* inclui a soma de *A* e *B* como \$300, o que está errado. Essa é uma ilustração do problema das recuperações inconsistentes. As recuperações de *W* são inconsistentes porque *V* realizou apenas a parte do saque de uma transferência no momento em que a soma foi calculada.

Equivalência serial • Se for sabido que cada uma de várias transações tem o efeito correto quando executada sozinha, podemos inferir que, se essas transações forem executadas uma por vez, em alguma ordem, o efeito combinado também será correto. Uma interposição das operações das transações em que o efeito combinado é igual ao que seria se as transações tivessem sido executadas uma por vez, em alguma ordem, é uma interposição *serialmente equivalente*. Quando dizemos que duas transações diferentes têm o *mesmo efeito*, queremos dizer que as operações de leitura retornam os mesmos valores e que as variáveis de instância dos objetos têm os mesmos valores no final.

O uso da equivalência serial como critério para uma execução concorrente correta evita a ocorrência de atualizações perdidas e recuperações inconsistentes.

O problema da atualização perdida ocorre quando duas transações leem o valor antigo de uma variável e depois o utilizam para calcular o novo valor. Isso não pode acontecer se uma transação for executada antes da outra, pois a transação posterior lerá o valor escrito pela anterior. Como a interposição serialmente equivalente de duas transações produz o mesmo efeito de uma serial, podemos resolver o problema da atualização perdida por meio da equivalência serial. A Figura 16.7 mostra uma dessas interposições, na qual as operações que afetam a conta compartilhada *B* na verdade são seriais, pois a transação *T* executa todas as duas operações sobre *B* antes que a transação *U* o faça. Outra interposição de *T* e *U* que tem essa propriedade é aquela em que a transação *U* conclui suas operações sobre a conta *B* antes que a transação *T* comece.

Consideraremos agora o efeito da equivalência serial em relação ao problema das recuperações inconsistentes, no qual a transação *V* está transferindo uma quantia da conta *A* para *B*, e a transação *W* está obtendo a soma de todos os saldos (veja a Figura 16.6). O problema das recuperações inconsistentes pode ocorrer quando uma transação de recuperação é executada concorrentemente com uma transação de atualização. Isso não ocorre se a transação de recuperação for executada antes ou depois da transação de atualização.

Transação <i>V</i> :	Transação <i>W</i> :	
<i>a.withdraw(100);</i>		<i>aBranch.branchTotal()</i>
<i>b.deposit(100)</i>		
<i>a.withdraw(100);</i>	\$100	
<i>b.deposit(100)</i>	\$300	
		<i>total = a.getBalance()</i> \$100
		<i>total = total + b.getBalance()</i> \$400
		<i>total = total + c.getBalance()</i>
		...

Figura 16.8 Uma interposição equivalente em série de *V* e *W*.

Uma interposição serialmente equivalente de uma transação de recuperação e de uma transação de atualização, como na Figura 16.8, por exemplo, impedirá a ocorrência de recuperações inconsistentes.

Operações conflitantes • Quando dizemos que duas operações estão em conflito, queremos dizer que seus efeitos combinados dependem da ordem em que elas são executadas. Para simplificar as coisas, consideraremos duas operações, *read* e *write*. A operação *read* acessa o valor de um objeto e *write* altera seu valor. O *efeito* de uma operação se refere ao valor de um objeto configurado por uma operação *write* e ao resultado retornado por uma operação *read*. As regras de conflito para as operações de *leitura* e *escrita* aparecem na Figura 16.9.

Para quaisquer duas transações, é possível determinar a ordem dos pares de operações conflitantes nos objetos acessados por ambas. A equivalência serial pode ser definida em termos de conflitos de operação, como segue:

Para que duas transações sejam *serialmente equivalentes*, é necessário e suficiente que todos os pares de operações conflitantes das duas transações sejam executados na mesma ordem em todos os objetos que ambas acessam.

<i>Operações de diferentes transações</i>		<i>Conflito</i>	<i>Motivo</i>
<i>leitura</i>	<i>leitura</i>	Não	Porque o efeito de duas operações de leitura não depende da ordem em que elas são executadas
<i>leitura</i>	<i>escrita</i>	Sim	Porque o efeito de uma operação de leitura e de uma operação de escrita depende da ordem de sua execução
<i>escrita</i>	<i>escrita</i>	Sim	Porque o efeito de duas operações de escrita depende da ordem de sua execução

Figura 16.9 Regras de conflito das operações de *leitura* e *escrita*.

Transação <i>T</i> :	Transação <i>U</i> :
$x = \text{read}(i)$	
$\text{write}(i, 10)$	
	$y = \text{read}(j)$
	$\text{write}(j, 30)$
$\text{write}(j, 20)$	
	$z = \text{read}(i)$

Figura 16.10 Uma interposição não serialmente equivalente de operações das transações *T* e *U*.

Considere como exemplo as transações *T* e *U*, definidas como segue:

T: $x = \text{read}(i); \text{write}(i, 10); \text{write}(j, 20);$
U: $y = \text{read}(j); \text{write}(j, 30); z = \text{read}(i);$

Então, considere a interposição de suas execuções, mostrada na Figura 16.10. Note que o acesso de cada transação aos objetos *i* e *j* se dá em série com relação um ao outro, pois *T* faz todos os seus acessos a *i* antes que *U* o faça, e *U* faz todos os seus acessos a *j* antes que *T* o faça. No entanto, a ordem não é serialmente equivalente, pois os pares de operações conflitantes não são executados na mesma ordem nos dois objetos. As ordens equivalentes em série exigem uma das duas condições a seguir:

1. *T* acessa *i* antes de *U* e *T* acessa *j* antes de *U*.
2. *U* acessa *i* antes de *T* e *U* acessa *j* antes de *T*.

A equivalência serial é usada como critério para a obtenção de protocolos de controle de concorrência. Esses protocolos tentam dispor as transações em série no acesso aos objetos. Três estratégias alternativas de controle de concorrência são comumente usadas: travamento, controle de concorrência otimista e ordenação de carimbo de tempo (*timestamp*). Entretanto, a maioria dos sistemas práticos usa o travamento, o qual será discutido na Seção 16.4. Quando o travamento é usado, o servidor configura uma trava rotulada com o identificador de transação em cada objeto, imediatamente antes que ele seja acessado, e remove essas travas quando a transação é concluída. Enquanto um objeto está travado, somente a transação para a qual ele está travado pode acessar esse objeto; as outras transações devem esperar até que o objeto seja destravado ou, em alguns casos, compartilhar seu uso. O uso de travamentos pode levar ao impasse (*deadlock*), com transações esperando umas às outras para liberar as travas; como, por exemplo, quando cada uma de duas transações tem um objeto travado que a outra precisa acessar. Discutiremos o problema do impasse, e algumas soluções, na Seção 16.4.1.

O controle de concorrência otimista será descrito na Seção 16.5. Nos esquemas otimistas, uma transação prossegue até que peça para ser confirmada. Antes que seja permitida a confirmação, o servidor realiza uma verificação para descobrir se ela efetuou operações sobre quaisquer objetos que entrem em conflito com as operações de outras transações concorrentes, caso em que o servidor a cancela e o cliente pode reiniciá-la. O objetivo da verificação é garantir que todos os objetos estejam corretos.

A ordenação por carimbo de tempo será descrita na Seção 16.6. Nela, um servidor registra o tempo da leitura e escrita mais recentes de cada objeto e operação, e o carimbo

Transação <i>T</i> :	Transação <i>U</i> :
<i>a.getBalance()</i>	<i>a.getBalance()</i>
<i>a.setBalance(balance + 10)</i>	<i>a.setBalance(balance + 20)</i>
<i>balance = a.getBalance()</i>	\$100
<i>a.setBalance(balance + 10)</i>	\$110
	<i>balance = a.getBalance()</i> \$110
	<i>a.setBalance(balance + 20)</i> \$130
	<i>commit transaction</i>
	<i>abort transaction</i>

Figura 16.11 Uma leitura suja quando a transação *T* é cancelada.

de tempo da transação é comparado com o do objeto para determinar se ela será realizada imediatamente, retardada ou rejeitada. Quando uma operação é retardada, a transação espera; quando ela é rejeitada, a transação é cancelada.

Basicamente, o controle de concorrência pode ser obtido pelas transações dos clientes esperando umas às outras, pelo reinício de transações, após um conflito entre as operações ter sido detectado, ou por uma combinação dos dois.

16.2.2 Capacidade de recuperação de cancelamentos

Os servidores devem registrar os efeitos de todas as transações confirmadas e nenhum dos efeitos das transações canceladas. Portanto, eles devem se preparar para o fato de que uma transação pode ser cancelada, impedindo-a de afetar outras transações concorrentes, caso isso aconteça.

Esta seção ilustra dois problemas associados ao cancelamento de transações no contexto do exemplo de transações bancárias. Esses problemas são chamados de *leituras sujas* e *escritas prematuras*, e ambos podem ocorrer na presença de execuções serialmente equivalentes de transações. Esses problemas estão relacionados com os efeitos de operações sobre objetos, como o saldo de uma conta bancária. Para simplificar as coisas, as operações serão consideradas em duas categorias: operações de *leitura* e operações de *escrita*. Em nossas ilustrações, *getBalance* é uma operação de *leitura* e *setBalance* é uma operação de *escrita*.

Leituras sujas (dirty read) • A propriedade do isolamento das transações exige que elas não vejam o estado não confirmado de outras transações. O problema da leitura suja é causado pela interação entre uma operação de leitura em uma transação e uma operação de escrita anterior em outra transação, sobre o mesmo objeto. Considere as execuções ilustradas na Figura 16.11, nas quais *T* recebe o saldo da conta *A* e a configura com mais \$10; em seguida, *U* recebe o saldo da conta *A* e a configura com mais \$20, e as duas execuções são serialmente equivalentes. Agora, suponha que a transação *T* seja cancelada após *U* ter sido confirmada. Então, a transação *U* terá visto um valor que nunca existiu, pois *A* será restaurada em seu valor original. Nesse caso, dizemos que a transação *U* realizou uma leitura suja. Como ela foi confirmada, não pode ser desfeita.

Transação <i>T</i>:	Transação <i>U</i>:
<i>a.setBalance(105)</i>	<i>a.setBalance(110)</i>
\$100	
<i>a.setBalance(105)</i>	\$105
	<i>a.setBalance(110)</i>
	\$110

Figura 16.12 Sobreposição de valores não confirmados.

Capacidade de recuperação das transações • Se uma transação (como *U*) é confirmada após ter visto os efeitos de uma transação que é subsequentemente cancelada, a situação é não recuperável. Para garantir que tais situações não surjam, toda transação (como *U*), que corre o risco de uma leitura suja, retarda sua operação de confirmação. A estratégia para a capacidade de recuperação é retardar as confirmações para depois da confirmação de qualquer outra transação cujo estado não confirmado tenha sido observado. Em nosso exemplo, *U* retarda sua confirmação para depois que *T* é confirmada. No caso de *T* ser cancelada, *U* também deve ser cancelada.

Cancelamentos em cascata • Na Figura 16.11, suponha que a transação *U* retarde a confirmação para depois que *T* for cancelada. Conforme dissemos, *U* também deve ser cancelada. Infelizmente, se quaisquer outras transações tiverem visto os efeitos causados por *U*, elas também deverão ser canceladas. O cancelamento destas últimas transações pode fazer com que ainda mais transações sejam canceladas. Tais situações são chamadas de *cancelamentos em cascata*. Para evitar os cancelamentos em cascata, as transações só podem ler objetos que foram escritos por transações confirmadas. Para garantir que isso ocorra, toda operação de leitura deve ser retardada até que as outras transações que fizeram uma operação de escrita sobre o mesmo objeto tenham sido confirmadas ou canceladas. O fato de evitar cancelamentos em cascata é uma condição mais importante do que a capacidade de recuperação.

Escritas prematuras • Considere outra implicação da possibilidade de uma transação ser cancelada. Esta relacionada à interação entre operações de *escrita* no mesmo objeto, pertencentes a diferentes transações. Como ilustração, consideremos duas transações *setBalance*, *T* e *U*, na conta *A*, como se vê na Figura 16.12. Antes das transações, o saldo da conta *A* era de \$100. As duas execuções são serialmente equivalentes, com *T* determinando o saldo como \$105 e *U* determinando como \$110. Se a transação *U* for cancelada e *T* for confirmada, o saldo deverá ser de \$105.

Alguns sistemas de banco de dados implementam a ação de *cancelamento* restaurando “imagens de antes” de todas as *escritas* de uma transação. Em nosso exemplo, *A* tem inicialmente \$100, que é a “imagem de antes” da *escrita* de *T*; de modo semelhante \$105 é a “imagem de antes” da *escrita* de *U*. Assim, se *U* for cancelada, obteremos o saldo correto de \$105.

Agora, considere o caso de quando *U* é confirmada e depois *T* é cancelada. O saldo deve ser de \$110, mas como a “imagem de antes” da *escrita* de *T* é \$100, obtemos o saldo errado de \$100. Analogamente, se *T* for cancelada, e depois *U* for cancelada, a “imagem de antes” da *escrita* de *U* será \$105 e obteremos o saldo errado de \$105 – o saldo deve voltar para \$100.

Para garantir resultados corretos em um esquema de recuperação que utiliza imagens de antes, as operações de *escritas* devem ser retardadas até que as transações anteriores que atualizaram os mesmos objetos tenham sido confirmadas ou canceladas.

Execuções restritas de transações • Geralmente, é exigido que as transações retardem tanto as operações de *leitura* como de *escrita*, para evitar leituras sujas e escritas prematuras. As execuções das transações são chamadas de *restritas* se o serviço atrasar tanto as operações de *leitura* como de *escrita* sobre um objeto até que todas as transações que escreveram nesse objeto anteriormente tenham sido confirmadas ou canceladas. A execução restrita de transações impõe a desejada propriedade do isolamento.

Versões de tentativa • Para que um servidor de objetos recuperáveis participe nas transações, ele deve ser projetado de modo que todas as atualizações de objetos possam ser removidas, se, e quando, uma transação for cancelada. Para tornar isso possível, todas as operações de atualização realizadas durante uma transação são executadas em versões de tentativa dos objetos, na memória volátil. Cada transação recebe seu próprio conjunto privado de versões de tentativa de todos os objetos que tiver alterado. Todas as operações de atualização de uma transação armazenam valores no conjunto privado próprio da transação. Se possível, as operações de acesso em uma transação pegam valores do conjunto privado da própria transação ou, falhando isso, dos objetos.

As versões de tentativa são transferidas para os objetos somente quando uma transação é confirmada, quando então elas também são escritas no meio de armazenamento permanente. Isso é feito em uma única etapa, durante a qual as outras transações são excluídas do acesso aos objetos que estão sendo alterados. Quando uma transação é cancelada, suas versões de tentativa são excluídas.

16.3 Transações aninhadas

As transações aninhadas ampliam o modelo de transação anterior, permitindo que as transações sejam compostas de outras transações. Assim, várias transações podem ser iniciadas dentro de uma transação, possibilitando que as transações sejam consideradas módulos que podem ser compostos conforme for exigido.

A transação mais externa em um conjunto de transações aninhadas é chamada de transação de *nível superior*. As outras transações são chamadas de *subtransações*. Por exemplo, na Figura 16.13, T é uma transação de nível superior, a qual inicia duas subtransações T_1 e T_2 . A subtransação T_1 inicia seu próprio par de subtransações T_{11} e T_{12} . Além disso, a subtransação T_2 inicia sua própria subtransação T_{21} , que inicia outra subtransação T_{211} .

Uma subtransação aparece como atômica para sua ascendente com relação às faihas de transação e ao acesso concorrente. As subtransações que estão no mesmo nível, como T_1 e T_2 , podem ser executadas concorrentemente, mas seu acesso a objetos comuns é organizado em série, por exemplo, pelo esquema de travamento descrito na Seção 16.4. Cada subtransação pode falhar independentemente de sua ascendente e das outras subtransações. Quando uma subtransação é cancelada, a transação ascendente às vezes pode escolher uma subtransação alternativa para completar sua tarefa. Por exemplo, uma transação para enviar uma mensagem de correio para uma lista de destinatários poderia ser estruturada como um conjunto de subtransações, cada uma das quais enviando a mensagem para um dos destinatários. Se uma ou mais das subtransações falhar, a transação ascendente poderia registrar o fato, e ser confirmada, com base em que apenas as transações descendentes bem-sucedidas foram confirmadas. Então, ela poderia iniciar outra transação para tentar enviar as mensagens que não foram enviadas na primeira vez.

Quando precisamos distinguir nossa forma de transação original das transações aninhadas, usamos o termo transação *plana*. Ela é plana porque todo o seu trabalho é

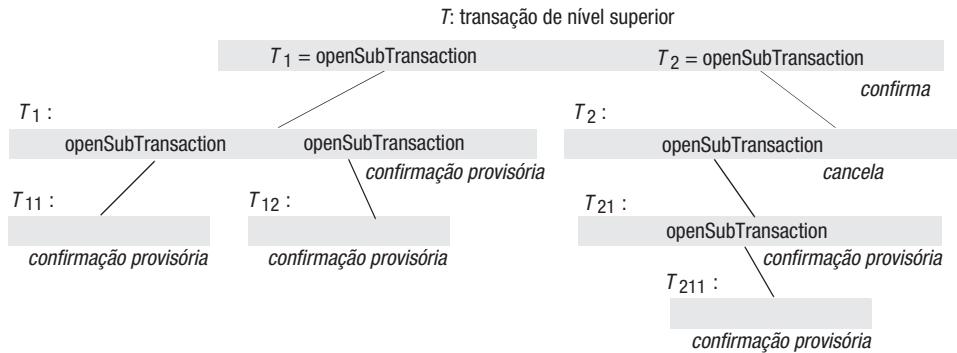


Figura 16.13 Transações aninhadas.

feito no mesmo nível entre uma transação *openTransaction* e uma *confirmação* ou um *cancelamento*, e não é possível efetivar ou cancelar partes dela. As transações aninhadas têm as seguintes vantagens principais:

1. As subtransações de um nível (e suas descendentes) podem ser executadas concurrentemente com outras subtransações de mesmo nível na hierarquia. Isso pode possibilitar um nível maior de concorrência em uma transação. Quando as subtransações são executadas em servidores diferentes, elas podem trabalhar em paralelo. Por exemplo, considere a operação *branchTotal* em nosso exemplo de transações bancárias. Ela pode ser implementada pela invocação de *getBalance* em cada conta da agência. Agora, cada uma dessas invocações pode ser executada como uma subtransação, no caso em que elas podem ser executadas concurrentemente. Como cada operação se aplica a uma conta diferente, não existirão operações conflitantes entre as subtransações.
2. As subtransações podem ser confirmadas ou canceladas independentemente. Em comparação com uma única transação, um conjunto de subtransações aninhadas é potencialmente mais robusto. O exemplo anterior de envio de correspondência mostra que isso é verdade – com uma transação plana, uma falha de transação causaria o reinício da transação inteira. Na verdade, uma transação ascendente pode decidir sobre diferentes ações, de acordo com o fato de uma subtransação ter sido cancelada ou não.

As regras de confirmação das transações aninhadas são bastante refinadas:

- Uma transação só pode ser confirmada ou cancelada depois que suas transações descendentes tiverem sido concluídas.
- Quando uma subtransação é concluída, ela toma uma decisão independente de ser confirmada provisoriamente ou ser cancelada. Sua decisão de cancelar é final.
- Quando uma transação ascendente é cancelada, todas as suas subtransações são canceladas. Por exemplo, se T_2 é cancelada, então T_{21} e T_{211} também devem ser canceladas, mesmo que possam ter sido confirmadas provisoriamente.
- Quando uma subtransação é cancelada, a transação ascendente pode decidir se vai ser cancelada ou não. Em nosso exemplo, T decide ser confirmada, embora T_2 tenha sido cancelada.

- Se a transação de nível superior é confirmada, então todas as subtransações que foram confirmadas provisoriamente também podem ser confirmadas, desde que nenhuma de suas ascendentes tenha sido cancelada. Em nosso exemplo, a confirmação de T permite que T_1 , T_{11} e T_{12} sejam confirmadas, mas não T_{21} e T_{211} , pois sua ascendente T_2 foi cancelada. Note que os efeitos de uma subtransação não são permanentes até a transação de nível superior seja confirmada.

Em alguns casos, a transação de nível superior pode decidir ser cancelada porque uma ou mais de suas subtransações foi cancelada. Como exemplo, considere a seguinte transação *Transferência*:

Transferência de \$100 de B para A
 $a.\text{deposit}(100)$
 $b.\text{withdraw}(100)$

Isso pode ser estruturado como um par de subtransações, uma para a operação *withdraw* e a outra para *deposit*. Quando as duas subtransações forem confirmadas, a transação *Transferência* também poderá ser confirmada. Suponha que uma subtransação *withdraw* seja cancelada quando uma conta não tiver fundos. Agora, considere o caso de quando a subtransação *withdraw* é cancelada e a subtransação *deposit* é confirmada – e lembre-se de que a confirmação de uma transação descendente está condicionada à confirmação da transação ascendente. Presumimos que a transação de nível superior (*Transferência*) decida ser cancelada. O cancelamento da transação ascendente causa o cancelamento das subtransações – de modo que a transação *deposit* é cancelada, e todos os seus efeitos são anulados.

O *Object Transaction Service* do CORBA suporta transações planas e aninhadas. As transações aninhadas são particularmente úteis nos sistemas distribuídos, pois as transações descendentes podem ser executadas concorrentemente em diferentes servidores. Voltaremos a essa questão no Capítulo 17. Essa forma de transações aninhadas é atribuída a Moss [1985]. Foram propostas outras variantes de transações aninhadas, com diferentes propriedades de disposição em série; para exemplos, consulte Weikum [1991].

16.4 Travas

As transações devem ser programadas de modo que seus efeitos sobre os dados compartilhados sejam serialmente equivalentes. Um servidor pode obter equivalência serial das transações, dispondo em série o acesso aos objetos. A Figura 16.7 mostra um exemplo de como a equivalência serial pode ser obtida com certo grau de concorrência – as transações T e U acessam a conta B , mas T conclui seu acesso antes que U comece a acessá-la.

Um exemplo simples de mecanismo para disposição em série é o uso de travas exclusivas (*exclusive locks*). Nesse esquema, o servidor tenta impedir o acesso (travar) a qualquer objeto que esteja para ser usado por qualquer operação da transação de um cliente. Se um cliente solicitar o acesso a um objeto que já está travado devido à transação de outro cliente, a requisição será suspensa e o cliente deverá esperar até que o objeto seja destravado.

A Figura 16.14 ilustra o uso de travas exclusivas. Ela mostra as mesmas transações da Figura 16.7, mas com uma coluna extra para cada transação, mostrando o travamento, a espera e o destravamento. Nesse exemplo, supõe-se que, quando as transações T e U começam, os saldos das contas A , B e C ainda não estão travados. Quando a transação T está para usar a conta B , ela é travada por T . Subsequentemente, quando a transação U

Transação <i>T</i> :		Transação <i>U</i> :	
Operações	Travas	Operações	Travas
<i>balance = b.getBalance()</i>		<i>balance = b.getBalance()</i>	
<i>b.setBalance(bal*1.1)</i>		<i>b.setBalance(bal*1.1)</i>	
<i>a.withdraw(bal/10)</i>		<i>c.withdraw(bal/10)</i>	
Operações	Travas	Operações	Travas
<i>openTransaction</i>			
<i>bal = b.getBalance()</i>	trava <i>B</i>		
<i>b.setBalance(bal*1.1)</i>		<i>openTransaction</i>	
<i>a.withdraw(bal/10)</i>	trava <i>A</i>	<i>bal = b.getBalance()</i>	espera por <i>T</i> travado em <i>B</i>
<i>closeTransaction</i>	trava <i>A</i> , <i>B</i>	• • •	trava <i>B</i>
		<i>b.setBalance(bal*1.1)</i>	
		<i>c.withdraw(bal/10)</i>	trava <i>C</i>
		<i>closeTransaction</i>	destrava <i>B</i> , <i>C</i>

Figura 16.14 Transações *T* e *U* com travas exclusivas.

está para usar *B*, ela ainda está travada por *T* e a transação *U* espera. Quando a transação *T* é confirmada, *B* é destravada, em consequência disso a transação *U* é retomada. O uso de trava em *B* efetivamente organiza em série o acesso a *B*. Note que se, por exemplo, *T* tivesse liberado a trava sobre *B*, entre suas operações *getBalance* e *setBalance*, a operação *getBalance* da transação *U* poderia ser interposta entre elas.

A equivalência serial exige que todos os acessos de uma transação a um objeto em particular sejam dispostos em série com relação aos acessos feitos por outras transações. Todos os pares de operações conflitantes de duas transações devem ser executados na mesma ordem. Para garantir isso, uma transação não pode solicitar novas travas após ter liberado uma. A primeira fase de cada transação é uma fase de crescimento, durante a qual as novas travas são adquiridas. Na segunda fase, as travas são liberadas (uma fase de redução). Esse processo é chamado de *travamento de duas fases* (*two-phase locking*).

Vimos na Seção 16.2.2 que, como as transações podem ser canceladas, são necessárias execuções restritas para evitar leituras sujas e escritas prematuras. Sob um regime de execução restrita, uma transação que precise ler ou escrever um objeto deve ser retardada até que outras transações que escreveram o mesmo objeto tenham sido confirmadas ou canceladas. Para impor essa regra, todas as travas aplicadas durante o andamento de uma transação são mantidas, até que a transação seja confirmada ou cancelada. Esse processo é chamado de *travamento de duas fases restrito*. A presença do travamento impede que outras transações leiam ou escrevam os objetos. Quando uma transação é confirmada, para garantir a capacidade de recuperação, os travamentos devem ser mantidos até que todos os objetos que ela atualizou tenham sido escritos no armazenamento permanente.

Geralmente, um servidor contém um grande número de objetos e uma transação típica acessa apenas alguns deles, sendo improvável um conflito com outras transações concorrentes. Uma questão importante é a *granularidade* com que o controle de con-

corrência pode ser aplicado nos objetos, pois a abrangência do acesso concorrente aos objetos em um servidor será seriamente limitada, caso o controle de concorrência (por exemplo, travas) só possa ser aplicado a todos os objetos simultaneamente. Em nosso exemplo de transações bancárias, se uma só trava fosse aplicada a todas as contas de uma agência, somente um funcionário do banco poderia realizar uma transação *online* em dado momento – dificilmente essa seria uma restrição aceitável!

A parte dos objetos à qual o acesso deve ser organizado em série deve ser a menor possível; isto é, apenas a parte envolvida em cada operação solicitada pelas transações. Em nosso exemplo, uma agência contém um conjunto de contas, cada uma das quais com um saldo. Cada operação bancária afeta um ou mais saldos de conta – *deposit* e *withdraw* afetam o saldo de uma conta, e *branchTotal* afeta todos eles.

A descrição dos esquemas de controle de concorrência dada a seguir não presume nenhuma granularidade em particular. Discutiremos os protocolos de controle de concorrência aplicáveis aos objetos cujas operações podem ser modeladas em termos das operações de *leitura* e de *escrita* nos objetos. Para que os protocolos funcionem corretamente, é fundamental que cada operação de *leitura* e de *escrita* seja atômica em seus efeitos nos objetos.

Os protocolos de controle de concorrência são projetados para suportar *conflitos* entre operações de diferentes transações sobre o mesmo objeto. Neste capítulo, usamos a noção de conflito entre operações para explicar os protocolos. As regras de conflito das operações de *leitura* e de *escrita* aparecem na Figura 16.9, a qual mostra que pares de operações de *leitura* de diferentes transações sobre o mesmo objeto não entram em conflito. Portanto, uma trava exclusiva simples, usada tanto para operações de *leitura* como de *escrita*, reduz a concorrência mais do que o necessário.

É preferível adotar um esquema de travamento que controle o acesso a cada objeto para que possam existir várias transações concorrentes lendo um objeto, ou uma única transação escrevendo um objeto, mas não ambas. Isso é normalmente referido como esquema de muitos leitores/um escritor. São usados dois tipos de travas: *travas de leitura* e *travas de escrita*. Antes que a operação de *leitura* de uma transação seja executada, deve ser alocada uma trava de leitura no objeto. Antes que a operação de *escrita* de uma transação seja executada, deve ser alocada uma trava de escrita no objeto. Quando for impossível alocar uma trava imediatamente, a transação (e o cliente) deverão esperar até que isso seja possível – a requisição de um cliente nunca é rejeitada.

Duas operações de *leitura* de diferentes transações não entram em conflito; uma tentativa de alocar uma operação de leitura em um objeto é sempre bem-sucedida. Todas as transações que leem o mesmo objeto compartilham a trava de leitura – por isso, as travas de leitura às vezes são chamadas de *travas compartilhadas*.

As regras de conflito de operação nos dizem que:

1. Se uma transação *T* já executou uma operação de *leitura* sobre um objeto em particular, então uma transação concorrente *U* não deve *escrever* esse objeto até que *T* seja confirmada ou cancelada.
2. Se uma transação *T* já executou uma operação de *escrita* sobre um objeto em particular, então uma transação concorrente *U* não deve *ler* nem *escrever* esse objeto até que *T* seja confirmada ou cancelada.

Para impor a condição (1), uma requisição para uma trava de escrita sobre um objeto é atrasada pela presença de uma trava de leitura pertencente à outra transação. Para impor (2), uma requisição para uma trava de leitura ou para uma trava de escrita sobre um objeto é atrasada pela presença de uma trava de escrita pertencente à outra transação.

<i>Para um objeto</i>		<i>Trava solicitada</i>	
		<i>leitura</i>	<i>escrita</i>
<i>Trava já alocada</i>	<i>nenhum</i>	OK	OK
	<i>leitura</i>	OK	espera
	<i>escrita</i>	espera	espera

Figura 16.15 Compatibilidade de travas.

A Figura 16.15 mostra a compatibilidade das travas de leitura e de escrita sobre qualquer objeto específico. As entradas à esquerda da coluna *Trava solicitada* mostram o tipo de trava já alocada, se houver. As entradas acima da primeira linha mostram o tipo de trava solicitada. Cada entrada na tabela mostra o efeito sobre uma transação que solicita o tipo de trava dado acima, quando o objeto tiver sido travado em outra transação com o tipo de trava da esquerda.

As recuperações inconsistentes e atualizações perdidas são causadas pelo conflito entre operações de *leitura* em uma transação e operações de *escrita* em outra, sem a proteção de um esquema de controle de concorrência como o travamento. As recuperações inconsistentes são evitadas executando-se a transação de recuperação antes ou depois da transação de atualização. Se a transação de recuperação vier primeiro, suas travas de leitura atrasarão a transação de atualização. Se ela vier depois, sua requisição de trava de leitura a fará ser retardada até que a transação de atualização tenha terminado.

Atualizações perdidas ocorrem quando duas transações leem um valor de um objeto e depois o utilizam para calcular um novo valor. As atualizações perdidas são evitadas fazendo-se transações posteriores retardarem suas leituras até que as anteriores tenham terminado. Isso é conseguido em cada transação configurando uma trava de leitura ao ler um objeto e depois *promovendo-o* a uma trava de escrita, ao escrever o mesmo objeto – quando uma transação subsequente exigir uma trava de leitura, ela será retardada até que a transação corrente tenha terminado.

Uma transação com uma trava de leitura compartilhada com outras transações não pode promover sua trava de leitura para um trava de escrita, pois esta última entraria em conflito com as travas de leitura mantidas pelas outras transações. Portanto, tal transação deve solicitar uma trava de escrita e esperar que as outras travas de leitura sejam liberadas.

A promoção de travas se refere à conversão de uma trava em uma outra mais forte – isto é, em uma trava mais exclusiva. A tabela de compatibilidade de travas mostra quais delas são mais ou menos exclusivas. A trava de leitura permite outras travas de leitura, enquanto a trava de escrita, não. E também não permite outras travas de escrita. Portanto, uma trava de escrita é mais exclusiva do que uma trava de leitura. As travas podem ser promovidas porque o resultado é uma trava mais exclusiva. Não é seguro rebaixar uma trava mantida por uma transação antes que ela seja confirmada, pois o resultado será mais permissivo do que o anterior e pode possibilitar execuções de outras transações que sejam inconsistentes com a equivalência serial.

As regras para o uso de travas em uma implementação do travamento de duas fases restrito estão resumidas na Figura 16.16. Para garantir que essas regras sejam obedecidas, o cliente não tem acesso às operações de travamento ou destravamento de itens de dados. O travamento é realizado quando os pedidos de operações de *leitura* e de *escrita* estão para ser aplicados aos objetos recuperáveis e o destravamento é realizado pelas operações de *confirmação* ou *cancelamento* do coordenador de transação.

1. Quando uma operação acessa um objeto dentro de uma transação:
 - (a) Se o objeto ainda não estiver travado, ele será travado e a operação prosseguirá.
 - (b) Se o objeto tiver uma trava conflitando com a configurada por outra transação, a transação deverá esperar até que ela seja liberada.
 - (c) Se o objeto tiver uma trava não conflitante com a configurada por outra transação, a trava será compartilhada e a operação prosseguirá.
 - (d) Se o objeto já tiver sido travado na mesma transação, a trava será promovida, se necessário, e a operação prosseguirá. (Onde a promoção é impedida por uma trava conflitante, será usada a regra (b).)
2. Quando uma transação é confirmada ou cancelada, o servidor destrava todos os objetos que travou para a transação.

Figura 16.16 Uso de travas no travamento de duas fases restrito.

Por exemplo, o *Concurrency Control Service* do CORBA [OMG 2000b] pode ser usado para aplicar controle de concorrência em nome de transações ou para proteger objetos sem usar transações. Ele fornece uma maneira de associar uma coleção de travas (chamada de *conjunto de travas*) a um recurso como um objeto recuperável. Um conjunto de travas permite que as travas sejam adquiridas ou liberadas. O método *lock* de um conjunto de travas adquirirá uma trava, ou um bloco, até que a trava seja liberada; outros métodos permitem que as travas sejam promovidas ou liberadas. Os conjuntos de travas transacionais suportam os mesmos métodos que os conjuntos de travas, mas seus métodos exigem identificadores de transação como argumentos. Mencionamos anteriormente que o serviço de transação CORBA rotula todas as requisições de cliente de uma transação com o identificador de transação. Isso permite que uma trava necessária seja adquirida, antes que cada um dos objetos recuperáveis seja acessado durante uma transação. O coordenador de transação é responsável por liberar as travas quando uma transação é confirmada ou cancelada.

As regras dadas na Figura 16.16 garantem o rigor, pois as travas são mantidas até que uma transação tenha sido confirmada ou cancelada. Entretanto, não é necessário manter travas de leitura para garantir o rigor. As travas de leitura só precisam ser mantidas até que chegue a requisição para confirmar ou cancelar.

Implementação de travas • A atribuição de travas será implementada por um objeto separado no servidor, que chamamos de *gerenciador de travas*. O gerenciador de travas mantém um conjunto de travas, por exemplo, em uma tabela de *hashing*. Cada trava é uma instância da classe *Lock* e é associada a um objeto em particular. A classe *Lock* aparece na Figura 16.17. Cada instância de *Lock* mantém as seguintes informações em suas variáveis de instância:

- o identificador do objeto travado;
- os identificadores das transações que correntemente mantêm as travas (as travas compartilhadas podem ter vários proprietários);
- um tipo de trava.

Os métodos de *Lock* são sincronizados para que as *threads* que estão tentando adquirir ou liberar uma trava não interfiram umas nas outras. Contudo, além disso, tentativas de adquirir uma trava usam o método *wait* quando precisam esperar que outra *thread* a libere.

```

public class Lock {
    private Object object;      // o objeto que está sendo protegido pela trava
    private Vector holders;     // os TIDs dos proprietários correntes
    private LockType lockType;  // o tipo corrente

    public synchronized void acquire(TransID trans, LockType aLockType) {
        while( /*outra transação possui a trava em modo conflitante*/ ) {
            try {
                wait();
            } catch ( InterruptedException e){/*...*/}
        }
        if (holders.isEmpty( )) { // nenhum TID mantém travas
            holders.addElement(trans);
            lockType = aLockType;
        } else if ( /*outra transação possui a trava e a compartilha*/ ) {
            if ( /* esta transação não é proprietária*/ ) holders.addElement(trans);
        } else if (/* esta transação é proprietária, mas precisa de uma trava mais exclusiva*/ )
            lockType.promote();
        }
    }

    public synchronized void release(TransID trans) {
        holders.removeElement(trans); // remove este proprietário
        // configura o tipo de trava como nenhum
        notifyAll();
    }
}

```

Figura 16.17 Classe Lock.

O método *acquire* executa as regras dadas na Figura 16.15 e na Figura 16.16. Seus argumentos especificam um identificador de transação e o tipo de trava exigido por essa transação. Ele testa se a requisição pode ser atendida. Se outra transação possuir a trava em um modo conflitante, ela ativa *wait*, o que faz a *thread* do chamador ser suspensa até um método *notify* correspondente. Note que o método *wait* é englobado em uma instrução *while*, pois todas as transações que estão esperando são notificadas, e algumas delas podem não ser capazes de prosseguir. Quando finalmente a condição é satisfeita, o restante do método configura a trava apropriadamente:

- se nenhuma outra transação possui a trava, apenas adiciona a transação dada no grupo de proprietários e configura o tipo;
- se outra transação possui a trava, a compartilha, adicionando a transação dada no grupo de proprietários (a não ser que ela já seja proprietária);
- se essa transação é proprietária, mas está solicitando uma trava mais exclusiva, promove a trava.

Os argumentos do método *release* especificam o identificador da transação que está liberando a trava. Ele remove o identificador de transação dos proprietários, configura o tipo

```
public class LockManager {  
    private Hashtable theLocks;  
  
    public void setLock(Object object, TransID trans, LockType lockType){  
        Lock foundLock;  
        synchronized(this){  
            // localiza a trava associada ao objeto  
            // se não houver nenhuma, cria e a adiciona na tabela de hashing  
            }  
            foundLock.acquire(trans, lockType);  
        }  
  
        // sincroniza este, pois queremos remover todas as entradas  
        public synchronized void unLock(TransID trans) {  
            Enumeration e = theLocks.elements();  
            while(e.hasMoreElements()){  
                Lock aLock = (Lock)e.nextElement();  
                if(/* trans é um proprietário desta trava*/) aLock.release(trans);  
            }  
        }  
    }  
}
```

Figura 16.18 Classe *LockManager*.

de trava como *nenhum* e chama o método *notifyAll*. O método notifica todas as *threads* que estão esperando, para o caso de haver várias transações esperando para adquirir travas de leitura, situação na qual todas elas podem ser capazes de prosseguir.

A classe *LockManager* aparece na Figura 16.18. Todas as requisições para configurar travas e para liberá-las em nome de transações são enviadas para uma instância de *LockManager*.

- Os argumentos do método *setLock* especificam o objeto que a transação dada deseja travar e o tipo de trava. Ele encontra uma trava para esse objeto em sua tabela de *hashing* ou, se necessário, cria uma. Então, ele invoca o método *acquire* dessa trava.
- O argumento do método *unLock* especifica a transação que está liberando suas travas. Ele encontra todas as travas na tabela de *hashing* que tem como proprietária a transação dada. Para cada uma, ele chama o método *release*.

Algumas questões de política: note que, quando várias *threads* esperam no mesmo item travado, a semântica de *wait* garante que cada transação tenha sua vez. No programa anterior, as regras de conflito permitem que os proprietários de uma trava sejam vários leitores ou um escritor. A chegada de uma requisição de trava de leitura é sempre atendida, a não ser que o proprietário tenha uma trava de escrita. Você, leitor, é convidado a considerar o seguinte:

Qual é a consequência das transações de *escrita* na presença de um fluxo constante de travas de leitura? Pense em uma implementação alternativa.

Quando o proprietário tem uma trava de escrita, vários leitores e escritores podem estar esperando. Você, leitor, deve considerar o efeito de *notifyAll* e pensar em uma implementação alternativa. Se o proprietário de uma trava de leitura que está sendo compartilhada tentar promovê-la, ele será impedido. Existe uma solução para essa dificuldade?

Regras de travamento para transações aninhadas • O objetivo de um esquema de travamento para transações aninhadas é dispor em série o acesso aos objetos, para que:

1. cada conjunto de transações aninhadas seja uma entidade única que deve ser impedida de observar os efeitos parciais de qualquer outro conjunto de transações aninhadas;
2. cada transação dentro de um conjunto de transações aninhadas deve ser impedida de observar os efeitos parciais das outras transações do conjunto.

A primeira regra é imposta fazendo-se com que toda trava adquirida por uma subtransação bem-sucedida seja *herdada* por sua ascendente, quando for concluída. As travas herdadas também são herdadas pelas ancestrais. Note que essa forma de herança passa da descendente para a ascendente! A transação de nível superior finalmente herda todas as travas que foram adquiridas por subtransações bem-sucedidas, em qualquer profundidade de uma transação aninhada. Isso garante que as travas possam ser mantidas até que a transação de nível superior tenha sido confirmada ou cancelada, o que impede que membros de diferentes conjuntos de transações aninhadas observem os efeitos parciais uns dos outros.

A segunda regra é imposta como segue:

- as transações ascendentes não podem ser executadas concorrentemente com suas transações descendentes. Se uma transação ascendente tiver uma trava sobre um objeto, ela manterá a trava durante o período em que sua transação descendente estiver executando. Isso significa que a transação descendente adquire temporariamente a trava de sua ascendente, pelo tempo de sua duração;
- as subtransações no mesmo nível podem ser executadas concorrentemente; portanto, quando elas acessam os mesmos objetos, o esquema de travamento deve organizar seus acessos em série.

As regras a seguir descrevem a aquisição e a liberação de travas:

- para que uma subtransação adquira uma trava de leitura sobre um objeto, nenhuma outra transação ativa pode ter uma trava de escrita sobre esse objeto, e as únicas detentoras de uma trava de escrita são suas ancestrais;
- para que uma subtransação adquira uma trava de escrita sobre um objeto, nenhuma outra transação ativa pode ter uma trava de leitura ou escrita sobre esse objeto, e as únicas detentoras de travas de leitura e escrita sobre esse objeto são suas ancestrais;
- quando uma subtransação é confirmada, suas travas são herdadas por sua ascendente, permitindo que esta mantenha as travas no mesmo modo que a descendente;
- quando uma subtransação é cancelada, suas travas são descartadas. Se a ascendente já possui as travas, ela pode continuar a mantê-las.

Note que as subtransações no mesmo nível, que acessam o mesmo objeto, terão sua oportunidade de adquirir as travas mantidas por suas ascendentes. Isso garante que os acessos a um objeto comum sejam organizados em série.

Transação <i>T</i>		Transação <i>U</i>	
Operações	Travas	Operações	Travas
<i>a.deposit(100);</i>	trava de escrita em A	<i>b.deposit(200)</i>	trava de escrita em B
<i>b.withdraw(100)</i>			
...	espera pela trava de <i>U</i> em <i>B</i>	<i>a.withdraw(200);</i>	espera pela trava de <i>T</i> em A
...		...	
...		...	

Figura 16.19 Impasse com travas de escrita.

Como exemplo, suponha que as subtransações T_1 , T_2 e T_{11} , na Figura 16.13, acessem um objeto comum, o qual não é acessado pela transação de nível superior T . Suponha que a subtransação T_1 seja a primeira a acessar o objeto, e que tenha êxito ao adquirir uma trava, o qual ela passa para T_{11} pelo tempo da duração de sua execução, obtendo-a de volta quando T_{11} termina. Quando T_1 conclui sua execução, a transação de nível superior T herda a trava, a qual a mantém até que o conjunto de transações aninhadas termine. A subtransação T_2 pode adquirir a trava de T pelo tempo da duração de sua execução.

16.4.1 Impasses

O uso de travas pode levar a impasses (*deadlocks*). Considere o uso de travas mostrado na Figura 16.19. Como os métodos *deposit* e *withdraw* são atômicos, foram mostrados adquirindo travas de escrita – embora, na prática, eles leiam o saldo e depois o escrevam. Cada um deles adquire uma trava sobre uma conta e, depois, ao tentar acessar a conta que o outro bloqueou, é impedido. Esta é uma situação de impasse – duas transações estão esperando e cada uma depende da outra liberar uma trava para que possa ser retomada.

O impasse é uma situação particularmente comum quando os clientes estão envolvidos em um programa interativo, pois, em um programa interativo, uma transação pode durar um longo período de tempo, resultando em muitos objetos sendo travados e permanecendo assim, impedindo com isso que outros clientes os utilizem.

Note que o travamento de subitens em objetos estruturados pode ser útil para evitar conflitos e possíveis situações de impasse. Por exemplo, em uma agenda, um dia poderia ser estruturado como um conjunto de repartições de tempo, cada uma das quais podendo ser travada independentemente da atualização. Os esquemas de travamento hierárquicos são úteis se a aplicação exigir uma granularidade diferente de travamento para diferentes operações. Veja a Seção 16.4.2.

Definição de impasse • O impasse é um estado no qual cada membro de um grupo de transações está esperando que algum outro membro libere uma trava. Um grafo *espera por* pode ser usado para representar os relacionamentos de espera entre transações correntes. Em um grafo *espera por*, os nós representam transações e as setas representam os relacionamentos *espera por* entre as transações – existe uma seta do nó T para o nó U , quando a transação T está esperando que a transação U libere uma trava. Veja a Figura 16.20, que ilustra o grafo *espera por* correspondente à situação de impasse ilustrada na

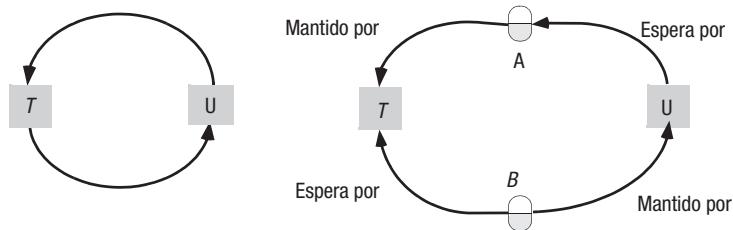


Figura 16.20 O grafo *espera por* da Figura 16.19.

Figura 16.19. Lembre-se de que o impasse surgiu porque as transações T e U tentaram adquirir um objeto mantido pela outra. Portanto T espera por U e U espera por T . A dependência entre as transações é indireta – via uma dependência nos objetos. O diagrama da direita mostra os objetos mantidos e esperados pelas transações T e U . Como cada transação pode esperar pelo único objeto, os objetos podem ser omitidos do grafo *espera por* – produzindo o grafo simples da esquerda.

Suponha que, como aparece na Figura 16.21, um grafo *espera por* contenha um ciclo $T \rightarrow U \rightarrow \dots \rightarrow V \rightarrow T$; então, cada transação está esperando pela próxima transação no ciclo. Todas essas transações estão paradas, esperando pelas travas. Nenhuma das travas pode ser liberada e as transações estão em um impasse. Se uma das transações em um ciclo for cancelada, então suas travas serão liberadas e esse ciclo será quebrado. Por exemplo, se a transação T na Figura 16.21 for cancelada, ela liberará uma trava sobre um objeto pelo qual V está esperando – e V não estará mais esperando por T .

Agora, considere um cenário no qual as três transações T , U e V compartilham uma trava de leitura sobre um objeto C , a transação W mantém uma trava de escrita sobre o objeto B , no qual a transação V está esperando para obter uma trava, como mostrado à direita na Figura 16.22. Então, as transações T e W solicitam travas de escrita sobre o objeto C e surge uma situação de impasse, na qual T espera por U e V , V espera por W , W espera por T , U e V , como mostrado à esquerda na Figura 16.22. Isso mostra que, embora cada transação possa esperar por apenas um objeto por vez, ela pode estar envolvida em vários ciclos. Por exemplo, a transação V está envolvida nos ciclos: $V \rightarrow W \rightarrow T \rightarrow V$ e $V \rightarrow W \rightarrow V$.

Nesse exemplo, suponha que a transação V seja cancelada. Isso liberará a trava de V sobre C e os dois ciclos envolvendo V serão quebrados.

Prevenção de impasses • Uma solução é evitar os impasses. Uma maneira aparentemente simples, mas não muito boa, de superar o impasse é adquirir as travas de todos os objetos usados por uma transação, quando ela inicia. Isso precisaria ser feito em um único

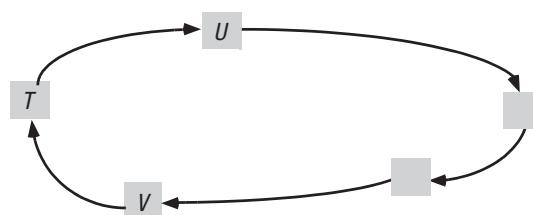


Figura 16.21 Um ciclo em um grafo *espera por*.

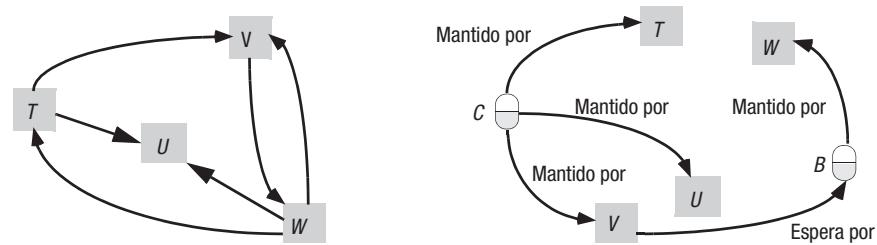


Figura 16.22 Outro grafo espera por.

passo atômico, para evitar um impasse nesse estágio. Tal transação não pode entrar em um impasse com outras transações, mas ela restringe desnecessariamente o acesso aos recursos compartilhados. Além disso, às vezes é impossível prever, no início de uma transação, quais objetos serão usados. Geralmente isso acontece em aplicativos interativos, pois o usuário teria de dizer com antecedência quais objetos exatamente estaria planejando usar, o que é inconcebível em aplicativos que fazem navegação, os quais permitem aos usuários localizar objetos que eles não conhecem antecipadamente. O impasse também pode ser evitado pela solicitação de travas sobre objetos em uma ordem predefinida, mas isso pode resultar em travamento prematuro e em uma redução na concorrência.

Travas upgrade • O *Concurrency Control Service* do CORBA apresenta um terceiro tipo de trava, chamada de *upgrade*, cuja utilização se destina a evitar impasses. Um impasse é frequentemente causado por duas transações conflitantes, primeiro obtendo travas de leitura e depois tentando promovê-las a travas de escrita. Uma transação com uma trava *upgrade* sobre um item de dados pode ler esse item, mas a trava entra em conflito com quaisquer travas *upgrade* configuradas por outras transações sobre o mesmo item de dados. Esse tipo de trava não pode ser configurado implicitamente pelo uso de uma operação de *leitura*, mas deve ser solicitado pelo cliente.

Detecção de impasses • Os impasses podem ser detectados pela descoberta de ciclos no grafo *espera por*. Tendo detectado um impasse, uma transação deve ser selecionada para cancelamento para quebrar o ciclo.

O *software* responsável pela detecção de impasses pode fazer parte do gerenciador de travas. Ele deve conter uma representação do grafo *espera por*, para que possa verificar a existência de ciclos de tempos em tempos. Setas são adicionadas e removidas do grafo pelas operações *setLock* e *unLock* do gerenciador de travas. No ponto ilustrado pela Figura 16.22 à esquerda haverá as seguintes informações:

Transação	Espera pela transação
T	U, V
V	W
W	T, U, V

Uma seta $T \rightarrow U$ é adicionada quando o gerenciador de travas impede uma requisição da transação T por uma trava sobre um objeto que já está travado pela transação U . Note que, quando uma trava é compartilhada, várias setas podem ser adicionadas. Uma seta T

Transação <i>T</i>		Transação <i>U</i>	
Operações	Travas	Operações	Travas
<i>a.deposit(100);</i>	trava de escrita <i>A</i>	<i>b.deposit(200)</i>	trava de escrita <i>B</i>
<i>b.withdraw(100)</i>			
...	espera pela trava <i>B</i> mantida por <i>U</i>	<i>a.withdraw(200);</i>	espera pela trava <i>A</i> mantida por <i>T</i>
(decorre o tempo limite)		...	sobre <i>A</i>
trava <i>A</i> em <i>T</i> se torna vulnerável, destrava <i>A</i> , cancela <i>T</i>		...	
		<i>a.withdraw(200);</i>	trava de escrita <i>A</i>
			destrava <i>A, B</i>

Figura 16.23 Solução do impasse da Figura 16.19.

→ *U* é excluída quando *U* libera uma trava pela qual *T* estava esperando e permite que *T* prossiga. Veja no Exercício 16.14 uma discussão mais detalhada sobre a implementação de detecção de impasses. Se uma transação compartilha uma trava, a trava não é liberada, mas as setas que levam a uma transação em particular são removidas.

A presença de ciclos pode ser verificada sempre que uma seta é adicionada (ou menos frequentemente, para evitar sobrecarga desnecessária). Quando um impasse é detectado, uma das transações no ciclo deve ser escolhida e, então, cancelada. O nó e as setas correspondentes que a envolvem devem ser removidos do grafo *espera por*. Isso acontecerá quando a transação cancelada tiver suas travas removidas.

A escolha da transação a ser cancelada não é simples. Alguns fatores que podem ser levados em conta são a idade da transação e o número de ciclos em que ela está envolvida.

Tempos limites (timeouts) • A limitação do tempo de bloqueio representa um método comumente usado para a solução de impasses. Cada trava recebe um período de tempo limitado, durante o qual ela é invulnerável. Após esse tempo, a trava se torna vulnerável. Desde que nenhuma outra transação esteja competindo pelo objeto travado, um objeto com uma trava vulnerável permanece travado. Entretanto, se qualquer outra transação estiver esperando para acessar o objeto protegido por uma trava vulnerável, a trava será quebrada (isto é, o objeto será destravado) e a transação em espera será retomada. A transação cuja trava foi quebrada normalmente é cancelada.

Existem muitos problemas no uso de tempos limites como solução para impasses: o pior deles é que, às vezes, as transações são canceladas porque suas travas se tornam vulneráveis quando outras transações estão esperando por elas, mas, na verdade, não há nenhum impasse. Em um sistema sobrecarregado, o número de transações com tempo limite esgotado aumentará, e as transações que demoram um longo tempo podem ser penalizadas. Além disso, é difícil decidir sobre a duração apropriada para um tempo limite. Em contraste, se for usada detecção de impasses, as transações serão canceladas, porque

Para um objeto		Trava a ser configurada		
		leitura	escrita	confirmação
Trava já configurada	nenhum	OK	OK	OK
	leitura	OK	OK	espera
	escrita	OK	espera	-
	confirmação	espera	espera	-

Figura 16.24 Compatibilidade entre travas (travas de *leitura*, *escrita* e *confirmação*).

ocorrem impasses e pode ser feita uma escolha com relação à qual transação vai ser cancelada.

Usando tempos limites para travas, podemos resolver o impasse da Figura 16.19, como mostrado na Figura 16.23, na qual a trava de escrita sobre *A* em *T* se torna vulnerável após seu período de tempo limite ter decorrido. A transação *U* está esperando para adquirir uma trava de escrita sobre *A*. Portanto, *T* é cancelada e libera sua trava sobre *A*, permitindo que *U* retome e complete a transação.

Quando as transações acessam objetos localizados em vários servidores diferentes, surge a possibilidade de impasses distribuídos. Em um impasse distribuído, o grafo *espera por* pode envolver objetos em vários locais. Voltaremos a esse assunto na Seção 17.5.

16.4.2 Aumento da concorrência em esquemas de travamento

Mesmo quando as regras de travamento são baseadas no conflito entre operações de *leitura* e *escrita* e a granularidade com que elas são aplicadas é a menor possível, ainda há uma chance de aumentar a concorrência. Vamos discutir duas estratégias que têm sido usadas. Na primeira (travamento de duas versões), a configuração de travas exclusivas é retardada até que uma transação seja confirmada. Na segunda estratégia (travas hierárquicas), são usadas travas com granularidade mista.

Travamento de duas versões • Trata-se de um esquema otimista que permite a uma transação gravar versões de tentativa dos objetos, enquanto outras transações leem a versão confirmada dos mesmos objetos. As operações de *leitura* só ficam na espera se outra transação estiver correntemente confirmando o mesmo objeto. Esse esquema possibilita mais concorrência do que as travas de leitura e escrita, mas as transações de escrita correm o risco de espera ou mesmo de anulação, quando tentam ser confirmadas. As transações não podem confirmar suas operações de *escrita* imediatamente, caso outras transações não concluídas tenham lido os mesmos objetos. Portanto, as transações que pedem para ser confirmadas em tal situação são obrigadas a esperar até que as transações de leitura tenham terminado. Um impasse pode ocorrer quando as transações estão esperando para serem confirmadas. Portanto, para resolver os impasses, talvez transações precisem ser canceladas quando estão esperando para serem confirmadas.

Essa variação do travamento de duas fases restrito usa três tipos de travas: uma trava de leitura, uma trava de escrita e uma trava de confirmação. Antes que a operação de *leitura* de uma transação seja efetuada, deve ser posta uma trava de leitura sobre o objeto – a tentativa de pôr uma trava de leitura é bem-sucedida, a não ser que o objeto tenha uma trava de confirmação, no caso em que a transação esperará. Antes que a operação de

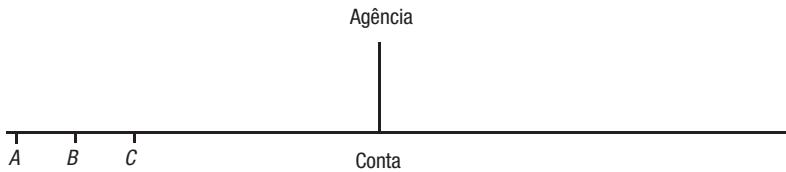


Figura 16.25 Hierarquia de travas do exemplo de transações bancárias.

escrita de uma transação seja efetuada, uma trava de escrita deve ser posta sobre o objeto – a tentativa de pôr uma trava de escrita é bem-sucedida, a não ser que o objeto tenha uma trava de escrita ou uma trava de confirmação, caso em que a transação esperará.

Quando o coordenador de transação recebe uma requisição para confirmar uma transação, ele tenta converter todas as travas de escrita dessa transação em travas de confirmação. Se qualquer um dos objetos tiver travas de leitura pendentes, a transação deverá esperar até que as transações que usam essas travas tenham terminado e as travas sejam liberadas. A compatibilidade das travas de leitura, escrita e confirmação aparecem na Figura 16.24.

Existem duas diferenças principais no desempenho entre o esquema de travamento de duas versões e um esquema de travamento de leitura e escrita normal. Por um lado, as operações de *leitura* no esquema de travamento de duas versões são retardadas apenas enquanto as transações estão sendo confirmadas, em vez de o serem durante a execução inteira das transações – na maioria dos casos, o protocolo de confirmação leva apenas uma pequena fração do tempo exigido para realizar uma transação inteira. Por outro lado, as operações de *leitura* de uma transação podem causar um atraso na confirmação de outras transações.

Travas hierárquicas • Em algumas aplicações, a granularidade conveniente para uma operação não é apropriada para outra. Em nosso exemplo de transações bancárias, a maioria das operações exige o travamento com a granularidade de uma conta. A operação *branchTotal* é diferente – ela lê os valores dos saldos de todas as contas e exige uma trava de leitura sobre todas elas. Para reduzir a sobrecarga do travamento, seria útil permitir a coexistência de travas de granularidade mista.

Gray [1978] propôs o uso de uma hierarquia de travas com diferentes granularidades. Em cada nível, a configuração de uma trava ascendente tem o mesmo efeito que configurar todas as travas descendentes equivalentes. Isso traz uma economia no número de travas a serem configuradas. Em nosso exemplo de transações bancárias, a agência é a transação ascendente e as contas são as descendentes (veja a Figura 16.25).

As travas de granularidade mista poderiam ser úteis em um sistema de agenda, no qual os dados poderiam ser estruturados, com a agenda de uma semana sendo

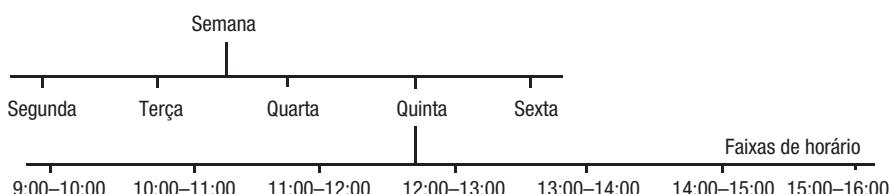


Figura 16.26 Hierarquia de travas para uma agenda.

Para um objeto		Trava a ser configurada			
		leitura	escrita	I-leitura	I-escrita
Trava já configurada	nenhum	OK	OK	OK	OK
	leitura	OK	espera	OK	espera
	escrita	espera	espera	espera	espera
	I-leitura	OK	espera	OK	OK
	I-escrita	espera	espera	OK	OK

Figura 16.27 Tabela de compatibilidade para travas.

composta de uma página para cada dia e cada dia subdividido ainda em uma repartição para cada hora, como mostrado na Figura 16.26. A operação para ver uma semana faria uma trava de leitura ser posta no topo dessa hierarquia, enquanto a operação para inserir um compromisso faria uma trava de escrita ser posta em uma faixa de horário. O efeito de uma trava de leitura sobre uma semana impediria operações de escrita em qualquer uma das subestruturas; por exemplo, as faixas de horário de cada dia nessa semana.

No esquema de Gray, cada nó na hierarquia pode ser travado – dando ao proprietário da trava acesso explícito ao nó e implícito aos seus descendentes. Em nosso exemplo, na Figura 16.25, uma trava de leitura/escrita sobre a agência trava implicitamente todas as contas para leitura e escrita. Antes que um nó descendente receba uma trava de leitura/escrita, a intenção de usar essa trava é configurada no nó ascendente e em seus ancestrais (se houver). A trava de intenção é compatível com outras travas de intenção, mas entra em conflito com as travas de leitura e escrita, de acordo com as regras normais. A Figura 16.27 fornece a tabela de compatibilidade para travas hierárquicas. Gray também propôs um terceiro tipo de trava, a de intenção – a qual combina a propriedade de uma trava de leitura com uma intenção de escrita.

Em nosso exemplo de transações bancárias, a operação *branchTotal* solicita uma trava de leitura sobre a agência, a qual configura travas de leitura implicitamente sobre todas as contas. Uma operação *deposit* precisa configurar uma trava de escrita sobre um saldo, mas primeiro ela tenta configurar uma intenção de trava de escrita sobre a agência. Essas regras impedem que essas operações sejam efetuadas concorrentemente.

As travas hierárquicas têm a vantagem de serem em menor número quando é exigido um travamento de granularidade mista. As tabelas de compatibilidade e as regras para promover travas são mais complexas.

A granularidade mista das travas poderia deixar que cada transação travasse uma parte, cujo tamanho seria escolhido de acordo com suas necessidades. Uma transação longa, que acessasse muitos objetos, poderia travar o conjunto inteiro, enquanto uma transação curta poderia fazer o travamento com uma granularidade menor.

O *Concurrency Control Service* do CORBA suporta travas de granularidade variável com tipos de travas de intenção de leitura e de escrita. Elas podem ser usadas conforme descrito anteriormente, para aproveitar a oportunidade de aplicar travas com diferentes granularidades em dados estruturados hierarquicamente.

16.5 Controle de concorrência otimista

Kung e Robinson [1981] identificaram várias desvantagens inerentes ao travamento e propuseram uma estratégia otimista alternativa à serialização das transações, que evita esses inconvenientes. Podemos resumir os inconvenientes do bloqueio, como segue:

- A manutenção da trava representa uma sobrecarga que não está presente em sistemas que não suportam acesso concorrente a dados compartilhados. Em geral, mesmo as transações somente de leitura (consultas), que possivelmente não afetam a integridade dos dados, devem usar travas para garantir que os dados que estão sendo lidos não sejam modificados simultaneamente por outras transações. No entanto, o travamento pode ser necessário apenas no pior caso.

Por exemplo, considere dois processos clientes que estão incrementando os valores de n objetos concorrentemente. Se os programas clientes começam simultaneamente e são executados durante o mesmo período de tempo, acessando os objetos em duas sequências não relacionadas e usando uma transação separada para acessar e incrementar cada item, as chances de que os dois programas tentem acessar o mesmo objeto ao mesmo tempo são de apenas uma em n , em média; portanto, o travamento é realmente necessário apenas uma vez em cada n transações.

- O uso de travas pode resultar em um impasse. A prevenção de impasses reduz seriamente a concorrência e, portanto, as situações de impasse devem ser resolvidas com o uso de tempos limites ou com detecção de impasses. Nenhuma delas é totalmente satisfatória para uso em programas interativos.
- Para evitar os cancelamentos em cascata, as travas não podem ser liberadas até o final da transação. Isso pode reduzir significativamente o potencial de concorrência.

A estratégia alternativa proposta por Kung e Robinson é otimista porque é baseada na observação de que, na maioria das aplicações, a probabilidade das transações de dois clientes acessarem o mesmo objeto é baixa. As transações podem prosseguir como se não houvesse nenhuma possibilidade de conflito com outras transações, até que o cliente conclua sua tarefa e emita uma requisição de *closeTransaction*. Quando surge um conflito, alguma transação geralmente é cancelada e precisará ser reiniciada pelo cliente. Cada transação tem as seguintes fases:

Fase de trabalho: durante a fase de trabalho, cada transação tem uma versão tentativa de cada um dos objetos que atualiza. Trata-se de uma cópia da versão do objeto confirmada mais recentemente. O uso de versões de tentativa permite que a transação seja cancelada (sem nenhum efeito sobre os objetos) durante a fase de trabalho ou se ela falhar na validação, devido a outras transações conflitantes. As operações de *leitura* são executadas imediatamente – se já existir uma versão de tentativa para essa transação, uma operação de *leitura* a acessará; caso contrário, ela acessará o valor do objeto confirmado mais recentemente. As operações de *escrita* registram os novos valores dos objetos como valores de tentativa (que são invisíveis para as outras transações). Quando existem várias transações concomitantes, diversos valores de tentativa diferentes do mesmo objeto podem coexistir. Além disso, são mantidos dois registros dos objetos acessados dentro de uma transação: um *conjunto de leitura*, contendo os objetos lidos pela transação, e um *conjunto de escrita*, contendo os objetos modificados por ela. Note que, como todas as operações de *leitura* são realizadas em versões confirmadas dos objetos (ou em cópias deles), não há ocorrência de leituras sujas.

Fase de validação: quando a requisição de *closeTransaction* é recebida, a transação é validada para estabelecer se suas operações sobre os objetos entram em conflito ou não com as operações de outras transações sobre os mesmos objetos. Se a validação for bem-sucedida, então a transação poderá ser confirmada. Se a validação falhar, alguma forma de solução de conflito deve ser usada, e a transação corrente ou, em alguns casos, aquelas com que ela está em conflito, precisarão ser canceladas.

Fase de atualização: se uma transação é validada, todas as alterações registradas em suas versões de tentativa tornam-se permanentes. As transações somente de leitura podem ser confirmadas imediatamente, após passarem pela validação. As transações de escrita estarão prontas para serem confirmadas quando as versões de tentativa dos objetos tiverem sido registradas no meio de armazenamento permanente.

Validação de transações • A validação usa as regras de conflito de leitura e escrita para garantir que a programação de uma transação em particular seja serialmente equivalente com relação a todas as outras transações *sobrepostas* – isto é, todas as transações que ainda não tinham sido confirmadas no momento que essa transação começou. Para ajudar na realização da validação, cada transação recebe um número ao entrar na fase de validação (isto é, quando o cliente emite uma requisição de *closeTransaction*). Se a transação for validada e terminar com sucesso, ela manterá esse número; se falhar nas verificações da validação e for cancelada, ou se a transação for somente de leitura, o número será liberado para uma nova atribuição posterior. Os números de transação são valores inteiros atribuídos em sequência ascendente; portanto, o número de uma transação define sua posição no tempo – uma transação sempre termina sua fase de trabalho após todas as transações que possuem números mais baixos. Isto é, uma transação com o número T_i sempre precede uma transação com o número T_j , se $i < j$. (Se o número da transação fosse atribuído no início da fase de trabalho, uma transação que chegasse ao final dessa fase antes de outra com um número menor teria de esperar até que a anterior tivesse terminado, antes de poder ser validada.)

O teste de validação na transação T_v é baseado no conflito entre operações em pares de transação T_i e T_v . Para que uma transação T_v seja disposta em série com relação a uma transação sobreposta T_i , suas operações devem obedecer às seguintes regras:

T_v	T_i	Regra
escrita	leitura	1. T_i não deve ler objetos escritos por T_v .
leitura	escrita	2. T_v não deve ler objetos escritos por T_i .
escrita	escrita	3. T_i não deve escrever em objetos modificados por T_v e T_v não deve escrever em objetos modificados por T_i .

Como as fases de validação e atualização de uma transação geralmente têm curta duração, comparadas com a fase de trabalho, uma simplificação pode ser obtida criando-se a regra de que apenas uma transação pode estar na fase de validação e atualização em dado momento. Quando duas transações não podem se sobrepor na fase de atualização, a regra 3 é satisfeita. Note que essa restrição sobre as operações de *escrita*, junto ao fato de que não podem ocorrer leituras sujas, produz execuções restritas. Para impedir a sobreposição, as fases de validação e atualização inteiras podem ser implementadas como uma seção crítica para que apenas um cliente por vez possa executá-la. Para aumentar a

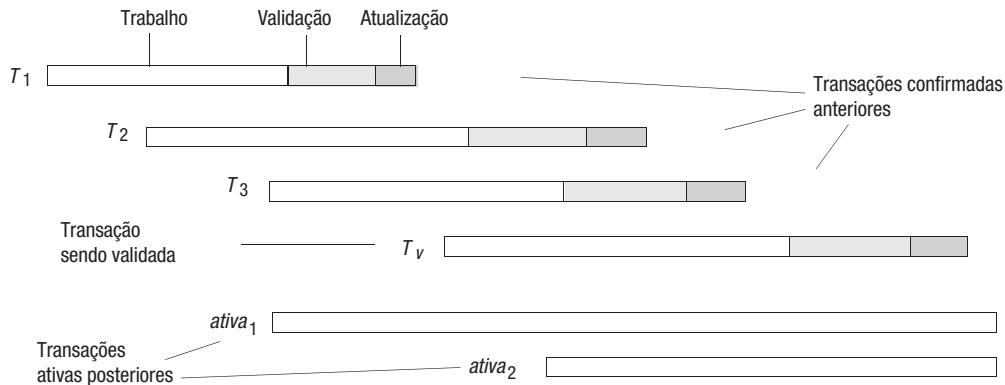


Figura 16.28 Validação de transações.

concorrência, parte da validação e da atualização pode ser implementada fora da seção crítica, mas é fundamental que a atribuição de números de transação seja realizada em sequência. Notamos que, a qualquer instante, o número da transação corrente é como um pseudo-relógio que começa a funcionar quando uma transação termina com sucesso.

A validação de uma transação deve garantir que as regras 1 e 2 sejam obedecidas, testando as sobreposições entre os objetos de pares de transações T_v e T_i . Existem duas formas de validação – para trás (*backward*) e para frente (*forward*) [Härder 1984]. A validação para trás verifica a transação sendo submetida à validação com outras transações sobrepostas precedentes – aquelas que entraram na fase de validação antes dela. A validação para frente verifica a transação sendo submetida à validação com outras transações posteriores, que ainda estão ativas.

Validação para trás (backward) • Como todas as operações de *leitura* das transações sobrepostas anteriores foram executadas antes que a validação de T_v começasse, elas não podem ser afetadas pelas escritas da transação corrente (e a regra 1 é satisfeita). A validação da transação T_v verifica se seu conjunto de leitura (os objetos afetados pelas operações de *leitura* de T_v) se sobrepõe a qualquer um dos conjuntos de escrita das transações sobrepostas anteriores, T_i (regra 2). Se houver qualquer sobreposição, a validação falhará.

Seja $startTn$ o maior número de transação atribuído (para alguma outra transação confirmada) no momento em que a transação T_v começou sua fase de trabalho e $finishTn$ o maior número de transação atribuído no momento em que T_v entrou na fase de validação. O programa a seguir descreve o algoritmo para a validação de T_v :

```
boolean valid = true;
for (int  $T_i = startTn+1$ ;  $T_i \leq finishTn$ ;  $T_i++$ ){
    if (conjunto de leitura de  $T_v$  possui interseção com o conjunto de escrita de  $T_i$ )
        valid = false;
}
```

A Figura 16.28 mostra transações sobrepostas que poderiam ser consideradas na validação de uma transação T_v . O tempo aumenta da esquerda para a direita. As transações confirmadas anteriores são T_1 , T_2 e T_3 . T_1 foi confirmada antes que T_v começasse. T_2 e T_3 foram confirmadas antes que T_v terminasse sua fase de trabalho. $startTn + 1 = T_2$ e $finishTn = T_3$. Na validação para trás, o conjunto de leitura de T_v deve ser comparado com os conjuntos de escrita de T_2 e T_3 .

Na validação para trás, o conjunto de leitura da transação que está sendo validada é comparado com os conjuntos de escrita das outras transações que já foram confirmadas. Portanto, a única maneira de resolver qualquer conflito é cancelar a transação que está passando pela validação.

Na validação para trás, as transações que não têm operações de *leitura* (operações somente de *escrita*) não precisam ser verificadas.

O controle de concorrência otimista com validação para trás exige que os conjuntos de escrita de versões confirmadas antigas dos objetos, correspondentes às transações recentemente confirmadas, sejam mantidos até que não existam transações sobrepostas não validadas com as quais poderiam entrar em conflito. Quando uma transação é validada com sucesso, seu número de transação, *startTn*, e o conjunto de escritas são registrados em uma lista de transações precedentes, mantida pelo serviço de transação. Note que essa lista é ordenada pelo número de transação. Em um ambiente com transações longas, a retenção de conjuntos de escrita de objetos antigos pode ser um problema. Por exemplo, na Figura 16.28, os conjuntos de escrita de T_1 , T_2 , T_3 e T_v devem ser mantidos até que a transação $ativa_1$ termine. Note que, embora as transações ativas tenham identificadores de transação, elas ainda não possuem números de transação.

Validação para frente (forward) • Na validação para frente da transação T_v , o conjunto de escrita de T_v é comparado com os conjuntos de leitura de todas as transações ativas sobrepostas – aquelas que ainda estão em sua fase de trabalho (regra 1). A regra 2 é satisfeita automaticamente, pois as transações ativas só escrevem depois que T_v tiver terminado. Se as transações ativas tiverem identificadores de transação (consecutivos) $ativa_1$ a $ativa_N$, então o programa a seguir descreve o algoritmo para a validação para frente de T_v :

```
boolean valid = true;
for (int  $T_{id} = active_1$ ;  $T_{id} \leq active_N$ ;  $T_{id}++$ ){
    if (conjunto de escrita de  $T_v$  possui interseção com o conjunto de leitura de  $T_{id}$ )
        valid = false;
}
```

Na Figura 16.28, o conjunto de escrita da transação T_v deve ser comparado com os conjuntos de leitura das transações com identificadores $ativa_1$ e $ativa_2$. (A validação para frente deve se preparar para o fato de que os conjuntos de leitura das transações ativas podem mudar durante a validação e a escrita.) Como os conjuntos de leitura da transação que está sendo validada não são incluídos na verificação, as transações somente de leitura sempre passam na verificação da validação. Como as transações que estão sendo comparadas com a transação que está sendo validada ainda estão ativas, temos a escolha de cancelar a transação que está sendo validada, ou adotar alguma maneira alternativa de resolver o conflito. Härder [1984] sugere várias estratégias alternativas:

- Adiar a validação até um momento posterior, quando as transações conflitantes tiverem terminado. Entretanto, não há garantia de que a transação que está sendo validada irá passar melhor no futuro. Sempre existe a chance de que mais transações ativas conflitantes possam começar, antes que a validação seja obtida.
- Cancelar todas as transações ativas conflitantes e confirmar a transação que está sendo validada.
- Cancelar a transação que está sendo validada. Esta é a estratégia mais simples, mas tem a desvantagem de que as futuras transações conflitantes podem vir a ser canceladas, no caso em que a transação sob validação tenha sido cancelada desnecessariamente.

Comparação das validações para frente e para trás • Já vimos que a validação para frente permite flexibilidade na solução de conflitos, enquanto a validação para trás permite apenas uma escolha – cancelar a transação que está sendo validada. Em geral, os conjuntos de leitura das transações são muito maiores do que os conjuntos de escrita. Portanto, a validação para trás compara um conjunto de leitura possivelmente grande com os conjuntos de escrita antigos, enquanto a validação para frente verifica um pequeno conjunto de escrita em relação aos conjuntos de leitura das transações ativas. Vemos que a validação para trás tem a sobrecarga de armazenar conjuntos de escrita antigos até que eles não sejam mais necessários. Por outro lado, a validação para frente tem que se preparar para novas transações que começem durante o processo de validação.

Inanição • Quando uma transação for cancelada, normalmente, ela será reiniciada pelo programa cliente, mas em esquemas que contam com o cancelamento e reinício das transações, não há nenhuma garantia de que uma transação em particular venha a passar nas verificações de validação, pois, sempre que é reiniciada, ela pode entrar em conflito com outras transações no uso de objetos. O processo de sempre impedir que uma transação seja capaz de ser confirmada é chamado de inanição.

As ocorrências de inanição provavelmente são raras, mas um servidor que utilize controle de concorrência otimista deve garantir que um cliente não tenha sua transação cancelada repetidamente. Kung e Robinson sugerem que isso poderia ser feito se o servidor detectasse uma transação que foi cancelada várias vezes. Eles sugerem que, quando o servidor detecta tal transação, ela deve receber acesso exclusivo para uso de uma seção crítica protegida por um semáforo.

16.6 Ordenação por carimbo de tempo

Nos esquemas de controle de concorrência baseados na ordenação por carimbo de tempo (*timestamp*), cada operação em uma transação é validada ao ser executada. Se a operação não puder ser validada, a transação será cancelada imediatamente e poderá, então, ser reiniciada pelo cliente. Cada transação recebe um valor de carimbo de tempo exclusivo ao iniciar. O carimbo de tempo define sua posição na sequência de tempo das transações. As requisições de transações podem ser totalmente ordenadas, de acordo com seus carimbos de tempo. A regra de ordenação básica por carimbo de tempo é baseada no conflito de operações e é muito simples:

A requisição de uma transação para escrever em um objeto é válida se esse objeto foi lido e escrito pela última vez por transações anteriores. A requisição de uma transação para ler um objeto é válida somente se esse objeto foi escrito pela última vez por uma transação anterior.

Essa regra presume que exista apenas uma versão de cada objeto e restringe o acesso a uma transação por vez. Se cada transação tiver sua própria versão de tentativa de cada objeto que acessa, então várias transações concorrentes poderão acessar o mesmo objeto. A regra de ordenação por carimbo de tempo é refinada para garantir que cada transação acesse um conjunto consistente de versões dos objetos. Ela também deve garantir que as versões de tentativa de cada objeto sejam efetivadas na ordem determinada pelos carimbos de tempo das transações que as fizeram. Isso é obtido com as transações esperando, quando necessário, que transações anteriores terminem suas escritas. As operações de *escrita* podem ser executadas depois que a operação *closeTransaction* tiver retornado, sem fazer o cliente

<i>Regra</i>	T_c	T_i	
1.	<i>escrita</i>	<i>leitura</i>	T_c não deve escrever um objeto que tenha sido lido por qualquer T_i , onde $T_i > T_c$; isso exige que $T_c \geq$ o carimbo de tempo de leitura máximo do objeto.
2.	<i>escrita</i>	<i>escrita</i>	T_c não deve escrever um objeto que tenha sido modificado por qualquer T_i , onde $T_i > T_c$; isso exige que $T_c >$ carimbo de tempo de escrita do objeto confirmado.
3.	<i>leitura</i>	<i>escrita</i>	T_c não deve ler um objeto que tenha sido modificado por qualquer T_i , onde $T_i > T_c$; isso exige que $T_c >$ carimbo de tempo de escrita do objeto confirmado.

Figura 16.29 Conflito de operação para ordenação por carimbo de tempo.

esperar. Contudo, o cliente deve esperar quando operações de *leitura* precisam esperar que transações anteriores sejam concluídas. Isso não pode levar a um impasse, pois as transações só esperam pelas anteriores (e nenhum ciclo poderia ocorrer no grafo espera por).

Os carimbos de tempo podem ser atribuídos a partir do relógio do servidor ou, como na seção anterior, uma pseudo-hora pode ser baseada em um contador incrementando quando um valor de carimbo de tempo for emitido. Deixaremos para o Capítulo 17 o problema da geração de carimbos de tempo quando o serviço de transação é distribuído e vários servidores estão envolvidos em uma transação.

Descreveremos agora uma forma de controle de concorrência baseado no carimbo de tempo, seguindo os métodos adotados no sistema SDD-1 [Bernstein *et al.* 1980] e descritos por Ceri e Pelagatti [1985].

Como sempre, as operações de *escrita* são registradas em versões de tentativa dos objetos e são invisíveis para as outras transações até que uma requisição *closeTransaction* seja emitida e a transação seja confirmada. Todo objeto tem o carimbo de tempo de escrita máximo e um conjunto de versões de tentativa, cada uma das quais tendo um carimbo de tempo de escrita associado e um conjunto de carimbo de tempo de leitura. O carimbo de tempo de escrita do objeto (confirmado) é anterior ao de qualquer um de suas versões de tentativa, e o conjunto de carimbo de tempo de leitura pode ser representado por seu membro máximo. Quando a operação de *escrita* de uma transação sobre um objeto é aceita, o servidor cria uma nova versão de tentativa do objeto, com carimbo de tempo de escrita configurado no carimbo de tempo da transação. A operação de *leitura* de uma transação é direcionada para a versão com um carimbo de tempo de escrita menor do que o carimbo de tempo da transação. Quando a operação de *leitura* de uma transação sobre um objeto é aceita, o carimbo de tempo da transação é adicionado em seu conjunto de carimbo de tempo de leitura. Quando uma transação é confirmada, os valores das versões de tentativa tornam-se os valores dos objetos, e os carimbos de tempo das versões de tentativa tornam-se os carimbos de tempo dos objetos correspondentes.

Na ordenação por carimbo de tempo, cada requisição de uma transação para uma operação de *leitura* ou *escrita* sobre um objeto é verificada para ver se obedece às regras de conflito de operação. Uma requisição da transação corrente T_c pode entrar em conflito com operações anteriores executadas por outras transações, T_i , cujos carimbos de tempo mostram que elas devem ser posteriores a T_c . Essas regras aparecem na Figura 16.29, na qual $T_i > T_c$ significa que T_i é posterior a T_c e $T_i < T_c$ significa que T_i é anterior a T_c .

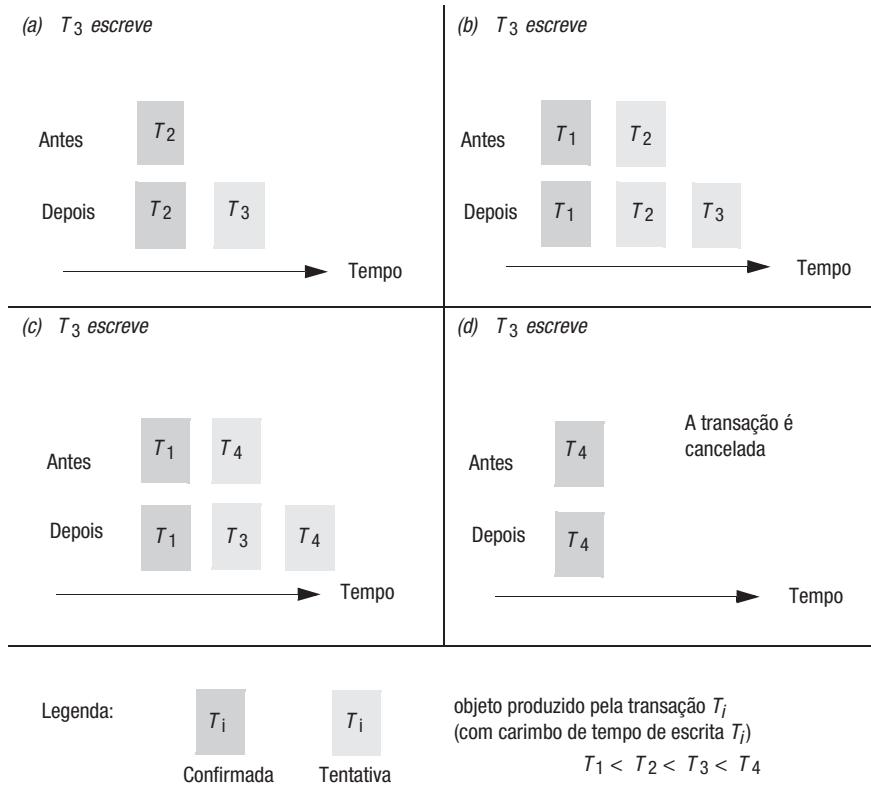


Figura 16.30 Operações de escrita e carimbos de tempo.

Regra de escrita na ordenação por carimbo de tempo: combinando as regras 1 e 2, temos a seguinte regra para decidir se devemos aceitar uma operação de *escrita* solicitada pela transação T_c sobre o objeto D :

```

se ( $T_c \geq$  carimbo de tempo de leitura máximo em  $D \&&$ 
 $T_c >$  carimbo de tempo de escrita na versão confirmada de  $D$ )
    executa a operação de escrita na versão de tentativa de  $D$ , com carimbo de
    tempo de escrita  $T_c$ 
senão /* a escrita se deu tarde demais */
    Cancela a transação  $T_c$ 
  
```

Se já existe uma versão de tentativa com carimbo de tempo de escrita T_c , a operação de *escrita* é endereçada a ela; caso contrário, uma nova versão de tentativa é criada e recebe um carimbo de tempo de escrita T_c . Note que qualquer *escrita* que chegue tarde demais será cancelada – ela ocorreu tarde demais no sentido de que uma transação com um carimbo de tempo posterior já leu ou escreveu no objeto.

A Figura 16.30 ilustra a ação de uma operação de escrita pela transação T_3 nos casos em que $T_3 \geq$ carimbo de tempo de leitura máximo no objeto (os carimbos de tempo de leitura não são mostrados). Nos casos (a) a (c), $T_3 >$ carimbo de tempo de escrita na versão confirmada do objeto, e uma versão de tentativa com carimbo de tempo de escrita

T_3 é inserida no lugar apropriado da lista de versões de tentativa ordenado por seus carimbos de tempo de transação. No caso (d), $T_3 <$ carimbo de tempo de escrita na versão confirmada do objeto, e a transação é cancelada.

Regra de leitura na ordenação por carimbo de tempo: usando a regra 3, temos a seguinte regra para decidir se devemos aceitar imediatamente, esperar ou rejeitar uma operação de leitura solicitada pela transação T_c sobre o objeto D :

```

se ( $T_c >$  carimbo de tempo de escrita na versão confirmada de  $D$ ) {
    seja  $D_{selecionada}$  a versão de  $D$  com o carimbo de tempo de escrita máximo  $\leq T_c$ 
    se ( $D_{selecionada}$  for confirmada)
        executa a operação de leitura na versão  $D_{selecionada}$ 
    senão
        Espera até que a transação que fez a versão  $D_{selecionada}$  seja confirmada ou
        cancelada e depois aplica a regra de leitura
    }
    senão
        Cancela a transação  $T_c$ 

```

Nota:

- Se a transação T_c já tiver escrito sua própria versão do objeto, isso será usado.
- Uma operação de *leitura* que chegue cedo demais esperará que a transação anterior termine. Se a transação anterior for confirmada, então T_c lerá sua versão confirmada. Se ela for cancelada, T_c repetirá a regra de leitura (e selecionará a versão anterior). Essa regra impede leituras sujas.
- Uma operação de *leitura* que chegue tarde demais será cancelada – é tarde demais no sentido de que uma transação com um carimbo de tempo posterior já escreveu no objeto.

A Figura 16.31 ilustra a regra de leitura da ordenação por carimbo de tempo. Ela inclui quatro casos, rotulados de (a) a (d), cada um dos quais ilustrando a ação de uma operação de *leitura* pela transação T_3 . Em cada caso, é selecionada uma versão cujo carimbo de tempo de escrita é menor ou igual a T_3 . Se tal versão existe, ela é indicada com uma linha. Nos casos (a) e (b), a operação de *leitura* é direcionada para uma versão confirmada – em (a), essa é a única versão, enquanto em (b) existe uma versão de tentativa pertencente a uma transação posterior. No caso (c), a operação de *leitura* é direcionada para uma versão de tentativa e deve esperar até que a transação que a fez seja confirmada ou cancelada. No caso (d), não existe nenhuma versão conveniente para ler, e a transação T_3 é cancelada.

Quando um coordenador receber uma requisição para confirmar uma transação, ele sempre poderá fazer isso, pois todas as operações das transações têm sua consistência verificada com relação às transações anteriores, antes de serem executadas. As versões confirmadas de cada objeto devem ser criadas na ordem do carimbo de tempo. Portanto, às vezes, um coordenador precisa esperar que as transações anteriores terminem, antes de escrever todas as versões confirmadas dos objetos acessados por uma transação em particular, mas não há necessidade de que o cliente espere. Para tornar uma transação recuperável após uma falha do servidor, as versões de tentativa dos objetos, e o fato de a transação ter sido confirmada, devem ser escritas no meio de armazenamento permanente antes do aceite da requisição do cliente para confirmar a transação.

Note que esse algoritmo de ordenação por carimbo de tempo é restrito – ele garante execuções restritas das transações (veja a Seção 16.2). A regra de leitura da ordenação por carimbo de tempo atrasa a operação de *leitura* de uma transação em qualquer objeto, até que todas as transações que haviam escrito esse objeto anteriormente tenham sido confirmadas

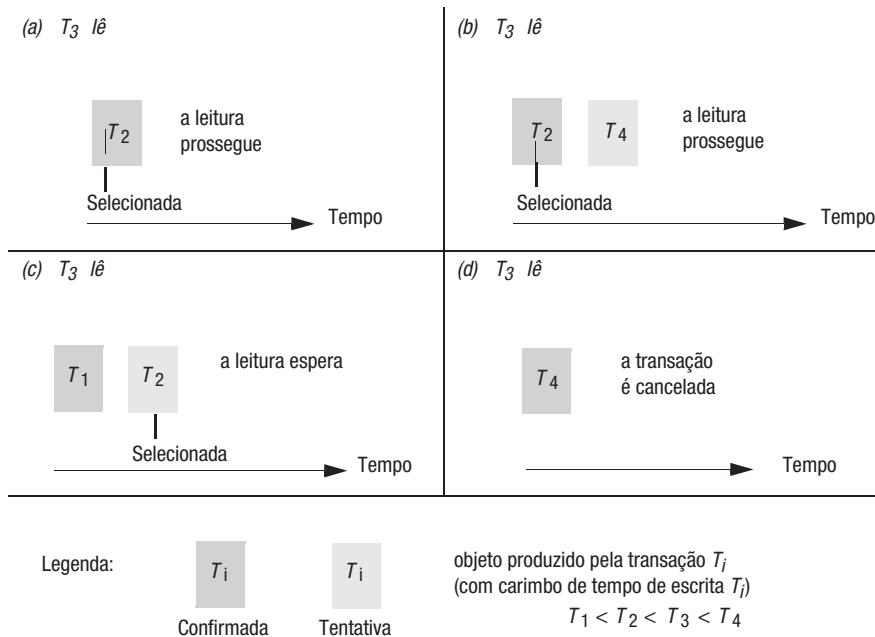


Figura 16.31 Operações de *leitura* e carimbos de tempo.

ou canceladas. Os preparativos para confirmar as versões garantem que a execução da operação de *escrita* de uma transação, em qualquer objeto, seja retardada até que todas as transações que haviam escrito esse objeto anteriormente tenham sido confirmadas ou canceladas.

Na Figura 16.32, voltamos à nossa ilustração relativa às duas transações bancárias concorrentes T e U , apresentadas na Figura 16.7. As colunas encabeçadas com A , B e C se referem às informações sobre contas com aqueles nomes. Cada conta tem uma entrada CTL (Carimbo de Tempo de Leitura) que registra o carimbo de tempo de leitura máximo e uma entrada CTE (Carimbo de Tempo de Escrita) que registra o carimbo de tempo de escrita de cada versão – com os carimbos de tempo das versões confirmadas em negrito. Inicialmente, todas as contas têm versões confirmadas gravadas pela transação S , e o conjunto de carimbos de tempo de leitura está vazio. Supomos $S < T < U$. O exemplo mostra que, quando a transação U estiver pronta para obter o saldo de B , esperará que T termine para que possa ler o valor configurado por T , se ela for confirmada.

O método de carimbo de tempo que acabamos de descrever evita impasses, mas é bastante propenso a reinícios. Uma modificação conhecida como regra de *ignorar escrita obsoleta* é uma melhoria. Trata-se de uma modificação na regra de escrita na ordenação por carimbo de tempo:

Se uma escrita for feita tarde demais, ela pode ser ignorada, em vez de cancelar a transação, pois se ela tivesse chegado a tempo, seus efeitos teriam sido sobreescritos de qualquer forma. Entretanto, se outra transação tiver lido o objeto, a transação com a escrita tardia falhará, devido ao carimbo de tempo de leitura presente no item.

Ordenação por carimbo de tempo de versão múltipla • Nesta seção, mostramos como a concorrência fornecida pela ordenação por carimbo de tempo básica é melhorada, permitindo-se que cada transação escreva suas próprias versões de tentativa dos objetos.

		Carimbos de tempo e versões de objetos					
T	U	A	B	C			
		CTL	CTE	CTL	CTE	CTL	CTE
		{}	S	{}	S	{}	S
<i>openTransaction</i>							
<i>bal = b.getBalance()</i>							{T}
	<i>openTransaction</i>						
<i>b.setBalance(bal*1.1)</i>							S, T
	<i>bal = b.getBalance()</i>						
	<i>wait for T</i>						
<i>a.withdraw(bal/10)</i>	•••						S, T
<i>commit</i>	•••						T
	<i>bal = b.getBalance()</i>						{U}
	<i>b.setBalance(bal*1.1)</i>						T, U
	<i>c.withdraw(bal/10)</i>						S, U

Figura 16.32 Carimbos de tempo nas transações T e U.

Na ordenação por carimbo de tempo de versão múltipla, que foi introduzida por Reed [1983], é mantida uma lista de versões confirmadas antigas, assim como das versões de tentativa, para cada objeto. Essa lista representa o histórico dos valores do objeto. A vantagem de usar múltiplas versões é que as operações de *leitura* que chegarem tarde demais não precisam ser rejeitadas.

Cada versão tem um carimbo de tempo de leitura registrando o maior carimbo de tempo de toda transação que a tiver lido, além de um carimbo de tempo de escrita. Como antes, quando uma operação de *escrita* é aceita, ela é direcionada para uma versão de tentativa com o carimbo de tempo de escrita da transação. Quando uma operação de *leitura* é executada, ela é direcionada para a versão com o maior carimbo de tempo de escrita menor do que o carimbo de tempo da transação. Se o carimbo de tempo da transação for maior do que o carimbo de tempo de leitura da versão que está sendo usada, o carimbo de tempo de leitura da versão será configurado com o carimbo de tempo da transação.

Quando uma leitura chegar tarde, ela poderá ter permissão para ler uma versão confirmada antiga, para que não haja necessidade de cancelar operações de *leitura* tardias. Na ordenação por carimbo de tempo de versão múltipla, as operações de *leitura* são sempre permitidas, embora talvez tenham de *esperar* que transações anteriores terminem (confirmadas ou canceladas), o que garante que as execuções sejam recuperáveis. Veja no Exercício 16.22 uma discussão sobre a possibilidade de cancelamentos em cascata. Isso tem a ver com a regra 3 das regras de conflito para ordenação por carimbo de tempo.

Não há nenhum conflito entre operações de *escrita* de diferentes transações, pois cada transação escreve sua própria versão confirmada dos objetos que acessa. Isso elimina a regra 2 das regras de conflito para ordenação por carimbo de tempo, deixando-nos com:

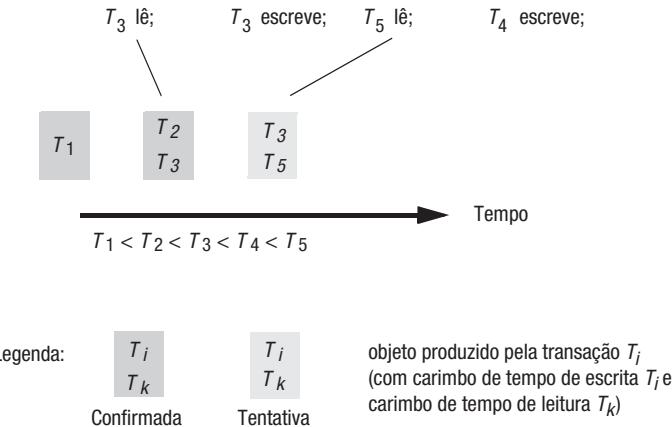


Figura 16.33 A operação de escrita tardia invalidaria uma leitura.

Regra 1. T_c não deve *escrever* objetos que foram lidos por qualquer T_i , onde $T_i > T_c$. Essa regra será violada se houver qualquer versão do objeto com carimbo de tempo de leitura $> T_c$, mas somente se essa versão tiver um carimbo de tempo de escrita menor ou igual a T_c . (Essa escrita não pode ter nenhum efeito sobre versões posteriores.)

Regra de escrita na ordenação por carimbo de tempo de versão múltipla: assim como qualquer operação de *leitura* potencialmente conflitante será direcionada para a versão mais recente de um objeto, o servidor inspeciona a versão $D_{maxAnterior}$ com o carimbo de tempo de escrita máximo menor ou igual a T_c . Temos a seguinte regra para executar uma operação de *escrita* solicitada pela transação T_c sobre o objeto D :

se (carimbo de tempo de leitura de $D_{maxAnterior} \leq T_c$)
 executa a operação de *escrita* em uma versão de tentativa de D com carimbo de tempo de escrita T_c
 senão Cancela a transação T_c

A Figura 16.33 ilustra um exemplo em que uma *escrita* é rejeitada. O objeto já possui versões confirmadas com carimbos de tempo de escrita T_1 e T_2 . O objeto recebe a seguinte sequência de requisições de operações sobre o objeto:

T_3 lê; T_3 escreve; T_5 lê; T_4 escreve.

1. T_3 solicita uma operação de *leitura*, a qual coloca um carimbo de tempo de leitura T_3 na versão de T_2 ;
2. T_3 solicita uma operação de *escrita*, a qual produz uma nova versão de tentativa com carimbo de tempo de escrita T_3 ;
3. T_5 solicita uma operação de *leitura*, que usa a versão com carimbo de tempo de escrita T_3 (o maior carimbo de tempo menor do que T_5);
4. T_4 solicita uma operação de *escrita*, a qual é rejeitada porque o carimbo de tempo de leitura T_5 da versão com carimbo de tempo de escrita T_3 é maior do que T_4 . (Se fosse permitido, o carimbo de tempo de escrita da nova versão seria T_4 . Se tal versão fosse permitida, ela invalidaria a operação de *leitura* de T_5 , que deve ter usado a versão com carimbo de tempo T_4 .)

Quando uma transação é cancelada, todas as versões que criou são excluídas. Quando uma transação é confirmada, todas as versões que criou são mantidas, mas para controlar o uso do espaço de armazenamento, as versões antigas devem ser excluídas de tempos em tempos. Embora tenha a sobrecarga do espaço de armazenamento, a ordenação por carimbo de tempo de versão múltipla permite uma concorrência considerável, não sofre do problema dos impasses e sempre permite as operações de *leitura*. Para mais informações sobre a ordenação por carimbo de tempo de versão múltipla, consulte Bernstein *et al.* [1987].

16.7 Comparação dos métodos de controle de concorrência

Descrevemos três métodos separados para controlar o acesso concomitante a dados compartilhados: travamento de duas fases restrito, métodos otimistas e ordenação por carimbo de tempo. Todos trazem consigo algumas sobrecargas no tempo e no espaço exigido, e todos limitam até certo ponto o potencial para operação concorrente.

O método da ordenação por carimbo de tempo é semelhante ao travamento de duas fases, pois ambos usam estratégias pessimistas, nas quais os conflitos entre as transações são detectados quando cada objeto é acessado. Por um lado, a ordenação por carimbo de tempo decide estaticamente a ordem da serialização – quando uma transação começa. Por outro lado, o travamento de duas fases decide dinamicamente a ordem da serialização – de acordo com a ordem em que os objetos são acessados. A ordenação por carimbo de tempo e, em particular, a ordenação por carimbo de tempo de versão múltipla, são melhores do que o travamento de duas fases restrito para transações somente de leitura. O travamento de duas fases é melhor quando as operações nas transações são predominantemente atualizações.

Um trabalho utiliza a observação de que a ordenação por carimbo de tempo é vantajosa para transações com operações predominantemente de *leitura* e que o travamento é vantajoso para transações com mais *escritas* do que *leituras*, como argumento para permitir a existência de esquemas mistos, nos quais algumas transações usam ordenação por carimbo de tempo e outras usam travamento para controle de concorrência. Os leitores que estiverem interessados no uso de métodos mistos devem ler Bernstein *et al.* [1987].

Os métodos pessimistas diferem na estratégia usada quando é detectado um acesso conflitante a um objeto. A ordenação por carimbo de tempo cancela a transação imediatamente, enquanto o travamento faz a transação esperar – mas com a possível penalidade posterior de cancelar para evitar impasse.

Quando é usado o controle de concorrência otimista, todas as transações podem prosseguir, mas algumas são canceladas quando tentam ser confirmadas ou, nas transações com validação para frente, são canceladas anteriormente. Isso resulta em uma operação relativamente eficiente quando existem poucos conflitos, mas um volume de trabalho substancial talvez tenha de ser repetido quando uma transação for cancelada.

O travamento está em uso há muitos anos em sistemas de banco de dados, mas a ordenação por carimbo de tempo foi usada no sistema de banco de dados SDD-1. Os dois métodos são usados em servidores de arquivo. Entretanto, historicamente, o método de controle de concorrência para o acesso a dados predominante em sistemas distribuídos é o travamento; por exemplo, conforme mencionado anteriormente, o *Concurrency Control Service* do CORBA é baseado inteiramente no uso de travas. Em particular, ele fornece travas hierárquicas, o que possibilita a existência de um travamento de granularidade mista em dados estruturados hierarquicamente.

Vários sistemas distribuídos de pesquisa, por exemplo Argus [Liskov 1988] e Arjuna [Shrivastava *et al.* 1991], exploraram o uso de travas semânticas, ordenação por carimbo de tempo e novas estratégias para transações longas.

Ellis *et al.* [1991] escreveram uma análise de requisitos para aplicativos multiusuário, nos quais todos os usuários esperam ver modos de visualização comuns dos objetos que estão sendo atualizados por qualquer um deles.

Muitos dos esquemas forneceram notificação de alterações feitas por outros usuários, mas isso é contrário à ideia do isolamento.

Barghouti e Kaiser [1991] escreveram uma análise do que às vezes são descritos como aplicativos de banco de dados avançados – como o CAD/CAM cooperativo e os sistemas de desenvolvimento de *software*. Em tais aplicativos, as transações duram por um longo tempo e os usuários trabalham em versões independentes dos objetos, que são retirados de um banco de dados comum e recolocados quando o trabalho está terminado. A integração das versões exige cooperação entre os usuários.

De modo semelhante, os mecanismos de controle de concorrência anteriores nem sempre são adequados para os aplicativos do século XXI, que permitem aos usuários compartilhar documentos pela Internet. Muitos destes últimos usam formas de controle de concorrência otimistas, seguidas por solução de conflito, em vez de cancelar uma de quaisquer duas operações conflitantes.

Exemplos aparecem a seguir.

Dropbox • Dropbox [www.dropbox.com] é um serviço em nuvem que fornece *backup* de arquivos e permite aos usuários compartilhar arquivos e pastas, acessando-os a partir de qualquer lugar. O Dropbox usa uma forma de controle de concorrência otimista, monitorando a consistência e evitando conflitos entre atualizações dos usuários – as quais têm granularidade de arquivos inteiros. Assim, se dois usuários fizerem atualizações concomitantes no mesmo arquivo, a primeira escrita será aceita e a segunda, rejeitada. Contudo, o Dropbox fornece um histórico de versão para permitir que os usuários mescliem suas atualizações manualmente ou restaurem versões anteriores.

Google apps • Os Google Apps [www.google.com] estão listados na Figura 21.2. Eles incluem o Google Docs, um serviço em nuvem que fornece aplicativos baseados na Web (processador de texto, planilha eletrônica e apresentação) que permitem aos usuários colaborarem uns com os outros por meio de documentos compartilhados. Se várias pessoas editarem o mesmo documento simultaneamente, elas verão as alterações umas das outras. No caso de um documento de processador de texto, os usuários podem ver os cursorres uns dos outros, e as atualizações são mostradas em nível de caracteres individuais, à medida que eles são digitados por cada participante. Fica por conta dos usuários resolver quaisquer conflitos que ocorram, mas estes geralmente são evitados, pois os usuários estão continuamente cientes das atividades uns dos outros. No caso de um documento de planilha eletrônica, os cursorres e as alterações dos usuários são exibidos e as atualizações têm granularidade de células individuais. Se dois usuários acessam a mesma célula simultaneamente, a última atualização vence.

Wikipédia • O controle de concorrência para edição é otimista, permitindo aos editores acesso concorrente às páginas Web, nas quais a primeira escrita é aceita. Um usuário que faça uma escrita subsequente verá uma tela “editar conflito” e será solicitado a resolver os conflitos.

Dynamo • O serviço de armazenamento de chave-valor da Amazon.com utiliza controle de concorrência otimista com solução de conflitos (veja o quadro a seguir).

Dynamo

Dynamo [DeCandia et al. 2007] é um dos serviços de armazenamento usados pela Amazon.com, cuja plataforma atende a dez milhões de clientes nos horários de pico, usando dezenas de milhares de servidores. Essa configuração atende a exigências muito rigorosas em termos de desempenho, confiabilidade e escalabilidade. O Dynamo foi projetado para suportar aplicativos como carrinhos de compra e as listas dos mais vendidos, que exigem somente acesso de chave primária a um valor em um banco de dados. Os dados são pesadamente replicados, com o objetivo de fornecer a escalabilidade e a disponibilidade fundamentais para esses serviços.

O Dynamo usa operações *get* e *put* simples, em vez de transações, e não oferece a garantia de isolamento especificada nas propriedades ACID. Para melhorar a disponibilidade, fornece também uma forma de consistência mais fraca – o que é aceitável nas aplicações que suporta.

São utilizados métodos otimistas para controle de concorrência. Nos casos em que as versões diferem, elas devem ser harmonizadas. No caso de um carrinho de compra, pode ser usada uma lógica programada no próprio aplicativo para mesclar versões.

Onde não for possível empregar a lógica do aplicativo, é aplicada harmonização baseada em carimbo de tempo. O Dynamo usa a regra “a última escrita vence” – a versão com o carimbo de tempo mais alto se torna a nova versão.

16.8 Resumo

As transações fornecem um meio pelo qual os clientes podem especificar sequências de operações, que são atômicas na presença de outras transações concorrentes e de falhas do servidor. O primeiro aspecto da atomicidade é obtido pela execução de transações de modo que seus efeitos sejam serialmente equivalentes. Os efeitos das transações confirmadas são registrados no meio de armazenamento permanente para que o serviço de transação possa se recuperar de falhas de processo. Para proporcionar às transações a capacidade de serem canceladas, sem efeitos colaterais prejudiciais nas outras transações, as execuções devem ser restritas – isto é, as leituras e escritas de uma transação devem ser retardadas até que as outras transações que modificaram os mesmos objetos tenham sido confirmadas ou canceladas. Para possibilitar às transações a escolha de confirmar ou cancelar, suas operações são executadas em versões de tentativa que não podem ser acessadas pelas outras transações. As versões de tentativa dos objetos são copiadas nos objetos reais e no meio de armazenamento permanente, quando uma transação é confirmada.

As transações aninhadas são formadas por meio da estruturação de transações de outras subtransações. O aninhamento é particularmente útil em sistemas distribuídos, pois permite a execução concorrente de subtransações em servidores separados. O aninhamento também tem a vantagem de permitir a recuperação independente das partes de uma transação.

Os conflitos de operação formam a base para a derivação de protocolos de controle de concorrência. Os protocolos não devem apenas garantir a serialização, mas também permitir a recuperação, usando execuções restritas para evitar os problemas associados ao cancelamento de transações, como os cancelamentos em cascata.

Três estratégias alternativas são possíveis na programação da execução de uma operação em uma transação. São elas: (1) executá-la imediatamente, (2) retardá-la ou (3) cancelá-la.

O travamento de duas fases restrito usa as duas primeiras estratégias, contando com o cancelamento apenas em caso de impasse. Ele garante a serialização por meio da ordenação das transações de acordo com o momento em que elas acessam objetos comuns. Seu principal inconveniente é que podem ocorrer impasses.

A ordenação por carimbo de tempo usa todas as três estratégias para garantir a serialização por meio da ordenação dos acessos das transações aos objetos, de acordo com a hora em que as transações começam. Esse método não pode sofrer de impasses e é vantajoso para transações somente de leitura. Entretanto, as transações devem ser canceladas quando chegam tarde demais. A ordenação por carimbo de tempo de versão múltipla é particularmente eficiente.

O controle de concorrência otimista permite que as transações prossigam, sem qualquer forma de verificação, até que sejam concluídas. As transações são validadas antes de poderem ser confirmadas. A validação para trás exige a manutenção de vários conjuntos de escrita de transações confirmadas, enquanto a validação para frente precisa ser validada em relação às transações ativas e tem a vantagem de permitir estratégias alternativas para resolver conflitos. A inanição pode ocorrer devido ao cancelamento repetido de uma transação que não consegue ser validada no controle de concorrência otimista e até na ordenação por carimbo de tempo.

Exercícios

- 16.1 O *TaskBag* (mochila de tarefas) é um serviço cuja funcionalidade é fornecer um repositório de descrições de tarefa. Ele permite que os clientes funcionando em vários computadores executem partes de um cálculo em paralelo. Um processo *mestre* coloca as descrições das subtarefas de um cálculo no *TaskBag*, e processos *operários* selecionam tarefas do *TaskBag* e as executam, retornando descrições dos resultados para o *TaskBag*. Então, o *mestre* reúne os resultados e os combina para produzir o resultado final.

O serviço *TaskBag* fornece as seguintes operações:

- | | |
|-----------------|--|
| <i>setTask</i> | permite que os clientes adicionem descrições de tarefa na mochila; |
| <i>takeTask</i> | permite que os clientes retirem descrições de tarefa da mochila. |

Um cliente faz a requisição *takeTask* quando uma tarefa não está disponível, mas poderá estar em breve. Discuta as vantagens e os inconvenientes das seguintes alternativas:

- (i) o servidor pode responder imediatamente, dizendo ao cliente para que tente novamente mais tarde;
- (ii) fazer a operação do servidor (e, portanto, do cliente) esperar até que uma tarefa se torne disponível;
- (iii) usar *callbacks*.

página 678

- 16.2 Um servidor gerencia os objetos a_1, a_2, \dots, a_n . O servidor fornece duas operações para seus clientes:

- | | |
|------------------------|--------------------------------|
| <i>read(i)</i> | retorna o valor de a_i ; |
| <i>write(i, Value)</i> | atribui <i>Value</i> a a_i . |

As transações *T* e *U* são definidas como segue:

- T*: $x = \text{read}(j); y = \text{read}(i); \text{write}(j, 44); \text{write}(i, 33);$
U: $x = \text{read}(k); \text{write}(i, 55); y = \text{read}(j); \text{write}(k, 66).$

Dê três interposições serialmente equivalentes das transações *T* e *U*.

página 685

- 16.3 Dê as interposições serialmente equivalentes de T e U no Exercício 16.2, com as seguintes propriedades: (1) que sejam restritas; (2) que não sejam restritas, mas não possam produzir cancelamentos em cascata; (3) que possam produzir cancelamentos em cascata. [página 689](#)
- 16.4 A operação *create* insere uma nova conta bancária em uma agência. As transações T e U são definidas como segue:

T : *aBranch.create("Z");*
 U : *z.deposit(10); z.deposit(20).*

Suponha que Z ainda não existe. Suponha também que a operação *deposit* não faz nada se a conta dada como argumento não existir. Considere a seguinte interposição das transações T e U :

T	U
	<i>z.deposit(10);</i>
<i>aBranch.create(Z);</i>	
	<i>z.deposit(20);</i>

Declare o saldo de Z após sua execução, nessa ordem. Essas execuções são consistentes com as execuções serialmente equivalentes de T e U ? [página 685](#)

- 16.5 Um objeto recentemente criado, como Z no Exercício 16.4, às vezes é chamado de *fantasma*. Do ponto de vista da transação U , Z não está lá inicialmente e depois aparece (como um fantasma). Explique, com um exemplo, como um fantasma poderia ocorrer quando uma conta é excluída.
- 16.6 As transações de transferência T e U são definidas como:

T : *a.withdraw(4); b.deposit(4);*
 U : *c.withdraw(3); b.deposit(3);*

Suponha que elas sejam estruturadas como pares de transações aninhadas:

T_1 : *a.withdraw(4);* T_2 : *b.deposit(4);*
 U_1 : *c.withdraw(3);* U_2 : *b.deposit(3);*

Compare o número das interposições serialmente equivalentes de T_1 , T_2 , U_1 e U_2 com o número de interposições serialmente equivalentes de T e U . Explique por que o uso dessas transações aninhadas geralmente permite um número maior de interposições serialmente equivalentes do que as não aninhadas. [página 685](#)

- 16.7 Considere os aspectos da recuperação das transações aninhadas definidas no Exercício 16.6. Suponha que uma transação *withdraw* será cancelada se a conta ficar sem fundos e que, nesse caso, a transação ascendente também será cancelada. Descreva as interposições serialmente equivalentes T_1 , T_2 , U_1 e U_2 , com as seguintes propriedades:
- que sejam restritas;
 - que não sejam restritas.

Até que ponto o critério do rigor reduz o ganho de concorrência em potencial das transações aninhadas? [página 685](#)

- 16.8 Explique por que a equivalência serial exige que, uma vez que a transação tenha liberado uma trava sobre um objeto, ela não pode obter mais nenhuma trava.

Um servidor gerencia os objetos a_1, a_2, \dots, a_n . O servidor fornece duas operações para seus clientes:

$read(i)$	retorna o valor de a_i
$write(i, Value)$	atribui $Value$ a a_i

As transações T e U são definidas como segue:

$T: x = read(i); write(j, 44);$

$U: write(i, 55); write(j, 66);$

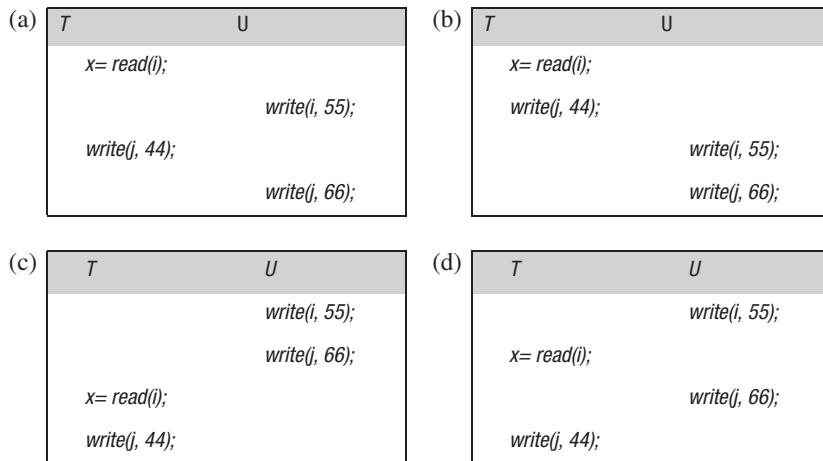
Descreva uma interposição das transações T e U na qual as travas sejam liberadas mais cedo, com o efeito de que a interposição não é serialmente equivalente. *página 693*

- 16.9 As transações T e U no servidor do Exercício 16.8 são definidas como segue:

$T: x = read(i); write(j, 44);$

$U: write(i, 55); write(j, 66);$

Os valores iniciais de a_i e a_j são 10 e 20 respectivamente. Quais das seguintes interposições são serialmente equivalentes e quais poderiam ocorrer com travamento de duas fases?



página 693

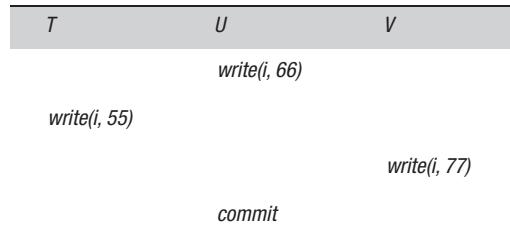
- 16.10 Considere um abrandamento do travamento de duas fases na qual as transações somente de leitura podem liberar as travas de leitura mais cedo. Uma transação somente de leitura teria recuperações consistentes? Os objetos se tornariam inconsistentes? Ilustre sua resposta com as seguintes transações T e U no servidor do Exercício 16.8:

$T: x = read(i); y = read(j);$

$U: write(i, 55); write(j, 66);$

nas quais os valores iniciais de a_i e a_j são 10 e 20. *página 690*

- 16.11 As execuções das transações são restritas caso as operações de *leitura* e de *escrita* sobre um objeto sejam atrasadas até que todas as transações que escreveram nesse objeto anteriormente tenham sido confirmadas ou canceladas. Explique como as regras de travamento da Figura 16.16 garantem execuções restritas. [página 696](#)
- 16.12 Descreva como poderia surgir uma situação irrecuperável, caso as travas de escrita fossem liberadas após a última operação de uma transação, mas antes de sua confirmação. [página 690](#)
- 16.13 Explique por que as execuções são sempre restritas, mesmo que as travas de leitura sejam liberadas após a última operação de uma transação, mas antes de sua confirmação. Forneça uma declaração melhorada da regra 2 da Figura 16.16. [página 690](#)
- 16.14 Considere um esquema de detecção de impasses para um único servidor. Descreva precisamente quando setas são adicionadas e removidas do grafo espera por. Ilustre sua resposta com relação às seguintes transações *T*, *U* e *V* no servidor do Exercício 16.8.



Quando *U* libera sua trava de escrita sobre a_i , tanto *T* como *V* estão esperando para obter travas de escrita sobre ela. Seu esquema funcionará corretamente se a trava for adquirida por *T* (a primeira a aparecer) antes de *V*? Se sua resposta for “Não”, modifique sua descrição. [página 702](#)

- 16.15 Considere as travas hierárquicas ilustradas na Figura 16.26. Quais travas devem ser postas quando um compromisso é atribuído a uma repartição de hora na semana *w*, dia *d*, na hora *t*? Em que ordem essas travas devem ser configuradas? A ordem em que elas são liberadas importa? Quais travas devem ser configuradas quando as repartições de hora de cada dia da semana *w* são vistas? Isso pode ser feito quando as travas de atribuição de um compromisso para uma repartição de hora já estão configuradas? [página 705](#)
- 16.16 Considere o controle de concorrência otimista conforme aplicado nas transações *T* e *U* definidas no Exercício 16.9. Suponha que as transações *T* e *U* estejam ativas ao mesmo tempo. Descreva o resultado em cada um dos seguintes casos:
- A requisição de *T* para confirmar vem primeiro e é usada validação para trás;
 - A requisição de *U* para confirmar vem primeiro e é usada validação para trás;
 - A requisição de *T* para confirmar vem primeiro e é usada validação para frente;
 - A requisição de *U* para confirmar vem primeiro e é usada validação para frente.
- Em cada caso, descreva a sequência em que as operações de *T* e *U* são executadas, lembrando que as escritas só são realizadas depois da validação. [página 707](#)

- 16.17 Considere a seguinte interposição das transações T e U :

T	U
<i>openTransaction</i>	<i>openTransaction</i>
<i>y = read(k);</i>	
	<i>write(i, 55);</i>
	<i>write(j, 66);</i>
	<i>commit</i>
<i>x = read(i);</i>	
	<i>write(j, 44);</i>

O resultado do controle de concorrência otimista com validação para trás é que T será cancelada, pois sua operação de leitura entra em conflito com a operação de *escrita* de U em a_i , embora as interposições sejam serialmente equivalentes. Sugira uma modificação no algoritmo que trate de tais casos.

página 707

- 16.18 Faça uma comparação das sequências de operações das transações T e U do Exercício 16.8 que são possíveis sob o travamento de duas fases (Exercício 16.9) e sob o controle de concorrência otimista (Exercício 16.16).

página 707

- 16.19 Considere o uso de ordenação por carimbo de tempo em cada um dos exemplos de interposições das transações T e U do Exercício 16.9. Os valores iniciais de a_i e a_j são 10 e 20, respectivamente, e os carimbos de tempo de leitura e escrita iniciais são t_0 . Suponha que cada transação seja aberta e obtenha um carimbo de tempo imediatamente antes de sua primeira operação; por exemplo, em (a) T e U obtêm os carimbos de tempo t_1 e t_2 respectivamente, onde $t_0 < t_1 < t_2$. Descreva, na ordem crescente de tempo, os efeitos de cada operação de T e U . Para cada operação, declare o seguinte:

- (i) se a operação pode prosseguir de acordo com a regra de escrita ou leitura;
- (ii) os carimbos de tempo atribuídos às transações ou aos objetos;
- (iii) criação de objetos de tentativa e seus valores.

Quais são os valores finais dos objetos e seus carimbos de tempo?

página 711

- 16.20 Repita o Exercício 16.19 para as seguintes interposições das transações T e U :

T	U
<i>openTransaction</i>	
	<i>openTransaction</i>
	<i>write(i, 55);</i>
	<i>write(j, 66);</i>
<i>x = read (i);</i>	
	<i>write(j, 44);</i>
	<i>commit</i>

T	U
<i>openTransaction</i>	
	<i>openTransaction</i>
	<i>write(i, 55);</i>
	<i>write(j, 66);</i>
	<i>commit</i>

página 711

- 16.21 Repita o Exercício 16.20 usando ordenação por carimbo de tempo de versão múltipla.
página 715
- 16.22 Na ordenação por carimbo de tempo de versão múltipla, as operações de *leitura* podem acessar versões de tentativa dos objetos. Dê um exemplo para mostrar como os cancelamentos em cascata podem acontecer se todas as operações de *leitura* puderem prosseguir imediatamente.
página 715
- 16.23 Quais são as vantagens e os inconvenientes da ordenação por carimbo de tempo de versão múltipla em comparação com a ordenação por carimbo de tempo normal?
página 715

17

Transações Distribuídas

- 17.1 Introdução
- 17.2 Transações distribuídas planas e aninhadas
- 17.3 Protocolos de confirmação atômica
- 17.4 Controle de concorrência em transações distribuídas
- 17.5 Impasses distribuídos
- 17.6 Recuperação de transações
- 17.7 Resumo

Este capítulo apresenta as transações distribuídas – aquelas que envolvem mais de um servidor. As transações distribuídas podem ser planas ou aninhadas.

Um protocolo de confirmação atômica é um procedimento cooperativo usado por um conjunto de servidores envolvidos em uma transação distribuída. Ele permite que os servidores cheguem a uma decisão conjunta quanto ao fato de uma transação poder ser confirmada ou cancelada. Este capítulo descreve o protocolo de confirmação de duas fases, que é o protocolo atômico de confirmação mais comumente usado.

A seção sobre controle de concorrência em transações distribuídas discute como o travamento, a ordenação por carimbo de tempo e o controle de concorrência otimista podem ser estendidos para uso com transações distribuídas.

O uso de esquemas de travas pode levar aos impasses distribuídos. Serão discutidos os algoritmos de detecção de impasse distribuído.

Os servidores que fornecem transações incluem um gerenciador de recuperação cuja função é garantir que os efeitos das transações sobre os objetos gerenciados por um servidor possam ser recuperados quando ele é substituído após uma falha. O gerenciador de recuperação salva os objetos no meio de armazenamento permanente, junto a listas de intenções e informações sobre o *status* de cada transação.

17.1 Introdução

No Capítulo 16, discutimos as transações planas e aninhadas que acessavam objetos em um único servidor. No caso geral, uma transação, seja plana ou aninhada, acessará objetos localizados em vários computadores diferentes. Usamos o termo *transação distribuída* para nos referirmos a uma transação plana ou aninhada que acessa objetos gerenciados por vários servidores.

Quando uma transação distribuída chega ao fim, a propriedade da atomicidade das transações exige que todos os servidores envolvidos confirmem a transação, ou que todos eles a cancelhem. Para se chegar a isso, um dos servidores assume o papel de coordenador, o que envolve garantir o mesmo resultado em todos os servidores. A maneira pela qual o coordenador chega a isso depende do protocolo escolhido. Um protocolo conhecido como protocolo de confirmação de duas fases é o mais comumente usado. Esse protocolo permite que os servidores se comuniquem para chegar a uma decisão conjunta quanto a confirmar ou cancelar uma transação.

O controle de concorrência em transações distribuídas é baseado nos métodos discutidos no Capítulo 16. Cada servidor aplica controle de concorrência local em seus próprios objetos, o que garante que as transações sejam serializadas de forma local. As transações distribuídas devem ser serializadas de forma global. O modo como isso é obtido varia de acordo com o fato de estarem em uso travas, ordenação por carimbo de tempo ou controle de concorrência otimista. Em alguns casos, as transações podem ser serializadas nos servidores individuais, mas, ao mesmo tempo, pode ocorrer um ciclo de dependências entre os diferentes servidores e surgir um impasse distribuído.

A recuperação de transação se preocupa em garantir que todos os objetos envolvidos nas transações sejam recuperáveis. Além disso, ela garante que os valores dos objetos reflitam todas as alterações feitas pelas transações confirmadas e nenhuma das alterações feitas pelas transações canceladas.

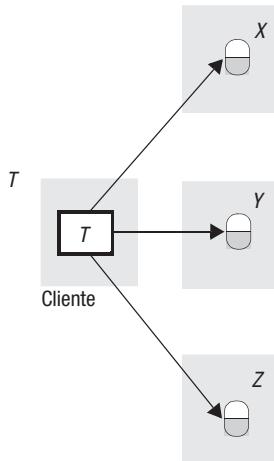
17.2 Transações distribuídas planas e aninhadas

Uma transação cliente se torna distribuída se ativa operações em vários servidores diferentes. Existem duas maneiras distintas pelas quais as transações distribuídas podem ser estruturadas: como transações planas e como transações aninhadas.

Em uma transação plana, um cliente faz requisições para mais de um servidor. Por exemplo, na Figura 17.1(a), a transação T é uma transação plana que invoca operações sobre objetos nos servidores X , Y e Z . Uma transação cliente plana conclui cada uma de suas requisições antes de passar para a próxima. Portanto, cada transação acessa objetos dos servidores em sequência. Quando os servidores usam travas, uma transação só pode estar esperando um objeto por vez.

Em uma transação aninhada, a transação de nível superior pode abrir subtransações, e cada subtransação pode abrir mais subtransações em qualquer profundidade de aninhamento. A Figura 17.1(b) mostra a transação T de um cliente, que abre duas subtransações, T_1 e T_2 , as quais acessam objetos nos servidores X e Y . As subtransações T_1 e T_2 abrem mais subtransações, T_{11} , T_{12} , T_{21} e T_{22} , as quais acessam objetos nos servidores M , N e P . No caso aninhado, as subtransações no mesmo nível podem ser executadas concomitantemente, de modo que T_1 e T_2 são concorrentes e, como invocam objetos em

(a) Transação plana



(b) Transações aninhadas

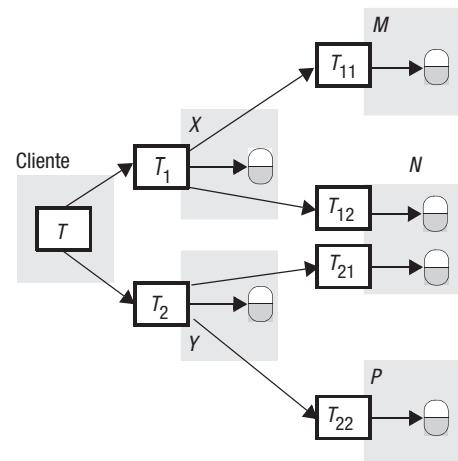


Figura 17.1 Transações distribuídas.

servidores diferentes, elas podem ser executadas em paralelo. As quatro subtransações T_{11} , T_{12} , T_{21} e T_{22} também são executadas concorrentemente.

Considere uma transação distribuída na qual um cliente transfere \$10 da conta A para C , e depois transfere \$20 de B para D . As contas A e B estão em servidores separados X e Y e as contas C e D estão no servidor Z . Se essa transação está estruturada como um conjunto de quatro transações aninhadas, como mostrado na Figura 17.2, as quatro requisições (dois depósitos [*deposit*] e dois saques [*withdraw*]) podem ser executadas em paralelo, e o efeito global pode ser obtido com melhor desempenho do que em uma transação simples, na qual as quatro operações são ativadas em sequência.

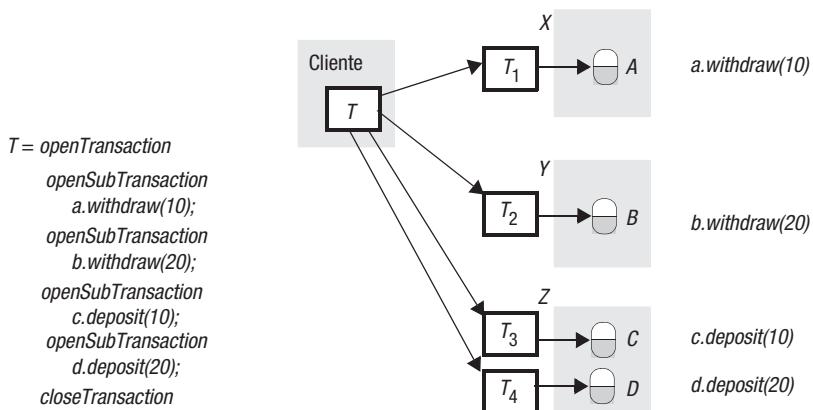
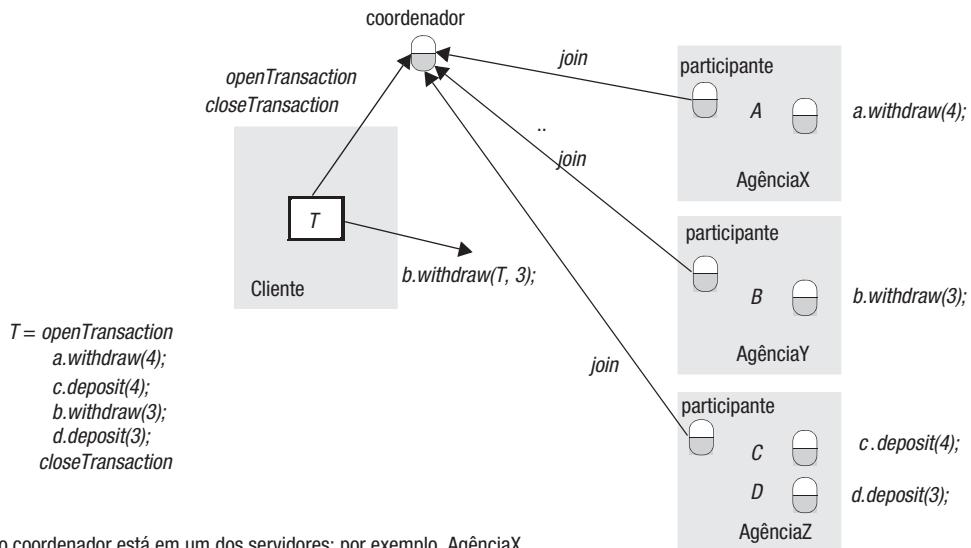


Figura 17.2 Transação bancária aninhada.



Nota: o coordenador está em um dos servidores; por exemplo, AgênciaX

Figura 17.3 Uma transação bancária distribuída.

17.2.1 O coordenador de uma transação distribuída

Os servidores que executam requisições como parte de uma transação distribuída precisam se comunicar uns com os outros para coordenar suas ações quando a transação é confirmada. Um cliente inicia uma transação enviando uma requisição de *openTransaction* para um coordenador em qualquer servidor, conforme descrito na Seção 16.2. O coordenador contatado executa a requisição *openTransaction* e retorna para o cliente o identificador de transação resultante. Os identificadores das transações distribuídas devem ser exclusivos dentro de um sistema distribuído. Uma maneira simples de obter isso é fazer com que um TID contenha duas partes: o identificador (por exemplo, um endereço IP) do servidor que o criou e um número exclusivo para o servidor.

O coordenador que abriu a transação torna-se o *coordenador* da transação distribuída e, no final, é responsável por confirmá-la ou cancelá-la. Cada um dos servidores que gerencia um objeto acessado por uma transação é um *participante* da transação. Cada participante é responsável por rastrear todos os objetos recuperáveis envolvidos na transação nesse servidor. Os participantes são responsáveis por cooperar com o coordenador na execução do protocolo de confirmação.

Durante o andamento da transação, o coordenador registra uma lista de referências nos participantes, e cada participante registra uma referência no coordenador.

A interface para *Coordinator*, mostrada na Figura 16.3, deve fornecer um método adicional, *join*, usado quando um novo participante se junta à transação:

join(Trans, referência ao participante)

Informa ao coordenador que um novo participante entrou na transação *Trans*.

O coordenador registra o novo participante em sua lista de participantes. O fato de o coordenador conhecer todos os participantes, e cada participante conhecer o coordenador, permitirá que eles reúnam as informações que serão necessárias no momento da confirmação.

A Figura 17.3 mostra um cliente cuja transação bancária (plana) envolve as contas *A*, *B*, *C* e *D*, nos servidores AgênciaX, AgênciaY e AgênciaZ. A transação do cliente, *T*, transfere \$4 da conta *A* para a conta *C*, e depois transfere \$3 da conta *B* para a conta *D*. A transação descrita à esquerda é expandida para mostrar que *openTransaction* e *closeTransaction* são direcionados para o coordenador, o qual estará situado em um dos servidores envolvidos na transação. Cada servidor é mostrado com um participante, o qual entra na transação invocando o método *join* no coordenador. Quando o cliente invoca um dos métodos na transação, por exemplo, *b.withdraw(T, 3)*, o objeto que recebe a invocação (*B* em AgênciaY, neste caso) informa ao seu objeto participante que o objeto pertence à transação *T*. Se ainda não tiver informado ao coordenador, o objeto participante usa a operação *join* para fazer isso. Neste exemplo, mostramos o identificador de transação sendo passado como um argumento adicional, para que o destinatário possa passá-lo para o coordenador. Quando o cliente chama *closeTransaction*, o coordenador tem referências para todos os participantes.

Note que é possível um participante chamar *abortTransaction* no coordenador se, por algum motivo, não for capaz de continuar com a transação.

17.3 Protocolos de confirmação atômica

Os protocolos de confirmação (*commit*) de transação foram inventados no início dos anos 70, e o protocolo de confirmação de duas fases apareceu em Gray [1978]. A atomicidade das transações exige que, quando uma transação distribuída chega ao fim, todas as suas operações sejam executadas ou que nenhuma delas seja executada. No caso de uma transação distribuída, o cliente solicita as operações em mais de um servidor. Uma transação chega ao fim quando o cliente solicita que ela seja confirmada ou cancelada. Uma maneira simples de concluir a transação de maneira atômica é fazer com que o coordenador comunique o pedido de confirmação, ou cancelamento, para todos os participantes da transação e fique repetindo o pedido até que todos tenham reconhecido que foram levados adiante. Esse é um exemplo de *protocolo de confirmação atômica de uma fase*.

Esse protocolo de confirmação atômica de uma fase é inadequado, pois, quando o cliente solicita uma confirmação, ele não permite que um servidor tome a decisão unilateral de cancelar a transação. Os motivos que impedem um servidor de confirmar sua parte de uma transação geralmente estão relacionados a problemas de controle de concorrência. Por exemplo, se estiverem sendo usadas travas, a solução de um impasse poderá levar ao cancelamento de uma transação sem que o cliente saiba, a não ser que faça outro pedido para o servidor. Se estiver em uso o controle de concorrência otimista, a falha da validação em um servidor o faria decidir cancelar a transação. O coordenador pode não saber quando um servidor falhou e foi substituído durante o andamento de uma transação distribuída – tal servidor precisará cancelar a transação.

O *protocolo de confirmação de duas fases* é projetado de forma a permitir que qualquer participante cancele sua parte de uma transação. Devido ao requisito da atomicidade, se uma parte de uma transação for cancelada, a transação inteira também deverá ser cancelada. Na primeira fase do protocolo, cada participante vota na confirmação ou no cancelamento de uma transação. Quando um participante tiver votado na confirmação de uma transação, ele não poderá cancelá-la. Portanto, antes que um participante vote na confirmação de uma transação, ele deve garantir que poderá executar sua parte do protocolo de confirmação, mesmo que falhe e seja substituído nesse

meio-tempo. Diz-se que um participante de uma transação está em um estado *preparado* para uma transação se puder confirmá-la. Para garantir isso, cada participante salva, no meio de armazenamento permanente, todos os objetos que tiver alterado na transação, junto com seu status – preparado.

Na segunda fase do protocolo, todos os participantes da transação tomam uma decisão conjunta. Se um participante votar pelo cancelamento, a decisão deverá ser o cancelamento da transação. Se todos os participantes votarem na confirmação, a decisão será a de confirmar a transação.

O problema é garantir que todos os participantes votem, e que todos cheguem à mesma decisão. Se nenhum erro ocorrer, isso é muito simples, mas o protocolo deve funcionar corretamente mesmo quando alguns dos servidores falham, mensagens são perdidas ou servidores são temporariamente incapazes de se comunicar uns com os outros.

Modelo de falha dos protocolos de confirmação • A Seção 16.1.2 apresentou um modelo de falha para transações que se aplica igualmente ao protocolo de confirmação de duas fases (ou qualquer outro). Os protocolos de confirmação são feitos para funcionar em um sistema assíncrono, no qual os servidores podem falhar e mensagens podem ser perdidas. Presume-se que um protocolo de requisição e resposta subjacente remova as mensagens corrompidas e replicadas. Não existem falhas bizantinas – ou os servidores falham ou obedecem às mensagens enviadas.

O protocolo de confirmação de duas fases é um exemplo de protocolo para chegar a um consenso. O Capítulo 15 declarou que o consenso não pode ser atingido em um sistema assíncrono, caso os processos às vezes falhem. Entretanto, o protocolo de confirmação de duas fases chega ao consenso sob essas condições. Isso porque as falhas por colapso dos processos são mascaradas pela substituição de um processo falho por um novo processo, cujo estado é configurado a partir das informações gravadas no meio de armazenamento permanente e das informações mantidas por outros processos.

17.3.1 O protocolo de confirmação de duas fases

Durante o andamento de uma transação, não há nenhuma comunicação entre o coordenador e os participantes, a não ser os participantes informando o coordenador quando entram na transação. O pedido de um cliente para confirmar (ou cancelar) uma transação é direcionado ao coordenador. Se o cliente solicitar *abortTransaction*, ou se a transação for cancelada por um dos participantes, o coordenador informará aos participantes imediatamente. O protocolo de confirmação de duas fases é utilizado quando o cliente pede ao coordenador para que confirme a transação.

Na primeira fase do protocolo de confirmação de duas fases, o coordenador pergunta a todos os participantes se eles estão preparados para confirmar; e, na segunda, o coordenador diz a eles para que confirmem (ou cancelem) a transação. Se um participante puder confirmar sua parte de uma transação, ele concordará assim que tiver registrado as alterações e seu status no meio de armazenamento permanente – e estiver preparado para confirmar. O coordenador de uma transação distribuída se comunica com os participantes para executar o protocolo de confirmação de duas fases por intermédio das operações resumidas na Figura 17.4. Os métodos *canCommit*, *doCommit* e *doAbort* estão na interface do participante. Os métodos *haveCommitted* e *getDecision* estão na interface do coordenador.

O protocolo de confirmação de duas fases consiste em uma fase de votação e uma fase de conclusão, como se vê na Figura 17.5. No final da etapa (2), o coordenador e to-

canCommit?(trans) → Sim / Não

Chamada do coordenador ao participante para perguntar se ele pode confirmar uma transação. O participante responde com seu voto.

doCommit(trans)

Chamada do coordenador ao participante para dizer a ele para que confirme sua parte de uma transação.

doAbort(trans)

Chamada do coordenador ao participante para dizer a ele para que cancele sua parte de uma transação.

haveCommitted(trans, participante)

Chamada do participante ao coordenador para confirmar que efetivou a transação.

getDecision(trans) → Sim / Não

Chamada do participante ao coordenador para solicitar a decisão em uma transação, após ter votado em *Sim*, mas ainda não ter recebido resposta devido a algum atraso. Usada para se recuperar de uma falha de servidor ou de mensagens retardadas.

Figura 17.4 Operações do protocolo de confirmação de duas fases.

dos os participantes que votaram *Sim* estão preparados para confirmar. No final da etapa (3), a transação é efetivamente concluída. Na etapa (3a), o coordenador e os participantes estão confirmados; portanto, o coordenador pode informar a decisão de confirmar para o cliente. Em (3b), o coordenador informa a decisão de cancelar para o cliente.

Na etapa (4), os participantes indicam que confirmaram, para que o coordenador saiba quando a informação que registrou sobre a transação não é mais necessária.

Esse protocolo aparentemente simples poderia não funcionar devido à falha de um ou mais servidores ou devido a uma interrupção na comunicação entre os servidores. Para lidar com a possibilidade de falha, cada servidor salva informações relacionadas ao protocolo de confirmação de duas fases no meio de armazenamento permanente. Essas informações podem ser recuperadas por um novo processo que seja iniciado para substituir um servidor falho. Os aspectos da recuperação de transações distribuídas serão discutidos na Seção 17.6.

A troca de informações entre o coordenador e os participantes pode fracassar quando um dos servidores falhar ou quando mensagens forem perdidas. São usados tempos limites (*timeouts*) para impedir que os processos sejam bloqueados para sempre. Quando decorre o tempo limite de um processo, ele deve executar uma ação apropriada. Para possibilitar isso, o protocolo inclui um tratador que executa uma ação sempre que o tempo limite for atingido para cada etapa em que um processo pode ser bloqueado. Essas ações são projetadas de modo a admitir o fato de que, em um sistema assíncrono, o fato de atingir um tempo limite pode não implicar necessariamente que um servidor falhou.

Ações de tempo limite (timeout) no protocolo de confirmação de duas fases • Existem diversos estágios nos quais o coordenador, ou um participante, não pode dar andamento em sua parte do protocolo, até que receba outra requisição ou uma resposta uns dos outros.

Considere primeiro a situação em que um participante votou *Sim* e está esperando que o coordenador informe o resultado dos votos dizendo a ele para que confirme ou cancele a transação. Veja a etapa (2) na Figura 17.6. Tal participante está *incerto* do resul-

Fase 1 (fase de votação):

1. O coordenador envia uma requisição *canCommit?* para cada um dos participantes da transação.
2. Quando um participante recebe uma requisição *canCommit?*, ele responde com seu voto (*Sim* ou *Não*) para o coordenador. Antes de votar em *Sim*, ele se prepara para confirmar, salvando objetos no armazenamento permanente. Se o voto for *Não*, o participante cancelará imediatamente.

Fase 2 (conclusão de acordo com o resultado do voto):

3. O coordenador reúne os votos (incluindo o seu próprio).
 - (a) Se não houver falhas, e todos os votos forem *Sim*, o coordenador decide confirmar a transação e envia uma requisição *doCommit* para cada um dos participantes.
 - (b) Caso contrário, o coordenador decide cancelar a transação e envia requisições *doAbort* para todos os participantes que votaram *Sim*.
4. Os participantes que votaram *Sim* estão esperando por uma requisição *doCommit* ou *doAbort* do coordenador. Quando um participante recebe uma dessas mensagens, ele age correspondentemente e, em caso de confirmação, faz uma chamada de *haveCommitted* para o coordenador.

Figura 17.5 O protocolo de confirmação de duas fases.

tado, e não pode prosseguir até que receba o resultado dos votos do coordenador. O participante não pode decidir unilateralmente o que fazer em seguida e, nesse meio-tempo, os objetos usados por sua transação não podem ser liberados para uso por outras transações. O participante faz uma requisição *getDecision* para o coordenador, para determinar o resultado da transação. Quando recebe a resposta, ele continua o protocolo na etapa (4) da Figura 17.5. Se o coordenador tiver falhado, o participante não poderá obter a decisão até que o coordenador seja substituído, o que pode resultar em grandes atrasos para os participantes no estado de *incerteza*.

Estratégias alternativas estão disponíveis para os participantes chegarem a uma decisão cooperativamente, em vez de entrarem em contato com o coordenador. Essas estratégias têm a vantagem de poderem ser usadas quando o coordenador tiver falhado. Consulte o Exercício 17.5 e Bernstein *et al.* [1987] para mais detalhes. Entretanto,

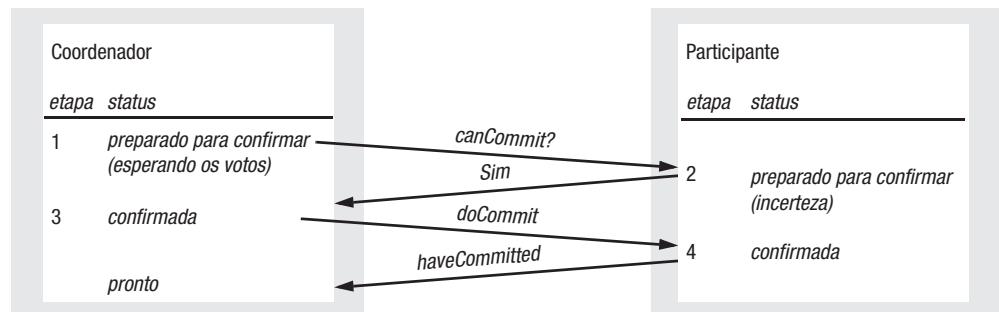


Figura 17.6 Comunicação no protocolo de confirmação de duas fases.

mesmo com um protocolo cooperativo, se todos os participantes estiverem no estado de *incerteza*, não poderão chegar a uma decisão até que o coordenador ou um participante com esse conhecimento esteja disponível.

Outro ponto no qual um participante pode ser retardado é quando tiver executado todas as requisições de seu cliente na transação, mas ainda não tiver recebido uma chamada de *canCommit?* do coordenador. Quando o cliente envia a requisição *close-Transaction* para o coordenador, um participante só pode detectar tal situação se notar que não tinha uma requisição em uma transação específica por um longo tempo; por exemplo, por um período de tempo limite sobre uma trava. Como nenhuma decisão foi tomada nesse estágio, o participante pode decidir cancelar unilateralmente, após algum período de tempo.

O coordenador pode ser retardado quando estiver esperando pelos votos dos participantes. Como ele ainda não decidiu o destino da transação, pode decidir cancelá-la após algum período de tempo. Ele deve, então, anunciar *doAbort* aos participantes que já enviaram seus votos. Alguns participantes lentos podem tentar votar *Sim* depois disso, mas seus votos serão ignorados e eles entrarão no estado de *incerteza*, conforme descrito anteriormente.

Desempenho do protocolo de confirmação de duas fases • Desde que tudo corra bem – isto é, que o coordenador, os participantes e a comunicação entre eles não falhem –, o protocolo de confirmação de duas fases envolvendo N participantes pode ser concluído com N mensagens e respostas *canCommit?*, seguidas de N mensagens *doCommit*. Ou seja, o custo nas mensagens é proporcional a $3N$ e o custo no tempo é o de três rodadas de mensagens. As mensagens *haveCommitted* não são contadas no custo estimado do protocolo, que pode funcionar corretamente sem elas – sua função é permitir que os servidores excluam informações velhas do coordenador.

No pior caso, pode haver, arbitrariamente, muitas falhas de servidor e de comunicação durante o protocolo de confirmação de duas fases. Entretanto, o protocolo é feito para tolerar uma sucessão de falhas (falhas de servidor ou mensagens perdidas) e é garantido que será concluído, embora não seja possível especificar um limite de tempo dentro do qual ele terminará.

Conforme mencionado na seção sobre tempos limites, o protocolo de confirmação de duas fases pode causar atrasos consideráveis para participantes no estado de *incerteza*. Esses atrasos ocorrem quando o coordenador falhou e não pode responder às requisições *getDecision* dos participantes. Mesmo que um protocolo cooperativo permita aos participantes fazerem requisições *getDecision* para outros participantes, ocorrerão atrasos se todos os participantes ativos estiverem *incertos*.

Foram projetados protocolos de confirmação de três fases para diminuir tais atrasos. Eles são mais dispendiosos no número de mensagens e no número de rodadas exigidas para o caso normal (livre de falhas). Para uma descrição dos protocolos de confirmação de três fases, consulte o Exercício 17.2 e Bernstein *et al.* [1987].

17.3.2 Protocolo de confirmação de duas fases para transações aninhadas

A transação mais externa em um conjunto de transações aninhadas é chamada de *transação de nível superior*. As outras transações são chamadas de *subtransações*. Na Figura 17.1(b), T é a transação de nível superior e T_1 , T_2 , T_{11} , T_{12} , T_{21} e T_{22} são subtransações. T_1 e T_2 são transações descendentes de T , que é referida como sua *ascendente*. Analogamente,

openSubTransaction(trans) → subTrans

Abre uma nova subtransação cuja ascendente é *trans* e retorna um identificador de subtransação exclusivo.

getStatus(trans) → confirmada, cancelada, provisória

Pede ao coordenador para que informe o status da transação *trans*. Retorna valores representando uma das seguintes opções: *confirmada, cancelada, provisória*.

Figura 17.7 Operações no coordenador de transações aninhadas.

T_{11} e T_{12} são transações descendentes de T_1 , e T_{21} e T_{22} são transações descendentes de T_2 . Cada subtransação começa depois de sua ascendente e termina antes dela. Assim, por exemplo, T_{11} e T_{12} começam depois de T_1 e terminam antes dela.

Quando uma subtransação termina, ela toma uma decisão, independente de ser confirmada provisoriamente ou cancelada. Uma confirmação provisória é diferente de estar preparado para confirmar: não é feito uma escrita no meio de armazenamento permanente. Se subsequentemente o servidor falhar, seu substituto não poderá confirmar. Após todas as subtransações terem terminado, as que foram provisoriamente confirmadas participam de um protocolo de confirmação de duas fases, no qual os servidores das subtransações confirmadas provisoriamente expressam sua intenção de confirmar e aquelas com uma ancestral cancelada serão canceladas. O fato de estar preparada para confirmação garante que uma subtransação poderá ser confirmada, enquanto uma confirmação provisória significa apenas que ela terminou corretamente – e provavelmente concordará em ser confirmada, quando for solicitada a fazer isso.

Um coordenador de uma subtransação fornecerá uma operação para abrir uma subtransação junto a uma operação que permite ao coordenador da subtransação perguntar se sua ascendente já foi confirmada ou cancelada, como mostra a Figura 17.7.

Um cliente inicia um conjunto de transações aninhadas abrindo uma transação de nível superior a uma operação *openTransaction*, a qual retorna um identificador de transação para a transação de nível superior. O cliente inicia uma subtransação ativando a operação *openSubTransaction*, cujo argumento especifica sua transação ascendente. A nova subtransação se junta automaticamente à transação ascendente e é retornado um identificador de transação para a subtransação.

O identificador de uma subtransação deve ser uma extensão do TID de sua ascendente construído de tal modo que o identificador da transação de nível superior, ou ascendente dessa subtransação, possa ser determinado a partir de seu próprio identificador de transação. Além disso, todos os identificadores de subtransação devem ser globalmente exclusivos. O cliente termina um conjunto de transações aninhadas invocando *closeTransaction*, ou *abortTransaction*, no coordenador da transação de nível superior.

Nesse meio-tempo, cada uma das transações aninhadas executa suas operações. Quando elas tiverem terminado, o servidor que estiver gerenciando a subtransação registrará informações quanto ao fato da subtransação ter sido confirmada provisoriamente ou ter sido cancelada. Note que, se sua ascendente for cancelada, a subtransação também será obrigada a ser cancelada.

Lembre-se, do Capítulo 16, que uma transação ascendente – incluindo uma transação de nível superior – pode ser confirmada mesmo que uma de suas subtransações descendentes tenha sido cancelada. Em tais casos, a transação ascendente será programada para executar ações diferentes, de acordo com o fato de uma subtransação ser

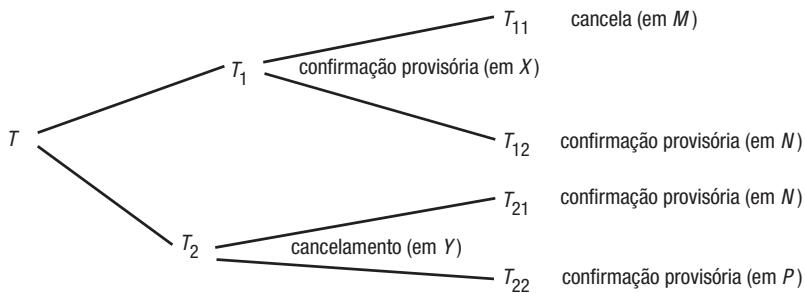


Figura 17.8 A transação T decide se vai ser efetivada.

confirmada ou cancelada. Por exemplo, considere uma transação bancária projetada para executar todos os pedidos de posição de uma agência em um dia específico. Essa transação é expressa como várias subtransações *Transfer* aninhadas, cada uma das quais decomposta em subtransações *deposit* e *withdraw* aninhadas. Supomos que, quando uma conta está sem fundos, *withdraw* é cancelada e, então, a subtransação *Transfer* correspondente é cancelada. No entanto, não há necessidade de cancelar todos os pedidos de posição apenas porque uma subtransação *Transfer* foi cancelada. Em vez de cancelar, a transação de nível superior notará as subtransações *Transfer* que foram canceladas e executará as ações apropriadas.

Considere a transação de nível superior T e suas subtransações, mostradas na Figura 17.8, que é baseada na Figura 17.1(b). Cada subtransação foi provisoriamente confirmada ou cancelada. Por exemplo, T_{12} foi provisoriamente confirmada e T_{11} foi cancelada, mas o destino de T_{12} depende de sua ascendente T_1 e, finalmente, da transação de nível superior, T . Embora T_{21} e T_{22} tenham sido provisoriamente confirmadas, T_2 foi cancelada e isso significa que T_{21} e T_{22} também devem ser canceladas. Suponha que T decida ser confirmada, apesar do fato de T_2 ter sido cancelada, e que T_1 também decida ser confirmada, embora T_{11} tenha sido cancelada.

Quando uma transação de nível superior termina, seu coordenador executa um protocolo de confirmação de duas fases. O único motivo para uma subtransação participante ser incapaz de terminar é se ela tiver falhado desde que concluiu sua confirmação provisória. Lembre-se de que, quando cada subtransação foi criada, ela se *juntou* à sua transação ascendente. Portanto, o coordenador de cada transação ascendente tem uma lista de suas subtransações descendentes. Quando uma transação aninhada é confirmada provisoriamente, ela informa seu próprio status, e o de suas descendentes, para sua ascendente. Quando uma transação aninhada é cancelada, ela informa o cancelamento apenas para sua ascendente, sem fornecer nenhuma informação sobre suas descendentes. Finalmente, a transação de nível superior recebe uma lista de todas as subtransações da árvore, junto ao status de cada uma. As descendentes das subtransações canceladas são omitidas dessa lista.

As informações mantidas em cada coordenador, no exemplo da Figura 17.8, aparecem na Figura 17.9. Note que T_{12} e T_{21} compartilham um coordenador, pois ambas são executadas no servidor N . Quando a subtransação T_2 foi cancelada, ela informou o fato para sua ascendente, T , mas sem passar nenhuma informação sobre suas subtransações T_{21} e T_{22} . Uma subtransação fica *órfã* se uma de suas ancestrais é cancelada, explicitamente ou porque seu coordenador falhou. Em nosso exemplo, as subtransações T_{21} e T_{22}

Coordenador da transação	Transações descendentes	Participante	Lista de confirmações provisórias	Lista de cancelamentos
T	T_1, T_2	sim	T_1, T_{12}	T_{11}, T_2
T_1	T_{11}, T_{12}	sim	T_1, T_{12}	T_{11}
T_2	T_{21}, T_{22}	não (cancelada)		T_2
T_{11}		não (cancelada)		T_{11}
T_{12}, T_{21}		T_{12} mas não T_{21}^*	T_{21}, T_{12}	
T_{22}		não (ascendente cancelada)	T_{22}	

* T_{21} : ascendente foi cancelada

Figura 17.9 Informações mantidas pelos coordenadores das transações aninhadas.

ficaram órfãs porque suas ascendentes foram canceladas sem passar informações sobre elas para a transação de nível superior. Entretanto, o coordenador pode fazer perguntas sobre o *status* de suas ascendentes, usando a operação *getstatus*. Uma subtransação provisoriamente confirmada de uma transação cancelada deve ser cancelada, sem levar em consideração se a transação de nível superior será confirmada.

A transação de nível superior desempenha o papel de coordenador no protocolo de confirmação de duas fases, e a lista de participantes consiste nos coordenadores de todas as subtransações da árvore que foram confirmadas provisoriamente, mas não têm ancestrais canceladas. Neste estágio, a lógica do programa determinou que a transação de nível superior deve tentar confirmar o que resta, apesar de algumas subtransações canceladas. Na Figura 17.8, os coordenadores de T , T_1 e T_{12} são participantes e serão solicitados a votar no resultado. Se eles votarem na confirmação, deverão *preparar* suas transações, salvando o estado dos objetos no armazenamento permanente. Esse estado é registrado como pertencente à transação de nível superior da qual formará uma parte. O protocolo de confirmação de duas fases pode ser executado de maneira hierárquica ou de maneira plana.

A segunda fase do protocolo de confirmação de duas fases é igual ao do caso não aninhado. O coordenador reúne os votos e depois informa o resultado aos participantes. Quando ela terminar, o coordenador e os participantes terão confirmado ou cancelado suas transações.

Protocolo hierárquico de confirmação de duas fases • Nesta estratégia, o protocolo de confirmação de duas fases se torna um protocolo aninhado de vários níveis. O coordenador da transação de nível superior se comunica com os coordenadores das subtransações dos quais ele é o ascendente imediato. Ele envia mensagens de *canCommit?* para cada um destes, os quais, por sua vez, as passam para os coordenadores de suas transações descendentes (e assim por diante, descendo na árvore). Cada participante reúne as respostas de seus descendentes, antes de responder ao seu ascendente. Em nosso exemplo, T envia mensagens de *canCommit?* para o coordenador de T_1 , e depois T_1 envia mensagens de *canCommit?* para T_{12} , perguntando sobre as descendentes de T_1 . O protocolo não inclui os coordenadores de transações como a T_2 , que foi cancelada. A Figura 17.10 mostra os argumentos exigidos para *canCommit?*. O primeiro argumento é o TID da transação de nível superior, para ser usado na preparação dos dados. O segundo argumento é o TID do participante que está fazendo a chamada de *canCommit?*. O participante que está

canCommit?(trans, subTrans) → Sim / Não

Chama um coordenador para perguntar ao coordenador da subtransação descendente se ele pode confirmar uma subtransação *subTrans*. O primeiro argumento, *trans*, é o identificador da transação de nível superior. O participante responde com seu voto *Sim / Não*.

Figura 17.10 *canCommit?* para o protocolo hierárquico de confirmação de duas fases.

recebendo a chamada procura em sua lista de transações qualquer transação, ou subtransação, provisoriamente confirmada correspondente ao TID do segundo argumento. Por exemplo, o coordenador de T_{12} também é o coordenador de T_{21} , pois elas são executadas no mesmo servidor, mas quando ele receber a chamada de *canCommit?*, o segundo argumento será T_1 e ele tratará apenas com T_{12} .

Se um participante encontra quaisquer subtransações que correspondam ao segundo argumento, ele prepara os objetos e responde com um voto *Sim*. Se não encontra nenhuma, ela deve ter falhado desde o momento em que executou a subtransação e responde com um voto *Não*.

Protocolo plano de confirmação de duas fases • Nesta estratégia, o coordenador da transação de nível superior envia mensagens de *canCommit?* para os coordenadores de todas as subtransações da lista de confirmações provisórias. Em nosso exemplo, ele envia a mensagem para os coordenadores de T_1 e T_{12} . Durante o protocolo de confirmação, os participantes se referem à transação por meio de seu TID de nível superior. Cada participante procura em sua lista de transações qualquer transação, ou subtransação, correspondente a esse TID. Por exemplo, o coordenador de T_{12} também é o coordenador de T_{21} , pois elas são executadas no mesmo servidor (N).

Infelizmente, isso não fornece informações suficientes para permitir ações corretas por parte dos participantes, como o coordenador no servidor N , que tem uma mistura de subtransações provisoriamente confirmadas e canceladas. Se o coordenador de N fosse solicitado a confirmar T , acabaria confirmando T_{12} e T_{21} , pois, de acordo com suas informações locais, ambas foram confirmadas provisoriamente. Isso está errado no caso de T_{21} , porque sua ascendente, T_2 , foi cancelada. Para considerar tais casos, a operação *canCommit?* do protocolo de confirmação plano tem um segundo argumento que fornece uma lista de subtransações canceladas, como mostra a Figura 17.11. Um participante pode confirmar descendentes da transação de nível superior, desde que elas não tenham ancestrais canceladas. Quando um participante recebe uma requisição de *canCommit?*, ele faz o seguinte:

- Se o participante tiver quaisquer transações provisoriamente confirmadas que sejam descendentes da transação de nível superior, *trans*:
 - ele verifica se elas não têm ancestrais canceladas na lista *abortList*. Então, prepara-se para confirmar (registrando a transação e seus objetos no armazenamento permanente);

canCommit?(trans, abortList) → Sim / Não

Chamada do coordenador ao participante para perguntar se ele pode confirmar uma transação. O participante responde com seu voto *Sim / Não*.

Figura 17.11 *canCommit?* para o protocolo plano de confirmação de duas fases.

- aquelas que têm ancestrais canceladas também são canceladas;
 - envia um voto *Sim* para o coordenador.
- Se o participante não tem uma descendente provisoriamente confirmada da transação de nível superior, ela deve ter falhado desde o momento em que ele executou a subtransação e ele envia um voto *Não* para o coordenador.

Uma comparação das duas estratégias • O protocolo hierárquico tem a vantagem de que, em cada estágio, o participante só precisa procurar subtransações de sua ascendente imediata, enquanto o protocolo plano precisa ter a lista de cancelamentos para eliminar transações cujas ascendentes foram canceladas. Moss [1985] preferia o algoritmo plano, pois ele permite que o coordenador da transação de nível superior se comunique diretamente com todos os participantes, enquanto a variante hierárquica envolve a passagem de uma série de mensagens para cima e para baixo da árvore, em estágios.

Ações para tempo limite (timeout) • O protocolo de confirmação de duas fases para transações aninhadas pode fazer com que o coordenador, ou um participante, seja retardado nas mesmas três etapas da versão não aninhada. Existe uma quarta etapa na qual as subtransações podem ser retardadas. Considere as subtransações descendentes provisoriamente confirmadas de subtransações canceladas: elas não são necessariamente informadas do resultado da transação. Em nosso exemplo, T_{22} é tal subtransação – ela foi provisoriamente confirmada, mas como sua ascendente T_2 foi cancelada, ela não se torna uma participante. Para lidar com tais situações, toda subtransação que não tiver recebido uma mensagem de *canCommit?* fará uma pergunta após um período de tempo limite. A operação *getstatus* na Figura 17.7 permite que uma subtransação pergunte se sua ascendente foi confirmada ou cancelada. Para tornar tais perguntas possíveis, os coordenadores das subtransações canceladas precisam sobreviver por um período de tempo. Se uma subtransação órfã não puder entrar em contato com sua ascendente, ela será cancelada.

17.4 Controle de concorrência em transações distribuídas

Cada servidor gerencia um conjunto de objetos e é responsável por garantir que eles permaneçam consistentes quando acessados por transações concorrentes. Portanto, cada servidor é responsável por aplicar controle de concorrência em seus próprios objetos. Os membros de um conjunto de servidores de transações distribuídas são conjuntamente responsáveis por garantir que elas sejam executadas de maneira serialmente equivalente.

Isso implica que, se a transação T for executada antes da transação U em seu acesso conflitante aos objetos de um dos servidores, então elas deverão estar nessa ordem em todos os servidores cujos objetos forem acessados de maneira conflitante por T e U .

17.4.1 Travamento

Em uma transação distribuída, as travas (*locks*) sobre um objeto são mantidas de maneira local (no mesmo servidor). O gerenciador de travas local pode decidir se vai conceder a trava ou se vai fazer a transação solicitante esperar. Entretanto, ele não pode liberar nenhuma trava até saber se a transação foi confirmada, ou cancelada, em todos os servidores envolvidos na transação. Quando as travas são usadas para controle de concorrência, os objetos permanecem travados e ficam indisponíveis para outras transações durante o

protocolo de confirmação atômica, embora uma transação cancelada libere suas travas após a fase 1 do protocolo.

Como os gerenciadores de travas de diferentes servidores configuram as travas independentemente uns dos outros, é possível que diferentes servidores imponham diferentes ordens nas transações. Considere a seguinte interposição das transações T e U nos servidores X e Y :

T			U		
$write(A)$	em X	trava A			
			$write(B)$	em Y	trava B
$read(B)$	em Y	espera por U			
			$read(A)$	em X	espera por T

A transação T trava o objeto A no servidor X , e a transação U trava o objeto B no servidor Y . Depois disso, T tenta acessar B no servidor Y e espera pela trava de U . Analogamente, a transação U tenta acessar A no servidor X e precisa esperar pela trava de T . Portanto, temos T antes de U em um servidor e U antes de T no outro. Essas ordens diferentes podem levar a dependências cíclicas entre as transações e surge uma situação de impasse distribuído. A detecção e a solução de impasses distribuídos serão discutidas na Seção 17.5. Quando um impasse é detectado, uma transação é cancelada para resolvê-lo. Neste caso, o coordenador será informado e cancelará a transação nos participantes envolvidos.

17.4.2 Controle de concorrência pela ordenação por carimbo de tempo

Em uma única transação de servidor, o coordenador publica um carimbo de tempo (*timestamp*) único para cada transação, quando ela começa. A equivalência serial é imposta por intermédio da confirmação das versões dos objetos na ordem dos carimbos de tempo das transações que as acessaram. Nas transações distribuídas, é exigido que cada coordenador publique carimbos de tempo globalmente exclusivos. Um carimbo de tempo de transação globalmente exclusiva é publicado para o cliente pelo primeiro coordenador acessado por uma transação. O carimbo de tempo da transação é passado para o coordenador em cada servidor cujos objetos executam uma operação na transação.

Os servidores de transações distribuídas são conjuntamente responsáveis por garantir que elas sejam executadas de maneira serialmente equivalente. Por exemplo, se a versão de um objeto acessado pela transação U é confirmada após a versão acessada por T em um servidor e, então, se T e U acessam o mesmo objeto em outros servidores, elas devem confirmá-los na mesma ordem. Para obter a mesma ordem em todos os servidores, os coordenadores devem concordar com a ordenação de seus carimbos de tempo. Um carimbo de tempo consiste em um par \langle carimbo de tempo local, id-servidor \rangle . A ordem acordada dos pares de carimbos de tempo é baseada em uma comparação na qual a parte de identificador do servidor (*id-servidor*) é menos significativa.

A mesma ordem de transações pode ser obtida em todos os servidores, mesmo que seus relógios locais não estejam sincronizados. Entretanto, por motivos de eficiência, é exigido que os carimbos de tempo publicados por um coordenador sejam aproximadamente sincronizadas com aqueles publicados pelos outros coordenadores. Quando isso acontece, a ordem das transações geralmente corresponde à ordem em que elas foram

iniciadas em tempo real. Os carimbos de tempo podem ser mantidos aproximadamente sincronizados usando-se relógios físicos locais sincronizados (veja o Capítulo 14).

Quando a ordenação por carimbo de tempo é usado para controle de concorrência, os conflitos são resolvidos à medida que cada operação é executada, usando as regras dadas na Seção 16.6. Se a solução de um conflito exigir que uma transação seja cancelada, o coordenador será informado e cancelará a transação em todos os participantes. Portanto, qualquer transação que receba o pedido de confirmação de um cliente, deve ser sempre confirmada. Assim, um participante no protocolo de confirmação de duas fases normalmente concordará com a confirmação. A única situação em que um participante não concordará com a confirmação será se ela tiver falhado durante a transação.

17.4.3 Controle de concorrência otimista

Lembre-se de que no controle de concorrência otimista, cada transação é validada antes de ser confirmada. São atribuídos números de transação no início da validação, e as transações são serializadas de acordo com a ordem dos números de transação. Uma transação distribuída é validada por um conjunto de servidores independentes, cada um dos quais validando transações que acessam seus próprios objetos. Essa validação ocorre durante a primeira fase do protocolo de confirmação de duas fases.

Considere as seguintes interposições das transações T e U , que acessam os objetos A e B nos servidores X e Y , respectivamente.

T		U	
$read(A)$	em X	$read(B)$	em Y
$write(A)$		$write(B)$	
$read(B)$	em Y	$read(A)$	em X
$write(B)$		$write(A)$	

As transações acessam os objetos na ordem T antes de U no servidor X , e na ordem U antes de T no servidor Y . Agora, suponha que T e U iniciem a validação praticamente ao mesmo tempo, mas que o servidor X valide T primeiro e o servidor Y valide U primeiro. Lembre-se de que a Seção 16.5 recomendava uma simplificação do protocolo de validação, que produzia uma regra dizendo que apenas uma transação por vez poderia executar as fases de validação e atualização. Portanto, cada servidor não poderá validar a outra transação até que a primeira tenha terminado. Esse é um exemplo de impasse de confirmação.

As regras de validação da Seção 16.5 presumem que a validação é rápida, o que é verdade para transações com um único servidor. Entretanto, em uma transação distribuída, o protocolo de confirmação de duas fases pode levar algum tempo e retardar a entrada de outras transações na validação até que uma decisão sobre a transação corrente seja tomada. Nas transações distribuídas otimistas, cada servidor aplica um protocolo de validação em paralelo. Essa é uma extensão da validação para trás, ou para frente, para permitir que várias transações estejam na fase de validação ao mesmo tempo. Nesta ampliação, a regra 3 deve ser verificada, assim como a regra 2 de validação para trás. Isto é, o conjunto de escrita da transação que está sendo validada deve ser verificado para saber se existem sobreposições com o conjunto de escrita de transações sobrepostas anteriores. Kung e Robinson [1981] descrevem a validação em paralelo.

Se for usada a validação em paralelo, as transações não sofrerão do impasse de confirmação. Entretanto, se os servidores simplesmente realizarem validações independentes, é possível que diferentes servidores de uma transação distribuída possam serializar o mesmo conjunto de transações, em ordens diferentes; por exemplo, com T antes de U no servidor X , e U antes de T no servidor Y , em nosso exemplo.

Os servidores de transações distribuídas devem impedir que isso aconteça. Uma estratégia é a realização de uma validação global após uma validação local em cada servidor [Ceri e Owicki 1982]. A validação global verifica se a combinação das ordens nos servidores individuais pode ser serializada; isto é, se a transação que está sendo validada não está envolvida em um ciclo.

Outra estratégia é que todos os servidores de uma transação em particular utilizem o mesmo número de transação globalmente exclusivo no início da validação [Schlageter 1982]. O coordenador do protocolo de confirmação de duas fases é responsável por gerar o número de transação globalmente exclusivo e passá-lo para os participantes nas mensagens de *canCommit?*. Como diferentes servidores podem coordenar diferentes transações, os servidores devem ter uma ordem acordada para os números de transação que geram (como no protocolo de ordenação distribuída por carimbo de tempo).

Agrawal *et al.* [1987] propuseram uma variação do algoritmo de Kung e Robinson que favorece as transações somente de leitura, junto a um algoritmo chamado MVGV (Multi-Version Generalized Validation – validação generalizada de versão múltipla). O MVGV é uma forma de validação em paralelo que garante que os números de transação reflitam a ordem serial, mas exige que a visibilidade de algumas transações seja retardada, após terem sido confirmadas. Ele também permite que o número de transação seja alterado, para deixar que algumas transações validem aquelas que, de outro modo, teriam falhado. O artigo também propõe um algoritmo para confirmar transações distribuídas. Ele é semelhante à proposta de Schlageter, no sentido de que um número de transação global precisa ser encontrado. No final da fase de leitura, o coordenador propõe um valor para o número de transação global, e cada participante tenta validar suas transações locais usando esse número. Entretanto, se o número de transação global proposto for pequeno demais, alguns participantes podem não conseguir validar suas transações e negociam um número maior com o coordenador. Se nenhum número conveniente puder ser encontrado, então esses participantes terão que cancelar suas transações. Finalmente, se todos os participantes puderem validar suas transações, o coordenador terá recebido propostas de números de transação de cada um deles. Se puderem ser encontrados números comuns, a transação será confirmada.

17.5 Impasses distribuídos

A discussão sobre impasses da Seção 16.4 mostrou que eles podem surgir dentro de um único servidor quando são usadas travas para controle de concorrência. Os servidores devem impedir ou detectar e resolver os impasses. Usar tempos limites para solucionar possíveis impasses é uma estratégia deselegante – é difícil escolher um intervalo de tempo ou limite apropriado e transações são canceladas desnecessariamente. Com os esquemas de detecção de impasse, uma transação só é cancelada quando está realmente envolvida em um impasse. A maioria dos esquemas de detecção de impasse funciona encontrando ciclos no grafo *espera por* da transação. Teoricamente, em um sistema distribuído envolvendo vários servidores sendo acessados por múltiplas transações, um grafo espera por global pode ser construído a partir dos grafos locais. Pode haver um ciclo no grafo espera

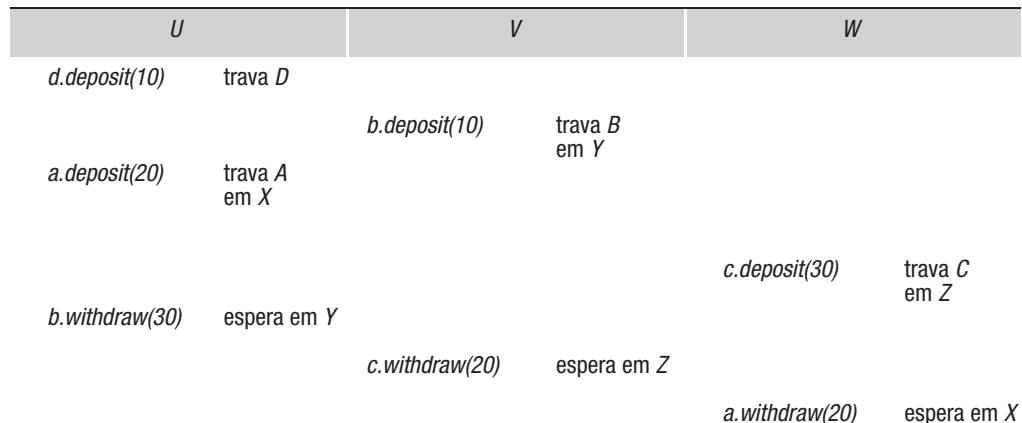


Figura 17.12 Interposições das transações U, V e W.

por global que não aparece em nenhum outro grafo local – isto é, pode haver um *impasse distribuído*. Lembre-se de que o grafo espera por é um grafo direcionado no qual os nós representam transações e objetos, e as arestas representam um objeto retido por uma transação ou uma transação esperando por um objeto. Existe um impasse se, e somente se, existe um ciclo no grafo espera por.

A Figura 17.12 mostra as interposições das transações U, V e W envolvendo os objetos A e B gerenciados pelos servidores X e Y e os objetos C e D gerenciados pelo servidor Z.

O grafo espera por completo da Figura 17.13(a) mostra que um ciclo de impasse consiste em arestas alternadas, as quais representam uma transação esperando por um objeto e um objeto retido por uma transação. Como qualquer transação só pode estar esperando um objeto por vez, os objetos podem ser omitidos dos grafos espera por, como mostrado na Figura 17.13(b).

A detecção de um impasse distribuído exige que um ciclo seja encontrado no grafo espera por de transação global, que é distribuído entre os servidores que estavam envolvidos nas transações. Os grafos espera por locais podem ser construídos pelo gerenciador de travas em cada servidor, conforme discutido no Capítulo 16. No exemplo anterior, os grafos espera por locais dos servidores são:

- servidor Y: $U \rightarrow V$ (adicionado quando U solicita *b.withdraw(30)*)
- servidor Z: $V \rightarrow W$ (adicionado quando V solicita *c.withdraw(20)*)
- servidor X: $W \rightarrow U$ (adicionado quando W solicita *a.withdraw(20)*)

Como o grafo espera por global é mantido parcialmente em cada um dos vários servidores envolvidos, é exigida uma comunicação entre esses servidores para encontrar ciclos no grafo.

Uma solução simples é usar detecção de impasse centralizada, na qual um servidor assume o papel de detector de impasse global. De tempos em tempos, cada servidor envia a cópia mais recente de seu grafo espera por local para o detector de impasse global, o qual reúne as informações dos grafos locais para construir um grafo espera por global. O detector de impasse global verifica a existência de ciclos no grafo espera por global.

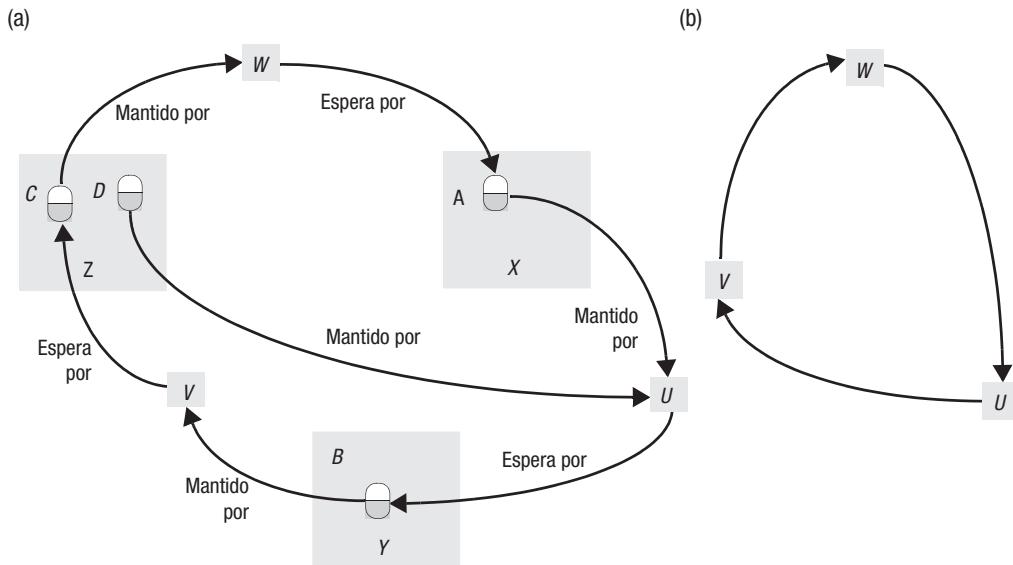


Figura 17.13 Impasse distribuído.

Quando encontra um ciclo, ele toma uma decisão sobre como resolver o impasse e informa os servidores sobre a transação a ser cancelada para resolver o impasse.

A detecção de impasse centralizada não é uma boa ideia, pois depende de um único servidor para executá-la. Ela sofre dos problemas comuns associados às soluções centralizadas em sistemas distribuídos – disponibilidade deficiente, falta de tolerância a falhas e nenhuma escalabilidade. Além disso, o custo da transmissão frequente de grafos espera por locais é alto. Se o grafo global for montado com menos frequência, os impasses poderão demorar mais para serem detectados.

Impasses fantasma • Um impasse que é detectado, mas não é realmente um impasse, é chamado de *impasse fantasma*. Na detecção de impasse distribuído, as informações sobre relacionamentos espera por entre as transações são transmitidas de um servidor para outro. Se houver um impasse, as informações necessárias serão finalmente reunidas em um só lugar e um ciclo será detectado. Como esse procedimento leva algum tempo, existe a chance de que uma das transações que mantém uma trava a tenha liberado nesse meio-tempo, no caso em que o impasse não existirá mais.

Considere o caso de um detector de impasse global que recebe grafos espera por locais dos servidores X e Y, como mostrado na Figura 17.14. Suponha que a transação U libere, então, um objeto no servidor X e solicite o objeto que é mantido por V no servidor Y. Suponha também que o detector global receba o grafo local do servidor Y antes do grafo do servidor X. Nesse caso, ele detectaria um ciclo $T \rightarrow U \rightarrow V \rightarrow T$, embora a seta $T \rightarrow U$ não exista mais. Esse é um exemplo de impasse fantasma.

O leitor atento terá percebido que, se as transações estão usando travas de duas fases, elas não podem liberar objetos e depois obter mais objetos, e ciclos de impasse fantasma não podem ocorrer da maneira sugerida anteriormente. Considere a situação em que um ciclo $T \rightarrow U \rightarrow V \rightarrow T$ é detectado: ou isso representa um impasse, ou cada

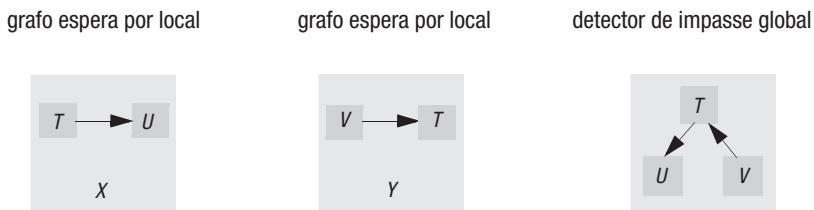


Figura 17.14 Grafos espera por local e global.

uma das transações T , U e V devem ser confirmadas em algum momento. Na verdade, é impossível que qualquer uma delas seja confirmada, pois cada uma está esperando por um objeto que nunca será liberado.

Um impasse fantasma poderia ser detectado se a transação que está em espera em um ciclo de impasse fosse cancelada durante o procedimento de detecção de impasse. Por exemplo, se houvesse um ciclo $T \rightarrow U \rightarrow V \rightarrow T$ e U fosse cancelada após a informação a respeito de U ter sido coletada, então o ciclo já teria sido quebrado e não haveria impasse.

Caminhamento pelas arestas • Uma estratégia distribuída para a detecção de impasse usa uma técnica chamada de caminhamento pelas arestas (*edge chasing*) ou avanço pelo caminho (*path pushing*). Nessa estratégia, o grafo espera por global não é construído, mas cada um dos servidores envolvidos tem conhecimento a respeito de algumas de suas arestas. Os servidores tentam encontrar ciclos encaminhando mensagens de *sondagem* (*probe*), as quais seguem as arestas do grafo por todo o sistema distribuído. Uma mensagem de sondagem consiste nos relacionamentos espera por da transação, representando um caminho no grafo espera por global.

A questão é: quando um servidor deve enviar uma mensagem de sondagem? Considere a situação no servidor X da Figura 17.13. Esse servidor acabou de adicionar a aresta $W \rightarrow U$ em seu grafo espera por local e, neste momento, a transação U está esperando para acessar o objeto B , o qual a transação V mantém no servidor Y . Possivelmente, essa aresta poderia fazer parte de um ciclo como $V \rightarrow T_1 \rightarrow T_2 \rightarrow \dots \rightarrow W \rightarrow U \rightarrow V$, envolvendo transações usando objetos em outros servidores. Isso indica que existe um ciclo de impasse distribuído em potencial, o qual poderia ser encontrado enviando-se uma mensagem de sondagem para o servidor Y .

Agora, considere a situação um pouco anterior, quando o servidor Z adicionou a aresta $V \rightarrow W$ em seu grafo local: nesse momento, W não estava esperando. Portanto, não haveria motivo para enviar uma mensagem de *sondagem*.

Cada transação distribuída começa em um servidor (chamado de coordenador da transação) e se move para vários outros servidores (chamados de participantes da transação), os quais podem se comunicar com o coordenador. A qualquer momento, uma transação pode estar ativa ou esperando em apenas um desses servidores. O coordenador é responsável por registrar se a transação está ativa, ou se está esperando por um objeto em particular, e os participantes podem obter essa informação de seu coordenador. Os gerenciadores de travas informam aos coordenadores quando as transações começam a esperar por objetos, quando elas adquirem objetos e quando se tornam ativas novamente. Quando uma transação for cancelada para desfazer um impasse, seu coordenador informará aos participantes e todas as suas travas serão removidas, com

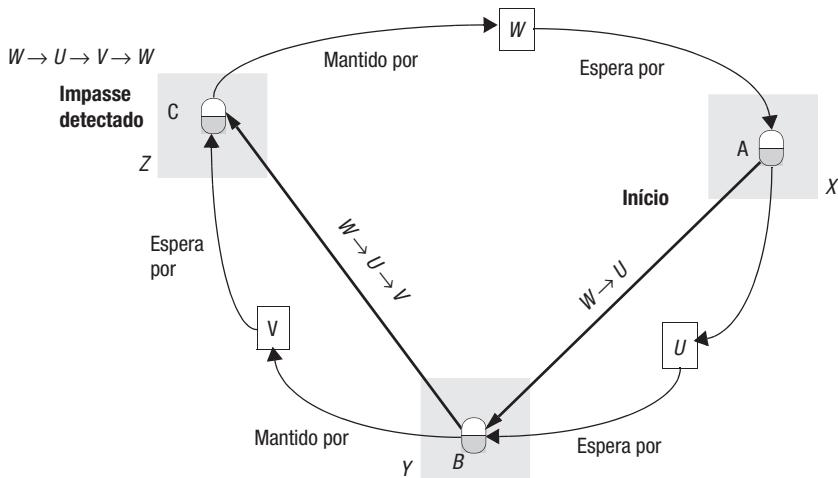


Figura 17.15 Mensagens de sondagem transmitidas para detectar impasse.

o efeito de que todas as arestas envolvendo essa transação serão removidas dos grafos espera por locais.

Os algoritmos de caminhamento pelas arestas têm três etapas – início, detecção e solução.

Início: quando um servidor nota que uma transação T começa a esperar por outra transação U , onde U está esperando para acessar um objeto em outro servidor, ele inicia a detecção enviando uma mensagem de sondagem contendo a aresta $\langle T \rightarrow U \rangle$ para o servidor do objeto em que a transação U está parada. Se U estiver compartilhando uma trava, mensagens de sondagem serão enviadas para todos os proprietários da trava. Às vezes, mais transações podem começar a compartilhar a trava posteriormente, no caso em que as mensagens de sondagem também poderão ser enviadas para elas.

Detecção: a detecção consiste no recebimento de mensagens de sondagem e em decidir se o impasse ocorreu e se as mensagens de sondagens devem ser encaminhadas.

Por exemplo, quando um servidor de um objeto recebe uma mensagem de sondagem $\langle T \rightarrow U \rangle$ (indicando que T está esperando por uma transação U que contém um objeto local), ele verifica se U também está esperando. Se estiver, a transação pela qual ela espera (por exemplo, V) é adicionada na mensagem de sondagem (tornando-a $\langle T \rightarrow U \rightarrow V \rangle$) e se a nova transação (V) estiver esperando por outro objeto de algum lugar, a mensagem de sondagem será encaminhada.

Desse modo, os caminhos pelo grafo espera por global são construídos uma aresta por vez. Antes de encaminhar uma mensagem de sondagem, o servidor verifica se a transação (por exemplo, T) que acabou de adicionar fez com que a mensagem de sondagem contivesse um ciclo (por exemplo, $\langle T \rightarrow U \rightarrow V \rightarrow T \rangle$). Se esse é o caso, ele encontrou um ciclo no grafo e o impasse foi detectado.

Solução: quando um ciclo é detectado, uma transação do ciclo é cancelada para desfazer o impasse.

Em nosso exemplo, as seguintes etapas descrevem como a detecção de impasse é iniciada e as mensagens de sondagem que são encaminhadas durante a fase de detecção correspondente.

- O servidor X inicia a detecção, enviando a mensagem de sondagem $\langle W \rightarrow U \rangle$ para o servidor de B (servidor Y).
- O servidor Y recebe a mensagem de sondagem $\langle W \rightarrow U \rangle$, nota que B é mantido por V e anexa V na mensagem de sondagem para produzir $\langle W \rightarrow U \rightarrow V \rangle$. Ele nota que V está esperando por C no servidor Z . Essa mensagem de sondagem é encaminhada para o servidor Z .
- O servidor Z recebe a mensagem de sondagem $\langle W \rightarrow U \rightarrow V \rangle$, nota que C é mantido por W e anexa W na mensagem de sondagem para produzir $\langle W \rightarrow U \rightarrow V \rightarrow W \rangle$.

Esse caminho contém um ciclo. O servidor detecta um impasse. Uma das transações do ciclo deve ser cancelada para desfazer o impasse. A transação a ser cancelada pode ser escolhida de acordo com as prioridades da transação, as quais serão descritas em breve.

A Figura 17.15 mostra o andamento das mensagens de sondagem desde o início pelo servidor de A até a detecção de impasse pelo servidor de C . As mensagens de sondagem são mostradas como setas grossas, os objetos, como formas ovais (como sempre) e os coordenadores de transação, como retângulos. Cada mensagem de sondagem é mostrada indo diretamente de um objeto a outro. Na realidade, antes que um servidor transmita uma mensagem de sondagem para outro servidor, ele consulta o coordenador da última transação no caminho para descobrir se ela está esperando por outro objeto de alguma parte. Por exemplo, antes que o servidor de B transmita a mensagem de sondagem $\langle W \rightarrow U \rightarrow V \rangle$, ele consulta o coordenador de V para descobrir se V está esperando por C . Na maioria dos algoritmos de caminhamento pelas arestas, os servidores de objetos enviam mensagens de sondagem para os coordenadores de transação, os quais então as encaminham (se a transação estiver esperando) para o servidor do objeto pelo qual a transação está esperando. Em nosso exemplo, o servidor de B transmite a mensagem de sondagem $\langle W \rightarrow U \rightarrow V \rangle$ para o coordenador de V , o qual a encaminha para o servidor de C . Isso mostra que, quando uma mensagem de sondagem é encaminhada, são exigidas duas mensagens.

O algoritmo anterior deve encontrar qualquer ocorrência de impasse, desde que as transações que estão esperando não sejam canceladas, e que não existam falhas como mensagens perdidas ou servidores falhando. Para entender isso, considere um ciclo de impasse no qual a última transação, W , começa a esperar e completa o ciclo. Quando W começa a esperar por um objeto, o servidor dispara uma mensagem de sondagem que vai para o servidor do objeto retido para toda transação pela qual W está esperando. Os destinatários estendem e encaminham as mensagens de sondagem para os servidores dos objetos solicitados por todas as transações que estão esperando que eles encontrarem. Assim, toda transação pela qual W espera, direta ou indiretamente, será adicionada na mensagem de sondagem, a não ser que um impasse seja detectado. Quando há um impasse, W está esperando por si mesma indiretamente. Portanto, a mensagem de sondagem retornará o objeto que W contém.

Em princípio, um grande número de mensagens é enviado para detectar um impasse. No exemplo anterior, vimos duas mensagens de sondagem para detectar um ciclo envolvendo três transações. Em geral, cada uma das mensagens de sondagem é, na verdade, duas mensagens (do objeto para o coordenador e depois do coordenador para o objeto).

Uma mensagem de sondagem que detecta um ciclo envolvendo N transações será encaminhada por $(N - 1)$ coordenadores de transação por intermédio de $(N - 1)$ servidores de objetos, exigindo $2(N - 1)$ mensagens. Felizmente, a maioria dos impasses envolve ciclos contendo apenas duas transações, e não há necessidade de preocupação excessiva com o número de mensagens envolvidas. Essa observação foi feita a partir dos estudos

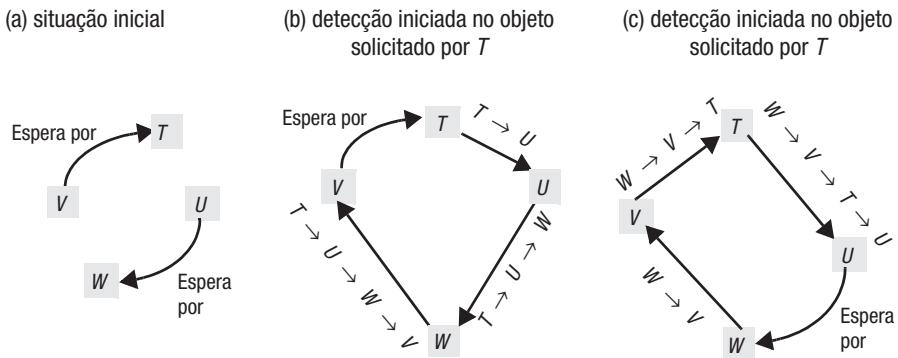


Figura 17.16 Duas mensagens de sondagem disparadas.

dos bancos de dados. Ela também pode ser demonstrada considerando-se a probabilidade de acesso conflitante aos objetos (consulte Bernstein *et al.* [1987]).

Prioridades de transação • No algoritmo anterior, toda transação envolvida em um ciclo de impasse pode causar o início da detecção de impasse. O efeito de várias transações iniciando a detecção de impasse é que esse procedimento pode acontecer em vários servidores diferentes que participam do ciclo, com o resultado de que mais de uma transação pode vir a ser cancelada.

Na Figura 17.16(a), considere as transações T , U , V e W , onde U está esperando por W e V está esperando por T . Praticamente ao mesmo tempo, T solicita o objeto mantido por U e W solicita o objeto mantido por V . Duas mensagens de sondagem separadas $\langle T \rightarrow U \rangle$ e $\langle W \rightarrow V \rangle$ são iniciadas pelos servidores desses objetos e circulam até que o impasse seja detectado em cada um dos dois diferentes servidores. Na Figura 17.16(b), o ciclo é $\langle T \rightarrow U \rightarrow W \rightarrow V \rightarrow T \rangle$, e na Figura 17.16(c), o ciclo é $\langle W \rightarrow V \rightarrow T \rightarrow U \rightarrow W \rangle$.

Para garantir que apenas uma transação em um ciclo seja cancelada, as transações recebem *prioridades*, de maneira que todas as transações são totalmente ordenadas. Carimbos de tempo, por exemplo, podem ser usados como prioridades. Quando é encontrado um ciclo de impasse, a transação com a menor prioridade é cancelada. Mesmo que vários servidores diferentes detectem o mesmo ciclo, todos chegarão à mesma decisão com relação a qual transação deve ser cancelada. Escrivemos $T > U$ para indicar que T tem prioridade mais alta do que U . No exemplo anterior, suponha que $T > U > V > W$. Então, a transação W será cancelada quando um dos ciclos $\langle T \rightarrow U \rightarrow W \rightarrow V \rightarrow T \rangle$ ou $\langle W \rightarrow V \rightarrow T \rightarrow U \rightarrow W \rangle$ for detectado.

As prioridades de transação também poderiam ser usadas para reduzir o número de situações que causam o início da detecção de impasse, usando a regra de que a detecção é iniciada apenas quando uma transação de prioridade mais alta começa a esperar por outra de prioridade mais baixa. Em nosso exemplo da Figura 17.16, como $T > U$, a mensagem de sondagem $\langle T \rightarrow U \rangle$ seria enviada, mas como $W < V$, a mensagem de sondagem $\langle W \rightarrow V \rangle$ não seria enviada. Se supuermos que quando uma transação começa a esperar por outra transação é igualmente provável que a transação que está esperando tenha prioridade mais alta, ou mais baixa, do que a transação pela qual se espera, então o uso dessa regra provavelmente reduz praticamente pela metade o número de mensagens de sondagem.

As prioridades de transação também poderiam ser usadas para reduzir o número de mensagens de sondagem encaminhadas. A ideia geral é que as mensagens de sondagem

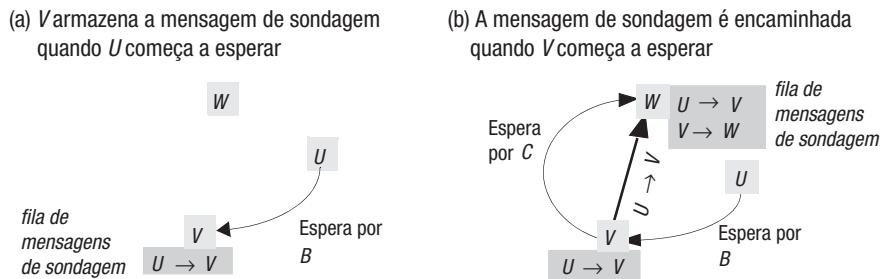


Figura 17.17 As mensagens de sondagem são enviadas “ladeira abaixo”.

devem “descer a ladeira” – isto é, das transações com prioridades mais altas para as transações com prioridades mais baixas. Para fazer isso, os servidores usam a regra de que eles não encaminham nenhuma mensagem de sondagem para um proprietário que tenha prioridade mais alta do que o iniciador. O argumento para se fazer isso é que, se o proprietário estiver esperando por outra transação, deverá ter iniciado a detecção enviando uma mensagem de sondagem quando começou a esperar.

Entretanto, há uma armadilha associada a essas aparentes melhorias. Em nosso exemplo da Figura 17.15, as transações U , V e W são executadas em uma ordem na qual U está esperando por V e V está esperando por W , quando W começa a esperar por U . Sem regras de prioridade, a detecção é iniciada quando W começa a esperar, enviando uma mensagem de sondagem $\langle W \rightarrow U \rangle$. Sob a regra da prioridade, essa mensagem de sondagem não seria enviada, pois $W < U$ e o impasse não seria detectado.

O problema é que a ordem em que as transações começam a esperar pode determinar se o impasse será detectado ou não. A armadilha acima pode ser evitada usando-se um esquema em que os coordenadores salvam cópias de todas as mensagens de sondagem recebidas em nome de cada transação, em uma fila de mensagens de sondagem. Quando uma transação começa a esperar por um objeto, ela encaminha as mensagens de sondagem dessa fila para o servidor do objeto, o qual as propaga nas rotas “ladeira abaixo”.

Em nosso exemplo da Figura 17.15, quando U começar a esperar por V , o coordenador de V salvará a mensagem de sondagem $\langle U \rightarrow V \rangle$ – veja a Figura 17.17(a). Então, quando U começar a esperar por W , o coordenador de W armazenará $\langle V \rightarrow W \rangle$ e V encaminhará sua fila de mensagens de sondagem $\langle U \rightarrow V \rangle$ para W . (Veja a Figura 17.17(b), na qual a fila de mensagens de sondagem de W tem $\langle U \rightarrow V \rangle$ e $\langle V \rightarrow W \rangle$). Quando W começar a esperar por U , encaminhará sua fila de mensagens de sondagem $\langle U \rightarrow V \rightarrow W \rangle$ para o servidor de U , o qual também notará a nova dependência $W \rightarrow U$ e a combinará com as informações das mensagens de sondagem recebidas para determinar que $U \rightarrow V \rightarrow W \rightarrow U$. O impasse foi detectado.

Quando um algoritmo exige que as mensagens de sondagem sejam armazenadas em filas, ele também exige providências para passar essas mensagens para novos proprietários e para descartar as mensagens que se refiram às transações confirmadas ou canceladas. Se mensagens de sondagens relevantes forem descartadas, poderão ocorrer impasses não detectados, e se mensagens de sondagem obsoletas forem mantidas, podem ser detectados impasses falsos. Isso aumenta muito a complexidade de qualquer algoritmo caminhamento pelas arestas. Os leitores que estiverem interessados nos detalhes de tais algoritmos devem consultar Sinha e Natarajan [1985] e Choudhary *et al.* [1989], que apresentam algoritmos para uso com travas exclusivas. No entanto, eles verão que Choudhary *et al.* mostraram que o algoritmo de Sinha e Natarajan está incorreto, pois não detecta todos os impasses e pode até relatar falsos impasses. Kshemkalyani e Singhal

[1991] corrigiram o algoritmo de Choudhary *et al.* (que não detecta todos os impasses e pode relatar impasses falsos) e fornecem uma prova da exatidão do algoritmo corrigido. Em um artigo subsequente, Kshemkalyani e Singhal [1994] argumentam que os impasses distribuídos não são muito bem entendidos, pois não existe nenhum estado ou tempo global em um sistema distribuído. Na verdade, qualquer ciclo reunido pode conter seções registradas em diferentes momentos. Além disso, os *sites* envolvidos podem ser informados da presença de impasses, mas não saberem que eles já foram resolvidos, devido a atrasos aleatórios. O artigo descreve os impasses distribuídos em termos do conteúdo da memória distribuída, usando relacionamentos causais entre eventos em diferentes *sites*.

17.6 Recuperação de transações

A propriedade atômica das transações exige que os efeitos de todas as transações confirmadas, e nenhum dos efeitos das transações incompletas ou canceladas sejam refletidos nos objetos que elas acessaram. Essa propriedade pode ser descrita em termos de dois aspectos: durabilidade e atomicidade da falha. A durabilidade exige que os objetos sejam escritos no meio de armazenamento permanente e, daí em diante, estejam disponíveis indefinidamente. Portanto, o fato de que uma requisição de confirmação por parte de um cliente foi realizada implica que todos os efeitos da transação foram armazenados no meio permanente, assim como nos objetos (voláteis) do servidor. A atomicidade da falha exige que os efeitos das transações sejam atômicos, mesmo que o servidor falhe. A recuperação está relacionada a garantir que os objetos de um servidor sejam duráveis e que o serviço forneça atomicidade de falha.

Embora os servidores de arquivo e de banco de dados mantenham dados em meios permanentes, outros tipos de servidores de objetos recuperáveis não precisam fazer isso, exceto para propósitos de recuperação. Neste capítulo, presumimos que, quando um servidor está sendo executado, ele mantém todos os seus objetos em sua memória volátil e registra os objetos confirmados em um (ou mais) *arquivo de recuperação*. Portanto, a recuperação consiste em restaurar o servidor com as versões mais recentemente de seus objetos, a partir do meio de armazenamento permanente. Os bancos de dados precisam lidar com grandes volumes de dados. Eles geralmente mantêm os objetos em armazenamento estável no disco, com uma cache na memória volátil.

Os dois requisitos de durabilidade e de atomicidade de falha não são independentes um do outro e podem ser tratados com um único mecanismo – o *gerenciador de recuperação*. As tarefas de um gerenciador de recuperação são:

- salvar objetos das transações confirmadas em um meio permanente (em um arquivo de recuperação);
- restaurar os objetos do servidor após uma falha;
- reorganizar o arquivo de recuperação para melhorar o desempenho da recuperação;
- recuperar espaço de armazenamento (no arquivo de recuperação).

Em alguns casos, é exigido que o gerenciador de recuperação trate rapidamente de falhas de mídia – falhas de seu arquivo de recuperação, de modo que alguns dados do disco sejam perdidos, ou por se tornarem corrompidos durante uma falha, deteriorando-se aleatoriamente, ou ainda, devido a uma falha permanente. Em tais casos, precisamos de outra cópia do arquivo de recuperação. Para isso, pode ser usado o armazenamento estável, que é implementado de forma que uma falha seja muito improvável, usando discos espelhados ou cópias em um local diferente.

<i>Tipo de entrada</i>	<i>Descrição do conteúdo da entrada</i>
Objeto	O valor de um objeto.
Status da transação	Identificador da transação, <i>status</i> da transação (<i>preparada</i> , <i>confirmada</i> , <i>cancelada</i>) – e outros valores de <i>status</i> usados pelo protocolo de confirmação de duas fases.
Lista de intenções	Identificador da transação e uma sequência de intenções, cada uma das quais consistindo em $\langle objectID, P_i \rangle$, em que P_i é a posição do valor do objeto no arquivo de recuperação.

Figura 17.18 Tipos de entrada em um arquivo de recuperação.

Lista de intenções • Qualquer servidor que forneça transações precisa rastrear os objetos acessados pelas transações dos clientes. Lembre-se, do Capítulo 16, que quando um cliente abre uma transação, o primeiro servidor contatado fornece um novo identificador de transação e o retorna para o cliente. Cada requisição de cliente subsequente dentro de uma transação, até (e incluindo) a requisição de *confirmação* ou *cancelamento*, inclui o identificador de transação como argumento. Durante o andamento de uma transação, as operações de atualização são aplicadas em um conjunto privado de versões de tentativa dos objetos pertencentes à transação.

Em cada servidor, é gravada uma *lista de intenções* para todas as suas transações correntemente ativas – uma lista de intenções de uma transação em particular contém uma lista das referências e dos valores de todos os objetos alterados por essa transação. Quando uma transação é confirmada, a lista de intenções da transação é usada para identificar os objetos afetados por ela. A versão confirmada de cada objeto é substituída pela versão de tentativa feita por essa transação e o novo valor é gravado no arquivo de recuperação do servidor. Quando uma transação é cancelada, o servidor usa a lista de intenções para excluir todas as versões de tentativa de objetos feitas por essa transação.

Lembre-se também de que uma transação distribuída deve executar um protocolo de confirmação atômica antes de poder ser confirmada ou cancelada. Nossa discussão sobre recuperação é baseada no protocolo de confirmação de duas fases, no qual todos os participantes envolvidos em uma transação primeiro dizem se estão preparados para confirmar e, posteriormente, se todos os participantes concordarem, executam as ações de confirmação reais. Se os participantes não puderem concordar com a confirmação, eles deverão cancelar a transação.

No momento em que um participante diz que está preparado para confirmar uma transação, seu gerenciador de recuperação deve ter salvado em seu arquivo de recuperação a lista de intenções para essa transação e os seus objetos, para que, posteriormente, possa realizar a confirmação, mesmo que falhe nesse ínterim.

Quando todos os participantes envolvidos em uma transação concordam em confirmá-la, o coordenador informa ao cliente e, então, envia mensagens para os participantes efetivarem sua parte da transação. Quando o cliente tiver sido informado de que uma transação foi confirmada, os arquivos de recuperação dos servidores participantes deverão conter informações suficientes para garantir que a transação seja confirmada por todos os servidores, mesmo que alguns deles falhem entre a preparação para confirmar e a confirmação em si.

Entradas no arquivo de recuperação • Para tratar da recuperação de um servidor que pode estar envolvido em transações distribuídas, mais informações, além dos valores dos objetos,

são armazenadas no arquivo de recuperação. Essas informações se relacionam ao *status* de cada transação – seja ela *confirmada*, *cancelada* ou *preparada para confirmar*. Além disso, cada objeto no arquivo de recuperação é associado a uma transação em particular por intermédio do salvamento da lista de intenções no arquivo de recuperação. A Figura 17.18 mostra um resumo dos tipos de entrada incluídos em um arquivo de recuperação.

Os valores de status da transação relacionados ao protocolo de confirmação de duas fases serão discutidos na Seção 17.6.4, sobre a recuperação do protocolo de confirmação de duas fases. Descreveremos, agora, duas estratégias para o uso de arquivos de recuperação: *log* e versões de sombra.

17.6.1 Log

Na técnica do *log*, o arquivo de recuperação é um registro contendo o histórico de todas as transações realizadas por um servidor. O histórico consiste nos valores dos objetos, em entradas de status da transação e nas listas de intenções das transações. A ordem das entradas no *log* reflete a ordem em que as transações foram preparadas, confirmadas e canceladas nesse servidor. Na prática, o arquivo de recuperação conterá um instantâneo recente dos valores de todos os objetos presentes no servidor, seguido de um histórico das transações após o instantâneo.

Durante a operação normal de um servidor, seu gerenciador de recuperação é chamado quando uma transação se prepara para confirmar, confirmando-a ou cancelando-a. Quando o servidor está preparado para confirmar uma transação, o gerenciador de recuperação anexa todos os objetos de sua lista de intenções no arquivo de recuperação, seguidos do status corrente dessa transação (*preparada*), junto a sua lista de intenções. Quando uma transação é confirmada ou cancelada, o gerenciador de recuperação anexa o status correspondente da transação em seu arquivo de recuperação.

Presume-se que a operação de anexação seja atômica, pois ela escreve uma, ou mais, entradas completas no arquivo de recuperação. Se o servidor falhar, apenas a última escrita poderá estar incompleta. Para fazer uso eficiente do disco, várias escritas subsequentes podem ser colocadas em um *buffer* e, então, escritas no disco como uma única operação. Uma vantagem adicional da técnica de *log* é que escritas sequenciais no disco são mais rápidas do que as escritas randômicas.

Após uma falha, toda transação que não tenha o status de *confirmada* no *log* é cancelada. Portanto, quando uma transação é confirmada, sua entrada de status *confirmada* deve ser *forçada* no *log* – isto é, escrita no *log*, junto a todas as outras entradas colocadas no *buffer*.

O gerenciador de recuperação associa um identificador exclusivo a cada objeto para que as sucessivas versões de um objeto no arquivo de recuperação possam ser associadas aos objetos do servidor. Por exemplo, uma forma durável de referência de objeto remoto, como uma referência persistente do CORBA, servirá como identificador de objeto.

A Figura 17.19 ilustra o mecanismo de *log* para as transações do serviço de transações bancárias *T* e *U* da Figura 16.7. O *log* foi recentemente reorganizado e as entradas à esquerda da linha dupla representam um instantâneo dos valores de *A*, *B* e *C*, antes que as transações *T* e *U* começassem. Nesse diagrama, usamos os nomes *A*, *B* e *C* como identificadores exclusivos dos objetos. Mostramos a situação de quando a transação *T* foi confirmada e a transação *U* estava preparada, mas não tinha sido confirmada. Quando a transação *T* se prepara para ser confirmada, os valores dos objetos *A* e *B* são gravados nas posições *P*₁ e *P*₂ do *log*, seguidos de uma entrada de status de transação preparada para *T*, com sua lista de intenções (*<A, P*₁*>*, *<B, P*₂*>*). Quando a transação *T* é confirmada, uma entrada status de transação confirmada para *T* é colocada na posição *P*₄. Então, quando a transação *U* se prepara para confirmar-

P_0				P_1	P_2	P_3	P_4	P_5	P_6	P_7
Objeto: <i>A</i>	Objeto: <i>B</i>	Objeto: <i>C</i>		Objeto: <i>A</i>	Objeto: <i>B</i>	Trans: <i>T</i> preparada $\langle A, P_1 \rangle$ $\langle B, P_2 \rangle$ P_0	Trans: <i>T</i> confirmada P_3	Objeto: <i>C</i>	Objeto: <i>B</i>	Trans: <i>U</i> preparada $\langle C, P_5 \rangle$ $\langle B, P_6 \rangle$ P_4
100	200	300		80	220			278	242	

Figura 17.19 Log do serviço de transação bancária.

da, os valores dos objetos *C* e *B* são escritos nas posições P_5 e P_6 do *log*, seguidos de uma entrada de status de transação preparada para *U*, com sua lista de intenções ($\langle C, P_5 \rangle$, $\langle B, P_6 \rangle$).

Cada entrada de status de transação contém um ponteiro para a posição no arquivo de recuperação da entrada de status de transação anterior, para permitir que o gerenciador de recuperação siga a entrada de status de transação na ordem inversa pelo arquivo de recuperação. O último ponteiro na sequência de entradas de status de transação aponta para o ponto de verificação.

Recuperação de objetos • Quando um servidor é substituído após uma falha, ele primeiro configura os valores padrão iniciais de seus objetos e depois transmite para seu gerenciador de recuperação. O gerenciador de recuperação é responsável por restaurar os objetos do servidor, para que eles incluam na ordem correta todos os efeitos das transações confirmadas realizadas e nenhum dos efeitos de transações incompletas ou canceladas.

As informações mais recentes sobre as transações estão no final do *log*. Existem duas estratégias para restaurar os dados do arquivo de recuperação. Na primeira, o gerenciador de recuperação começa no início e restaura os valores de todos os objetos a partir do ponto de verificação mais recente. Então, ele lê os valores de cada um dos objetos, associa-os às suas listas de intenções e, para as transações confirmadas, substitui os valores dos objetos. Nesta estratégia, as transações são reproduzidas na ordem em que foram executadas, e pode haver um grande número delas. Na segunda estratégia, o gerenciador de recuperação restaurará os objetos de um servidor “lendo o arquivo de recuperação de trás para frente”. O arquivo de recuperação foi estruturado de modo a existir um ponteiro de trás para frente de cada entrada de status de transação para a seguinte. O gerenciador de recuperação usa as transações com status de *confirmada* para restaurar os objetos que ainda não foram restaurados. Ele continua até ter restaurado os objetos de todos os servidores. Isso tem a vantagem de que cada objeto é restaurado apenas uma vez.

Para recuperar os efeitos de uma transação, um gerenciador de recuperação obtém a lista de intenções correspondente de seu arquivo de recuperação. A lista de intenções contém os identificadores e as posições no arquivo de recuperação dos valores de todos os objetos afetados pela transação.

Se o servidor falhar no ponto alcançado na Figura 17.19, seu gerenciador de recuperação recuperará os objetos como segue: ele começa na última entrada de status de transação do *log* (em P_7) e conclui que a transação *U* não foi confirmada e seus efeitos devem ser ignorados. Em seguida, ele passa para a entrada de status de transação anterior do *log* (em P_4) e conclui que a transação *T* foi confirmada. Para recuperar os objetos afetados pela transação *T*, ele passa para a entrada de status de transação anterior do *log* (em P_3) e encontra a lista de intenções de *T* ($\langle A, P_1 \rangle$, $\langle B, P_2 \rangle$). Depois, ele restaura os objetos *A* e

B a partir dos valores em P_1 e P_2 . Como ainda não restaurou *C*, ele volta para P_0 , que é um ponto de verificação, e restaura *C*.

Para ajudar na reorganização subsequente do arquivo de recuperação, o gerenciador de recuperação anota todas as transações preparadas que encontra durante o processo de restauração dos objetos do servidor. Para cada transação preparada, ele adiciona um status de transação cancelada no arquivo de recuperação. Isso garante que, no arquivo de recuperação, toda transação seja mostrada como confirmada ou cancelada.

O servidor poderia falhar novamente, durante os procedimentos de recuperação. É fundamental que a recuperação seja idempotente, no sentido de poder ser refeita qualquer número de vezes, com o mesmo efeito. Isso é simples, sob nossa suposição de que todos os objetos são restaurados na memória volátil. No caso de um banco de dados, que mantém seus objetos em um meio de armazenamento permanente, com uma cache em memória volátil, alguns dos objetos do meio permanente poderão estar desatualizados quando o servidor for substituído após uma falha. Portanto, seu gerenciador de recuperação precisa restaurar os objetos no meio permanente. Se ele falhar durante a recuperação, os objetos parcialmente restaurados ainda estarão lá. Isso torna a idempotência um pouco mais difícil de se conseguir.

Reorganização do arquivo de recuperação • Um gerenciador de recuperação é responsável por reorganizar seu arquivo de recuperação para tornar o processo de recuperação mais rápido e reduzir o uso de espaço. Se o arquivo de recuperação nunca for reorganizado, o processo de recuperação deverá fazer pesquisas para trás no arquivo de recuperação até encontrar um valor para cada um de seus objetos. Conceitualmente, a única informação exigida para a recuperação é uma cópia das versões confirmadas de todos os objetos presentes no servidor. Essa seria a forma mais compacta do arquivo de recuperação. A ação de *estabelecer um ponto de verificação (checkpointing)* é usada para se referir ao processo de gravar os valores confirmados correntes dos objetos de um servidor em um novo arquivo de recuperação, junto às entradas de status de transação e listas de intenções das transações que ainda não foram totalmente resolvidas (incluindo informações relacionadas ao protocolo de confirmação de duas fases). O termo *ponto de verificação (checkpoint)* é usado para se referir às informações armazenadas pelo processo de estabelecer um ponto de verificação. O objetivo de estabelecer pontos de verificação é reduzir o número de transações para se tratar durante a recuperação e para reaver espaço de arquivo.

O estabelecimento de pontos de verificação pode ser feito imediatamente após a recuperação, mas antes que quaisquer novas transações sejam iniciadas. Entretanto, a recuperação pode não ocorrer muito frequentemente. Portanto, o estabelecimento de pontos de verificação talvez precise ser feito de tempos em tempos, durante a atividade normal de um servidor. O ponto de verificação é gravado em um arquivo de recuperação futura e o arquivo de recuperação corrente permanece em uso até que o ponto de verificação esteja concluído. O estabelecimento de pontos de verificação consiste em “adicionar uma marca” no arquivo de recuperação, quando o processo começa, gravando os objetos do servidor no arquivo de recuperação futura e copiando (1) todas as entradas antes da marca relacionadas às transações ainda não resolvidas e (2) todas as entradas após a marca do arquivo de recuperação, no arquivo de recuperação futura. Quando o ponto de verificação está concluído, o arquivo de recuperação futura se torna o arquivo de recuperação.

O sistema de recuperação pode reduzir o uso de espaço, descartando o arquivo de recuperação antigo. Quando o gerenciador de recuperação está executando o processo de recuperação, ele pode encontrar um ponto de verificação no arquivo de recuperação. Quando isso acontece, ele pode restaurar imediatamente todos os objetos pendentes a partir do ponto de verificação.

	Mapa no início			Mapa quando T é confirmada			
	$A \rightarrow P_0$	$B \rightarrow P_0'$	$C \rightarrow P_0''$		$A \rightarrow P_1$	$B \rightarrow P_2$	$C \rightarrow P_0'''$
<i>Repositório de versões</i>	P_0	P_0'	P_0''	P_1	P_2	P_3	P_4
	100	200	300	80	220	278	242
	<i>Ponto de verificação</i>						

Figura 17.20 Versões de sombra.

17.6.2 Versões de sombra

A técnica de *log* grava entradas de status de transação, listas de intenções e objetos, tudo no mesmo arquivo – o *log*. A técnica das *versões de sombra* é uma maneira alternativa de organizar um arquivo de recuperação. Ela usa um *mapa* para localizar versões dos objetos do servidor em um arquivo chamado *repositório de versões*. O mapa associa os identificadores dos objetos do servidor às posições de suas versões correntes no repositório de versões. As versões gravadas pelas transações são *sombra*s das versões confirmadas anteriores. As entradas de status de transação e as listas de intenções são tratadas separadamente. As versões de sombra serão descritas primeiramente.

Quando uma transação está preparada para ser confirmada, todos os objetos alterados pela transação são anexados ao repositório de versões, deixando as versões confirmadas correspondentes intactas. Essas novas (até agora) versões de tentativa são chamadas de *versões de sombra*. Quando uma transação é confirmada, é feito um novo mapa, copiando o mapa antigo e inserindo as posições das versões de sombra. Para concluir o processo de confirmação, o novo mapa substitui o antigo.

Para restaurar os objetos quando um servidor é substituído após uma falha, seu gerenciador de recuperação lê o mapa e usa as informações constantes nele para localizar os objetos no repositório de versões.

A Figura 17.20 ilustra essa técnica com o mesmo exemplo envolvendo as transações T e U . A primeira coluna da tabela mostra o mapa antes das transações T e U , quando os saldos das contas A , B e C eram de \$100, \$200 e \$300, respectivamente. A segunda coluna mostra o mapa após a transação T ser confirmada.

O repositório de versões contém um ponto de verificação, seguido das versões de A e B em P_1 e P_2 feitas pela transação T . Ele também contém as versões de sombra de B e C feitas pela transação U , em P_3 e P_4 .

O mapa sempre deve ser gravado em um local conhecido (por exemplo, no início do repositório de versões ou em um arquivo separado) para que possa ser encontrado quando o sistema precisar ser recuperado.

A troca do mapa antigo pelo novo deve ser realizada em uma única etapa atômica. Para se conseguir isso é fundamental que um armazenamento estável seja usado para o mapa – de modo que haja garantia de que ele seja um mapa válido, mesmo quando uma operação de escrita de arquivo falhar. O método das versões de sombra proporciona recuperação mais rápida do que o do *log*, pois as posições dos objetos confirmados correntes são registradas no mapa, enquanto a recuperação a partir de um *log* exige uma busca pelos objetos em todo o *log*. O *log* deve ser mais rápido do que as versões de sombra durante a atividade normal do sistema. Isso porque o *log* exige apenas uma sequência de

operações de anexação no mesmo arquivo, enquanto as versões de sombra exigem uma escrita no armazenamento estável (envolvendo dois blocos de disco não relacionados).

As versões de sombra por si só não são suficientes para um servidor que manipule transações distribuídas. As entradas de status da transação e as listas de intenções são salvas em um arquivo chamado *arquivo de status de transação*. Cada lista de intenções representa a parte do mapa que será alterada por uma transação, quando for confirmada. O arquivo de status de transação pode, por exemplo, ser organizado como um *log*.

A figura a seguir mostra o mapa e o arquivo de status de transação de nosso exemplo corrente, quando T foi confirmada e U está preparada para ser confirmada.

<i>Arquivo de status de transação (armazenamento estável)</i>		
<i>Mapa</i>	T	U
$A \rightarrow P_1$	preparada	confirmada
$B \rightarrow P_2$	$A \rightarrow P_1$	$B \rightarrow P_3$
$C \rightarrow P_0$ "	$B \rightarrow P_2$	$C \rightarrow P_4$

Há uma chance de um servidor falhar entre o momento em que um status de *confirmada* é gravado no arquivo de status de transação e quando o mapa é atualizado – nesse caso, o cliente não receberá a informação de que a operação de confirmação foi realizada. O gerenciador de recuperação deve considerar essa possibilidade quando o servidor for substituído após uma falha; por exemplo, verificando se o mapa inclui os efeitos da última transação confirmada no arquivo de status de transação. Se ele não inclui, então esta última transação deve ser marcada como cancelada.

17.6.3 A necessidade de entradas de status de transação e lista de intenções em um arquivo de recuperação

É possível projetar um arquivo de recuperação simples que não inclua entradas para itens de status de transação, nem listas de intenções. Esse tipo de arquivo de recuperação pode ser conveniente quando todas as transações são direcionadas para um único servidor. O uso de itens de status de transação e listas de intenções no arquivo de recuperação é fundamental para um servidor destinado a participar de transações distribuídas. Essa estratégia também pode ser útil para servidores de transações não distribuídas por vários motivos, incluindo os seguintes:

- Alguns gerenciadores de recuperação são feitos para escrever os objetos no arquivo de recuperação antecipadamente – sob a suposição de que as transações normalmente são confirmadas.
- Se as transações usam um grande número de objetos grandes, a necessidade de escrevê-los no arquivo de recuperação pode complicar o projeto de um servidor. Quando os objetos são referenciados a partir de listas de intenções, eles podem ser encontrados onde quer que estejam.
- No controle de concorrência com ordenação por carimbo de tempo, às vezes um servidor sabe que uma transação poderá ser confirmada e envia um aviso para o cliente – nesse momento, os objetos são escritos no arquivo de recuperação (veja o Capítulo 16) para garantir sua permanência. Entretanto, a transação talvez tenha de esperar para ser confirmada, até que as transações anteriores tenham sido confirmadas. Em tais situações, as entradas de status de transação correspondentes no

arquivo de recuperação estarão *esperando para ser confirmadas* e, então, passam a ser *confirmadas* para garantir a ordenação por carimbo de tempo de transações confirmadas no arquivo de recuperação. Na recuperação, as transações que estão esperando para ser confirmadas podem receber permissão para isso, pois aquelas que estavam esperando acabaram de ser confirmadas ou, caso contrário, foram canceladas devido à falha do servidor.

17.6.4 Recuperação do protocolo de confirmação de duas fases

Em uma transação distribuída, cada servidor mantém seu próprio arquivo de recuperação. O gerenciamento de recuperação descrito na seção anterior deve ser ampliado para lidar com as transações que estão executando o protocolo de confirmação de duas fases no momento em que um servidor falha. Os gerenciadores de recuperação usam dois novos valores de status: *pronto* e *incerto*. Esses valores de status aparecem na Figura 17.6. Um coordenador usa *confirmada* para indicar que o resultado do voto é *Sim* e *pronto* para indicar que o protocolo de confirmação de duas fases está concluído. Um participante usa *incerto* para indicar que votou *Sim*, mas ainda não conhece o resultado. Dois tipos adicionais de entrada permitem que um coordenador registre uma lista de participantes e que um participante registre seu coordenador:

<i>Tipo de entrada</i>	<i>Descrição do conteúdo da entrada</i>
<i>Coordenador</i>	Identificador de transação, lista de participantes
<i>Participante</i>	Identificador de transação, coordenador

Na fase 1 do protocolo, quando o coordenador está preparado para confirmar (e já adicionou uma entrada de status preparada em seu arquivo de recuperação), seu gerenciador de recuperação adiciona uma entrada *coordenador* em seu arquivo de recuperação. Antes que um participante possa votar *Sim*, ele já deve ter se preparado para confirmar (e já deve ter adicionado uma entrada de status *preparada* em seu arquivo de recuperação). Quando ele vota *Sim*, seu gerenciador de recuperação registra uma entrada *participante* e adiciona um status de transação *incerto* em seu arquivo de recuperação. Quando um participante vota *Não*, ele adiciona um status de transação *cancelar* em seu arquivo de recuperação.

Na fase 2 do protocolo, o gerenciador de recuperação de um coordenador adiciona um status de transação *confirmada* ou *cancelada* em seu arquivo de recuperação, de acordo com a decisão. Essa deve ser uma escrita forçada (isto é, feita imediatamente no arquivo de recuperação). Os gerenciadores de recuperação dos participantes adicionam um status de transação *confirmar* ou *cancelar* em seus arquivos de recuperação, de acordo com a mensagem recebida do coordenador. Quando um coordenador tiver recebido uma resposta de todos os seus participantes, seu gerenciador de recuperação adicionará um status de transação *pronto* em seu arquivo de recuperação – essa escrita não precisa ser forçada. A entrada de status *pronto* não faz parte do protocolo, mas é usada quando o arquivo de recuperação é reorganizado. A Figura 17.21 mostra as entradas em um *log* da transação *T*, na qual o servidor desempenhou o papel de coordenador, e da transação *U*, na qual o servidor desempenhou o papel de participante. Para as duas transações, a entrada de status de transação *preparada* vem primeiro. No caso de um coordenador, ela é seguida de uma entrada de coordenador e de uma entrada de status de transação *confirmada*. A entrada de status de transação *pronto* não é mostrada na Figura 17.21. No caso

Trans: <i>T</i> preparada intenções lista	Coord: <i>T</i> lista de partici- pantes:... •	•	Trans: <i>T</i> confirmada	Trans: <i>U</i> preparada intenções lista	•	•	Participante: <i>U</i> Coord:... incerto	Trans: <i>U</i> incerto	Trans: <i>U</i> confirmada
--	--	---	-------------------------------	--	---	---	--	----------------------------	-------------------------------

Figura 17.21 Log com entradas relacionadas ao protocolo de confirmação de duas fases.

de um participante, a entrada de status de transação *preparada* é seguida de uma entrada de participante cujo estado é *incerto* e, depois, de uma entrada de status de transação *confirmada* ou *cancelada*.

Quando a servidor é substituído após uma falha, o gerenciador de recuperação precisa lidar com o protocolo de confirmação de duas fases, além de restaurar os objetos. Para qualquer transação em que o servidor tiver desempenhado o papel de coordenador, ele deve encontrar uma entrada de coordenador e um conjunto de entradas de status de transação. Para qualquer transação em que o servidor desempenhou o papel de participante, ele deve encontrar uma entrada de participante e um conjunto de entradas de status de transação. Nos ambos os casos, a entrada de status de transação mais recente – isto é, aquela mais próxima ao final do *log* – determina o status da transação no momento da falha. A ação do gerenciador de recuperação, com relação ao protocolo de confirmação de duas fases para qualquer transação, depende de o servidor ter sido coordenador ou participante, e de seu status no momento da falha, como mostrado na Figura 17.22.

Reorganização do arquivo de recuperação • Deve-se tomar cuidado ao estabelecer um ponto de verificação para garantir que as entradas *coordenador* das transações, sem status *pronto*, não sejam removidas do arquivo de recuperação. Essas entradas devem ser mantidas até que todos os participantes tenham garantido que concluíram suas transações. As entradas com status *pronto* podem ser descartadas. As entradas de participante com estado de transação *incerto* também devem ser mantidas.

Recuperação de transações aninhadas • No caso mais simples, cada subtransação de uma transação aninhada acessa um conjunto de objetos diferente. Quando cada participante se prepara para confirmar, durante o protocolo de confirmação de duas fases, ele escreve seus objetos e suas listas de intenções no arquivo de recuperação local, associando-os ao identificador da transação de nível superior. Embora as transações aninhadas utilizem uma variante especial do protocolo de confirmação de duas fases, o gerenciador de recuperação usa os mesmos valores de status das transações planas.

Entretanto, a recuperação do cancelamento é complicada pelo fato de que várias subtransações nos mesmos níveis, e em níveis diferentes na hierarquia do aninhamento, podem acessar o mesmo objeto. A Seção 16.4 descreveu um esquema de travamento no qual as transações ascendentes herdam travas, e as subtransações adquirem travas de suas ascendentes. O esquema de travamento obriga as transações ascendentes e as subtransações a acessarem objetos de dados comuns em diferentes momentos e garante que os acessos feitos por subtransações concorrentes aos mesmos objetos sejam organizados em série.

Os objetos acessados de acordo com as regras das transações aninhadas tornam-se recuperáveis por meio do fornecimento de versões de tentativa para cada subtransação. O relacionamento entre as versões de tentativa de um objeto usado pelas subtransações de uma transação aninhada é semelhante ao relacionamento entre as travas. Para suportar a

Papel	Status	Ação do gerenciador de recuperação
Coordenador	<i>preparada</i>	Não chegou a nenhuma decisão antes que o servidor falhasse. Ele envia <i>abortTransaction</i> para todos os servidores da lista de participantes e adiciona o status de transação <i>cancelada</i> em seu arquivo de recuperação. A mesma ação para o estado <i>cancelada</i> . Se não houver nenhuma lista de participantes, os participantes finalmente atingirão o tempo limite e cancelarão a transação.
Coordenador	<i>confirmada</i>	Chegou à decisão de confirmar antes que o servidor falhasse. Ele envia uma mensagem <i>doCommit</i> para todos os participantes de sua lista de participantes (no caso de não ter feito isso antes) e retoma o protocolo de duas fases na etapa 4 (veja a Figura 17.5).
Participante	<i>confirmada</i>	O participante envia uma mensagem <i>haveCommitted</i> para o coordenador (no caso de não ter feito isso antes da falha). Isso permitirá que o coordenador descarte as informações sobre essa transação no próximo ponto de verificação.
Participante	<i>incerto</i>	O participante falhou antes de saber o resultado da transação. Ele não pode determinar o status da transação até que o coordenador o informe sobre a decisão. Ele enviará uma mensagem <i>getDecision</i> para o coordenador, para determinar o status da transação. Quando receber a resposta, ele confirmará ou cancelará, de acordo com o resultado.
Participante	<i>preparada</i>	O participante ainda não votou e pode cancelar a transação.
Coordenador	<i>pronto</i>	Nenhuma ação é exigida.

Figura 17.22 Recuperação do protocolo de confirmação de duas fases.

recuperação dos cancelamentos, o servidor de um objeto compartilhado por transações em vários níveis fornece uma pilha de versões de tentativa – uma para cada transação aninhada a ser usada.

Quando a primeira subtransação de um conjunto de transações aninhadas acessa um objeto, ela recebe uma versão de tentativa que é uma cópia da versão confirmada corrente do objeto. Esse é considerado o topo da pilha, mas, a não ser que outras subtransações acessem o mesmo objeto, a pilha não se materializará.

Quando uma de suas subtransações acessa o mesmo objeto, ela copia a versão do topo da pilha e a coloca de volta na pilha. Todas as atualizações dessa subtransação são aplicadas na versão de tentativa do topo da pilha. Quando uma subtransação é confirmada provisoriamente, sua ascendente herda a nova versão. Para conseguir isso, a versão da subtransação e a versão de sua ascendente são descartadas da pilha e, depois, a nova versão da subtransação é colocada de volta na pilha (efetivamente substituindo a versão de sua ascendente). Quando uma subtransação é cancelada, sua versão que está no topo da pilha é descartada. Finalmente, quando a transação de nível superior for confirmada, a versão que está no topo da pilha (se houver) se torna a nova versão confirmada.

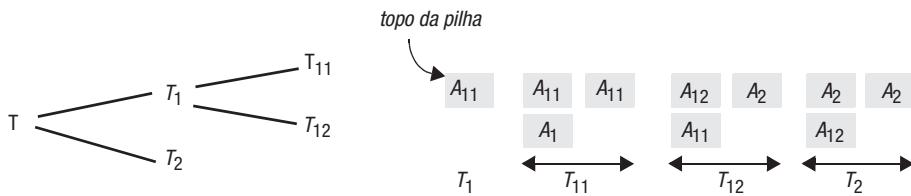


Figura 17.23 Transações aninhadas.

Por exemplo, na Figura 17.23, suponha que as transações T_1 , T_{11} , T_{12} e T_2 acessem, o mesmo objeto, A , na ordem T_1 ; T_{11} ; T_{12} ; T_2 . Suponha que suas versões de tentativa se chamem A_1 , A_{11} , A_{12} e A_2 . Quando T_1 começa a executar, A_1 é baseado na versão confirmada de A e colocado na pilha. Quando T_{11} começa a executar, ela baseia sua versão A_{11} em A_1 e o coloca na pilha; quando termina, substitui pela versão de seu ascendente na pilha. As transações T_{12} e T_2 agem de maneira semelhante, deixando finalmente o resultado de T_2 no topo da pilha.

17.7 Resumo

No caso mais geral, a transação de um cliente solicitará operações sobre objetos em diferentes servidores. Uma transação distribuída é qualquer transação cuja atividade envolve vários servidores diferentes. Uma estrutura de transação aninhada pode ser usada para possibilitar concorrência adicional e confirmação independente por parte dos servidores em uma transação distribuída.

A propriedade da atomicidade das transações exige que todos os servidores participantes de uma transação distribuída a confirmem ou a cancelem. Os protocolos de confirmação atômica são feitos para obter esse efeito, mesmo que os servidores falhem durante sua execução. O protocolo de confirmação de duas fases permite que um servidor decida cancelar unilateralmente. Ele inclui ações a serem executadas quando esgotar um tempo limite para lidar com atrasos resultantes da falha de servidores. O protocolo de confirmação de duas fases pode usar um período de tempo ilimitado para terminar, mas é garantido que ele finalmente termine.

O controle de concorrência em transações distribuídas é modular – cada servidor é responsável pela serialização das transações que acessam seus próprios objetos. Entretanto, são exigidos protocolos adicionais para garantir que as transações sejam serializadas de forma global. As transações distribuídas que usam ordenação por carimbo de tempo exigem que os diferentes servidores gerem carimbos de tempo de forma ordenada e acordada entre eles. Aquelas que usam controle de concorrência otimista exigem validação global ou uma maneira de impor uma ordem global nas transações que estão sendo confirmadas.

As transações distribuídas que usam travas de duas fases podem sofrer do problema dos impasses distribuídos. O objetivo da detecção de impasse distribuído é procurar ciclos no grafo espera por global. Se for encontrado um ciclo, uma ou mais transações devem ser canceladas para resolver o impasse. O caminhamento pelas arestas é uma estratégia não centralizada para a detecção de impasses distribuídos.

Os aplicativos baseados em transação têm fortes requisitos para a perenidade e a integridade das informações armazenadas, mas normalmente não têm requisitos de res-

posta imediata o tempo todo. Os protocolos de confirmação atômica são a chave para as transações distribuídas, mas não podem dar garantia de conclusão dentro de um limite de tempo específico. As transações se tornam duráveis pelo estabelecimento de pontos de verificação e pelo registro em um arquivo de recuperação, que é usado para recuperação quando um servidor é substituído após uma falha. Os usuários de um serviço de transação experimentam algum atraso durante a recuperação. Embora assuma-se que os servidores de transações distribuídas apresentem falhas por colapso e executem em um sistema assíncrono, eles são capazes de chegar a um consenso a respeito do resultado das transações, pois os servidores falhos são substituídos por novos processos que podem adquirir todas as informações relevantes a partir de um meio de armazenamento permanente ou de outros servidores.

Exercícios

- 17.1 Em uma variante descentralizada do protocolo de confirmação de duas fases, os participantes se comunicam diretamente uns com os outros, em vez de indiretamente por meio do coordenador. Na fase 1, o coordenador envia seu voto para todos os participantes. Na fase 2, se o voto do coordenador for *Não*, os participantes apenas cancelam a transação; se for *Sim*, cada participante envia seu voto para o coordenador e para os outros participantes, cada um dos quais decide sobre o resultado, de acordo com o voto, e o executa. Calcule o número de mensagens e o número de rodadas necessárias para isso. Quais são as vantagens ou desvantagens, em comparação com a variante centralizada? página 732

- 17.2 Um protocolo de confirmação de três fases tem as seguintes partes:

Fase 1: é igual à confirmação de duas fases.

Fase 2: o coordenador reúne os votos e toma uma decisão; se ela for *Não*, ele cancela e informa aos participantes que votaram *Sim*; se a decisão for *Sim*, ele envia uma requisição *preCommit* para todos os participantes. Os participantes que votaram em *Sim* esperam por uma requisição *preCommit* ou *doAbort*. Eles reconhecem as requisições *preCommit* e executam requisições *doAbort*.

Fase 3: o coordenador reúne as respostas. Quando todas foram recebidas, ele confirma (com *commit*) e envia a requisição *doCommit* aos participantes. Os participantes esperam por uma requisição *doCommit*. Quando a requisição chega, eles confirmam (com *commit*).

Explique como esse protocolo evita o atraso para os participantes durante seu período de incerto, devido à falha do coordenador ou de outros participantes. Presuma que a comunicação não falhe. página 735

- 17.3 Explique como o protocolo de confirmação de duas fases para transações aninhadas garante que, se a transação de nível superior for confirmada, todas as descendentes corretas serão confirmadas ou canceladas. página 736

- 17.4 Dê um exemplo das interposições de duas transações serialmente equivalentes em cada servidor, mas não serialmente equivalentes globalmente. página 740

- 17.5 O método *getDecision*, definido na Figura 17.4, é fornecido apenas por coordenadores. Defina uma nova versão de *getDecision* para ser fornecida pelos participantes para ser usada por outros participantes que precisem tomar uma decisão quando o coordenador estiver indisponível.

Presuma que qualquer participante ativo possa fazer uma requisição *getDecision* para qualquer outro participante ativo. Isso resolve o problema do atraso durante o período *incerto*? Explique sua resposta. Em que ponto no protocolo de confirmação de duas fases o coordenador informaria aos participantes sobre as identidades dos outros participantes (para permitir essa comunicação)? [página 732](#)

- 17.6 Amplie a definição do travamento de duas fases para aplicar nas transações distribuídas. Explique como isso é garantido pelas transações distribuídas usando travamento de duas fases restrito de forma local. [página 740, Capítulo 16](#)
- 17.7 Supondo que esteja em uso o travamento de duas fases restrito, descreva como as ações do protocolo de confirmação de duas fases se relaciona com as ações de controle de concorrência de cada servidor individual. Como a detecção de impasse distribuído se encaixa nisso? [páginas 732, 740](#)
- 17.8 Um servidor usa ordenação por carimbo de tempo para controle de concorrência local. Quais alterações devem ser feitas para adaptá-lo para uso com transações distribuídas? Sob quais condições poderia ser argumentado que o protocolo de confirmação de duas fases é redundante com ordenação por carimbo de tempo? [páginas 732, 741](#)
- 17.9 Considere o controle de concorrência otimista distribuído, no qual cada servidor realiza validação para trás local, em sequência (isto é, com apenas uma transação por vez na fase de validação e atualização), em relação à sua resposta para o Exercício 17.4. Descreva os possíveis resultados quando as duas transações tentam ser confirmadas. Que diferença fará se os servidores usarem validação em paralelo? [Capítulo 16, página 742](#)
- 17.10 Um detector de impasse global centralizado contém a união dos grafos espera por locais. Dê um exemplo para explicar como um impasse fantasma poderia ser detectado se uma transação em espera em um ciclo de impasse fosse cancelada durante o procedimento de detecção de impasse. [página 745](#)
- 17.11 Considere o algoritmo de caminhamento pelas arestas (sem prioridades). Dê exemplos para mostrar que ele poderia detectar impasses fantasmas. [página 746](#)
- 17.12 Um servidor gerencia os objetos a_1, a_2, \dots, a_n . Ele fornece duas operações para seus clientes:

read(i) retorna o valor de a_i ,
write(i, Value) atribui *Value* a a_i

As transações *T*, *U* e *V* são definidas como segue:

T: *x = read(i); write(j, 44);*
U: *write(i, 55); write(j, 66);*
V: *write(k, 77); write(k, 88);*

Descreva as informações escritas no arquivo de *log* em nome dessas três transações, se for usado o travamento de duas fases restrito e *U* adquirir a_i e a_j antes de *T*. Descreva como o gerenciador de recuperação usaria essas informações para recuperar os efeitos de *T*, *U* e *V*, quando o servidor fosse substituído após uma falha. Qual é o significado da ordem das entradas de confirmação no arquivo de *log*? [páginas 753-754](#)

- 17.13 A anexação de uma entrada no arquivo de *log* é atômica, mas as operações de anexação de diferentes transações podem ser interpostas. Como isso afeta a resposta do Exercício 17.12? [páginas 753-754](#)

- 17.14 As transações T , U e V do Exercício 17.12 usam travamento de duas fases restrito e suas requisições são interpostas como segue:

T	U	V
$x = \text{read}(i);$		
	$\text{write}(k, 77);$	
	$\text{write}(i, 55)$	
$\text{write}(j, 44)$		$\text{write}(k, 88)$
	$\text{write}(j, 66)$	

Supondo que o gerenciador de recuperação, em vez de esperar até o final da transação, anexe imediatamente a entrada de dados correspondente a cada operação $write$ no arquivo de log , descreva as informações escritas no arquivo de log em nome das transações T , U e V . A escrita antecipada afeta a correção do procedimento de recuperação? Quais são as vantagens e desvantagens da gravação antecipada?

páginas 753–754

- 17.15 As transações T e U são executadas com controle de concorrência com ordenação por carimbo de tempo. Descreva as informações gravadas no arquivo de log em nome de T e U , admitindo o fato de que U tem um carimbo de tempo posterior a T e que deve esperar ser confirmada após T . Por que é fundamental que as entradas de confirmação no arquivo de log sejam ordenadas pelos carimbos de tempo? Descreva o efeito da recuperação caso o servidor falhe (i) entre as duas operações *Commit* e (ii) após as duas operações.

T	U
$x = \text{read}(i);$	
	$\text{write}(i, 55);$
	$\text{write}(j, 66);$
$\text{write}(j, 44);$	
	commit
commit	

Quais as vantagens e desvantagens da escrita antecipada com ordenação por carimbo de tempo?

páginas 757

- 17.16 As transações T e U do Exercício 17.15 são executadas com controle de concorrência otimista usando validação para trás e reiniciando as transações que falham. Descreva as informações escritas no arquivo de log em nome delas. Por que é fundamental que as entradas de confirmação no arquivo de log sejam ordenadas pelos números de transação? Como os conjuntos de escrita das transações confirmadas são representados no arquivo de log ?

páginas 753–754

- 17.17 Suponha que o coordenador de uma transação falhe após ter gravado a entrada da lista de intenções, mas antes de ter escrito a lista de participantes ou enviado as requisições de *canCommit*? Descreva como os participantes resolvem a situação. O que o coordenador fará quando recuperar? Seria melhor escrever a lista de participantes antes da entrada da lista de intenções?

página 758

18

Replicação

- 18.1 Introdução
- 18.2 Modelo de sistema e o papel da comunicação em grupo
- 18.3 Serviços tolerantes a falhas
- 18.4 Estudos de caso de serviços de alta disponibilidade: Gossip, Bayou e Coda
- 18.5 Transações com replicação de dados
- 18.6 Resumo

A replicação é a chave para prover alta disponibilidade e tolerância a falhas em sistemas distribuídos. A alta disponibilidade é um tópico de crescente interesse, principalmente com a atual tendência em direção à computação móvel e à operação desconectada. A tolerância a falhas é uma preocupação permanente dos serviços fornecidos em sistemas nos quais a segurança funcional (*safety*) é fundamental e em outros tipos importantes de serviços e sistemas.

A primeira parte deste capítulo considera os sistemas que aplicam, em um dado momento, uma única operação em conjuntos de objetos replicados. Ele começa com uma descrição dos componentes da arquitetura e um modelo de sistema para serviços que empregam replicação. Descreveremos a implementação do gerenciamento dos membros de um grupo como parte da comunicação em grupo, que é particularmente importante para serviços tolerantes a falhas.

Em seguida, o capítulo descreve estratégias para se obter tolerância a falhas. Ele apresentará os critérios da correção da capacidade de linearização e da consistência sequencial. Depois, serão apresentadas e discutidas duas estratégias: a replicação passiva (*backup* primário), na qual os clientes se comunicam com uma réplica distinta, e a replicação ativa, em que os clientes se comunicam com todas as réplicas por meio de *multicast*.

Serão considerados estudos de caso de três sistemas para serviços de alta disponibilidade. Nas arquiteturas Gossip e Bayou, as atualizações são lentamente propagadas entre réplicas de dados compartilhados. Na arquitetura Bayou, é usada a técnica da transformação operacional para impor a consistência. O Coda é um exemplo de serviço de arquivo altamente disponível.

O capítulo termina considerando as transações – sequências de operações – sobre objetos replicados. Ele considerará as arquiteturas de sistemas transacionais replicados e como esses sistemas tratam de falhas de servidor e do particionamento da rede.

18.1 Introdução

Neste capítulo, estudaremos a replicação de dados: a manutenção de cópias dos dados em vários computadores. A replicação é o segredo da eficácia dos sistemas distribuídos, pois pode fornecer melhor desempenho, alta disponibilidade e tolerância a falhas. A replicação é amplamente usada. Por exemplo, o armazenamento de recursos de servidores Web na cache dos navegadores e em servidores *proxies* Web é uma forma de replicação, pois os dados mantidos nas caches e nos *proxies* são réplicas uns dos outros. O serviço de atribuição de nomes DNS, descrito no Capítulo 13, mantém cópias de mapeamentos entre nomes e atributos dos computadores e é fundamental para o acesso diário a serviços pela Internet.

A replicação é uma técnica para melhorar *serviços*. As motivações para a replicação são: melhorar o desempenho de um serviço, aumentar sua disponibilidade ou torná-lo tolerante a falhas.

Melhoria de desempenho: a colocação dos dados em cache de clientes e servidores é uma maneira conhecida de melhorar o desempenho. Por exemplo, o Capítulo 2 mencionou que navegadores e servidores *proxies* colocam em cache cópias de recursos Web para evitar a latência da busca desses recursos no servidor de origem. Além disso, às vezes os dados são replicados de forma transparente entre vários servidores de origem no mesmo domínio. A carga de trabalho é compartilhada entre esses servidores por meio da vinculação de seus endereços IP ao nome DNS do *site*, digamos *www.aWebSite.org*. Uma pesquisa de DNS de *www.aWebSite.org* resulta no retorno de um dos vários endereços IP de servidores, em um sistema de rodízio (veja a Seção 13.2.3). Para serviços mais complexos, baseados em dados replicados entre milhares de servidores, são necessárias estratégias de平衡amento de carga mais sofisticadas. Como exemplo, Dilley *et al.* [2002] descrevem a estratégia de resolução de nomes adotada na rede de distribuição de conteúdo Akamai.

A replicação de dados imutáveis é simples: ela aumenta o desempenho com pouco custo para o sistema. A replicação de dados mutáveis (dinâmicos), como os da Web, acarreta sobrecargas nos protocolos para garantir que os clientes recebam dados atualizados (veja a Seção 2.3.1). Assim, existem limites para a eficácia da replicação como uma técnica de melhoria de desempenho.

Maior disponibilidade: os usuários exigem que os serviços sejam de alta disponibilidade, isto é, a proporção do tempo durante a qual o serviço está acessível com tempo de resposta razoável deve ser próxima a 100%. Desconsiderando os atrasos decorrentes dos conflitos do controle de concorrência pessimista (devido ao travamento de dados), os fatores relevantes para a alta disponibilidade são:

- falhas do servidor;
- particionamento da rede e operação desconectada: as desconexões da comunicação, que frequentemente não são planejadas, são um efeito colateral da mobilidade do usuário.

Para considerarmos o primeiro deles, a replicação é uma técnica para manter automaticamente a disponibilidade dos dados, a despeito das falhas do servidor. Se os dados são replicados em dois ou mais servidores que falham independentemente, então o *software* cliente pode acessar os dados em um servidor alternativo, caso o servidor padrão falhe ou se torne inacessível. Dessa forma, a porcentagem do tempo durante a qual o *serviço* está disponível pode ser melhorada pela replicação dos dados do servidor. Se cada um de n servidores tiver uma probabilidade inde-

pendente p de falhar, ou se tornar inacessível, então a disponibilidade de um objeto armazenado em cada um desses servidores é de:

$$1 - \text{probabilidade (todos os gerenciadores falharem ou se tornarem inacessíveis)} = 1 - p^n$$

Por exemplo, se existe uma probabilidade de 5% de falha de qualquer servidor individual, em determinado período de tempo, e se existem dois servidores, então a disponibilidade é de $1 - 0,05^2 = 1 - 0,0025 = 99,75\%$. Uma diferença importante entre os sistemas que usam cache e a replicação do servidor é que as caches não contêm necessariamente conjuntos de objetos, como arquivos, em sua totalidade. Portanto, o uso de cache não melhora necessariamente a disponibilidade no nível da aplicação – um usuário pode ter um arquivo necessário, mas não outro.

O particionamento da rede (veja a Seção 15.1) e a operação desconectada são o segundo fator que vai contra a alta disponibilidade. Os usuários móveis desconectam deliberadamente seus computadores ou, ao se moverem, podem ser desconectados involuntariamente de uma rede sem fio. Por exemplo, um usuário em um trem, portando um *notebook*, pode não ter acesso à rede (a interligação em uma rede sem fio pode ser interrompida ou pode não existir essa capacidade disponível). Para poder trabalhar nessas circunstâncias – o assim chamado *trabalho desconectado* ou *operação desconectada* –, o usuário se preparará, copiando os dados muito utilizados, como o conteúdo de uma agenda compartilhada, de seu ambiente normal de trabalho para o *notebook*. Contudo, frequentemente, existe uma contrapartida com relação à disponibilidade durante tal período de desconexão: quando o usuário consulta ou atualiza a agenda, corre o risco de ler dados que foram alterados por outra pessoa nesse meio-tempo. Por exemplo, ele pode marcar um compromisso em uma hora já ocupada. O trabalho desconectado só será possível se o usuário (ou a aplicação, em nome do usuário) puder aceitar dados antigos e resolver posteriormente os conflitos que surgirem.

Tolerância a falhas: dados de alta disponibilidade não são necessariamente dados rigorosamente corretos. Eles podem estar desatualizados, por exemplo, ou dois usuários em lados opostos de uma rede que foi particionada podem fazer atualizações conflitantes e que precisem ser resolvidas. Em contraste, um serviço tolerante a falhas sempre garante comportamento rigorosamente correto, apesar de certo número e tipo de falhas. A correção está relacionada ao caráter atual dos dados fornecidos para o cliente e aos efeitos das operações do cliente sobre os dados. Às vezes, a correção também está relacionada ao tipo de respostas do serviço – como, por exemplo, no caso de um sistema de controle de tráfego aéreo, no qual dados corretos são necessários em escalas de tempo curtas.

A mesma técnica básica usada para alta disponibilidade – de replicação de dados e funcionalidade entre computadores – também pode ser aplicada para se obter tolerância a falhas. Se até f de $f + 1$ servidores falham, então, em princípio, pelo menos um permanece para fornecer o serviço. E se até f servidores podem apresentar falhas bizantinas, então, em princípio, um grupo de $2f + 1$ servidores pode fornecer um serviço correto, fazendo-se com que os servidores corretos vençam por voto os servidores falhos (que podem fornecer valores espúrios). No entanto, a tolerância a falhas é mais útil do que essa descrição simples faz parecer. O sistema precisa gerenciar a coordenação de seus componentes precisamente para manter as garantias de correção diante de falhas, as quais podem ocorrer a qualquer momento.

Um requisito comum quando dados são replicados é a *transparência da replicação*, isto é, normalmente, os clientes não devem saber que existem várias cópias físicas dos dados. No

que diz respeito a eles, os dados são organizados como objetos *lógicos* individuais e identificam apenas um item em cada caso, quando solicitam a execução de uma operação. Além disso, os clientes esperam que as operações retornem apenas um conjunto de valores, apesar de as operações poderem ser executadas sobre mais de uma cópia física de comum acordo.

O outro requisito geral para dados replicados – que pode variar com os níveis de rigor exigidos pelos aplicativos – é a consistência. Isso diz respeito ao fato de as operações executadas sobre um conjunto de objetos replicados produzirem resultados que satisfaçam a especificação da correção desses objetos.

Vimos, no exemplo da agenda, que durante a operação desconectada os dados podem se tornar inconsistentes, pelos menos temporariamente. Porém, quando os clientes permanecem conectados, frequentemente não é aceitável que diferentes clientes (usando diferentes cópias físicas dos dados) obtenham resultados inconsistentes ao fazerem requisições que afetem os mesmos objetos lógicos. Isto é, isso não é aceitável se os resultados violarem os critérios de correção do aplicativo.

Examinaremos agora, com mais detalhes, os problemas de projeto que surgem quando replicamos dados para obter serviços de alta disponibilidade e tolerantes a falhas. Examinaremos também algumas soluções e técnicas padrão para tratar desses problemas. Primeiramente, as Seções 18.2 a 18.4 abordarão o caso em que os clientes fazem invocações individuais sobre dados compartilhados. A Seção 18.2 apresentará uma arquitetura geral para gerenciamento de dados replicados e apresentará a comunicação em grupo como uma ferramenta importante. A comunicação em grupo é particularmente útil para se obter tolerância a falhas, que será o assunto da Seção 18.3. A Seção 18.4 descreverá técnicas para se obter alta disponibilidade, incluindo a operação desconectada. Ele incluirá estudos de caso da arquitetura Gossip, Bayou e o sistema de arquivos Coda. A Seção 18.5 examinará como se faz para suportar transações em dados replicados. Conforme os Capítulos 16 e 17 explicaram, as transações são constituídas de sequências de operações, em vez de operações simples.

18.2 Modelo de sistema e o papel da comunicação em grupo

Os dados de nosso sistema consistem em um conjunto de itens que chamamos de objetos. Um objeto poderia ser, digamos, um arquivo ou um objeto Java. Contudo, cada objeto *lógico* é implementado por um conjunto de cópias *físicas* chamadas de *réplicas*. As réplicas são objetos físicos, cada um armazenado em um único computador, com dados e comportamento ligados até certo grau de consistência pela operação do sistema. As réplicas de determinado objeto não são necessariamente idênticas, pelo menos não em um ponto em particular no tempo. Algumas réplicas podem ter recebido atualizações que as outras não receberam.

Nesta seção, forneceremos um modelo de sistema geral para gerenciar réplicas e, depois, descreveremos o papel dos sistemas de comunicação em grupo na obtenção de tolerância a falhas por meio de replicação, destacando a importância da comunicação em grupo com modo de visualização síncrono.

18.2.1 Modelo de sistema

Suponhamos um sistema assíncrono em que os processos só podem falhar por colapso. Nossa suposição padrão é a de que não pode ocorrer particionamento da rede, mas às vezes consideraremos o que acontece se eles ocorrem. O particionamento da rede torna mais difícil construir detectores de falha, os quais usamos para obter *multicast* confiável e totalmente ordenado.

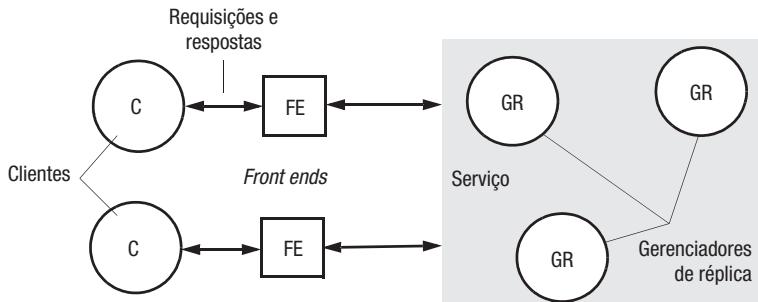


Figura 18.1 Um modelo de arquitetura básico para o gerenciamento de dados replicados.

Tendo em vista a generalidade, descreveremos os componentes da arquitetura por meio de suas funções, o que não implica que eles devam ser necessariamente implementados por processos (ou *hardware*) distintos. O modelo envolve réplicas mantidas por *gerenciadores de réplica* distintos (veja a Figura 18.1), que são componentes que contêm as réplicas em determinado computador e executam operações diretamente sobre elas. Esse modelo geral pode ser aplicado a um ambiente cliente-servidor, no caso em que um gerenciador de réplica é um servidor. Às vezes, os chamaremos simplesmente de servidores. Do mesmo modo, ele pode ser empregado para aplicativos e, nesse caso, os processos aplicativos podem atuar como clientes e gerenciadores de réplica. Por exemplo, o *notebook* do usuário em um trem pode conter um aplicativo que atue como gerenciador de réplica para sua agenda.

Sempre exigimos que um gerenciador de réplica aplique operações em suas réplicas de forma recuperável. Isso nos permite presumir que uma operação em um gerenciador de réplica não deixa resultados inconsistentes, caso venha a falhar no meio do processo. Às vezes, exigimos que cada gerenciador de réplica seja uma *máquina de estados* [Lamport 1978, Schneider 1990]. Tal gerenciador de réplica aplica operações em suas réplicas de forma atômica (indivisível), de modo que sua execução é equivalente a efetuar operações em alguma sequência restrita. Além disso, o estado de suas réplicas é uma função determinística de seus estados iniciais e da sequência de operações nelas aplicadas. Outros estímulos, como a leitura em um relógio, ou de um sensor associado, não têm relação com esses valores de estado. Sem essa suposição, não poderiam ser dadas garantias de consistência entre os gerenciadores de réplica que aceitam operações de atualização independentemente. O sistema só pode determinar quais operações serão aplicadas em todos os gerenciadores de réplica, e em que ordem – ele não pode reproduzir efeitos não determinísticos. A suposição implica que, dependendo do suporte da arquitetura para *threads*, não é possível que os servidores sejam *multithreadeds*.

Frequentemente, cada gerenciador de réplica mantém uma réplica de cada objeto, e presumimos que isso é verdade, a não ser que digamos o contrário. No entanto, em geral, as réplicas de diferentes objetos podem ser mantidas por diferentes conjuntos de gerenciadores de réplica. Por exemplo, um objeto pode ser necessário para clientes de uma rede e outro objeto para clientes de outra rede. Não há muita vantagem em replicá-los nos gerenciadores de ambas as redes.

O conjunto de gerenciadores de réplica pode ser estático ou dinâmico. Em um sistema dinâmico, novos gerenciadores de réplica podem aparecer (por exemplo, uma segunda secretária copia uma agenda em seu *notebook*); isso não é permitido em um sistema estático. Em um sistema dinâmico, os gerenciadores de réplica podem falhar e, então, eles são considerados como tendo deixado o sistema (embora possam ser substituídos).

Em um sistema estático, os gerenciadores de réplica não falham (a falha implica em *nunca* executar outro passo), mas deixam de funcionar por um período indefinido. Voltaremos ao problema de falhas na Seção 18.4.2.

O modelo geral de gerenciamento de réplicas aparece na Figura 18.1. Um conjunto de gerenciadores de réplica fornece um serviço para os clientes. Os clientes veem um serviço que dá acesso aos objetos (por exemplo, agendas ou contas bancárias), os quais, na verdade, estão replicados nos gerenciadores. Cada cliente solicita uma série de operações – invocações sobre um ou mais objetos. Uma operação envolve uma combinação de leituras e atualizações em objetos. As operações solicitadas que não envolvem atualizações são chamadas de *requisições somente de leitura*; as operações solicitadas que atualizam um objeto são chamadas de *requisições de atualização* (elas também podem envolver leituras).

As requisições de cada cliente são primeiramente manipuladas por um componente chamado *front-end*. A função do *front-end* é se comunicar, por meio da passagem de mensagens, com um ou mais gerenciadores de réplica, em vez de obrigar o cliente a fazer isso sozinho, explicitamente. Ele é o meio para tornar a replicação transparente. Um *front-end* pode ser implementado no espaço de endereçamento do cliente ou pode ser um processo separado.

Em geral, cinco fases afetam o desempenho de uma única requisição sobre objetos replicados [Wiesmann *et al.* 2000]. As ações em cada fase variam de acordo com o tipo de sistema, conforme se tornará claro nas duas próximas seções. Por exemplo, um serviço que suporta operação desconectada se comporta de modo diferente de outro que fornece um serviço tolerante a falhas. As fases são as seguintes:

Requisição: o *front-end* emite a requisição para um ou mais gerenciadores de réplica:

- ou o *front-end* se comunica com um único gerenciador de réplica, o qual, por sua vez, se comunica com outros gerenciadores de réplica;
- ou o *front-end* faz um *multicast* da requisição para os gerenciadores de réplica.

Coordenação: os gerenciadores de réplica fazem a coordenação, preparando-se para executar a requisição consistentemente. Se necessário, nesse estágio eles concordam se a requisição deve ser aplicada (ela pode não ser aplicada, caso ocorram falhas nesse estágio). Eles também decidem sobre a ordem dessa requisição em relação as outras. Todos os tipos de ordenação definidos para *multicast* na Seção 15.4.3 também se aplicam ao tratamento da requisição, e definimos essas requisições novamente para este contexto:

Ordem FIFO: se um *front-end* emite uma requisição r e depois uma r' , então um gerenciador de réplica correto que trate r' , processará r antes dela.

Ordem causal: se a emissão da requisição r aconteceu antes da emissão de r' , então um gerenciador de réplica correto que trate r' , processará r antes dela.

Ordem total: se um gerenciador de réplica correto tratar r antes da requisição r' , então um gerenciador de réplica correto que trate r' , processará r antes dela.

A maioria das aplicações exige a ordem FIFO. Discutiremos os requisitos da ordem causal e total – e as ordens mistas que são FIFO e totais ou causais e totais – nas duas próximas seções.

Execução: os gerenciadores de réplica executam a requisição – talvez, por tentativas; isto é, de maneira tal que possam desfazer seus efeitos posteriormente.

Acordo: os gerenciadores de réplica chegam a um consenso sobre o efeito da requisição – se houver – que será confirmado. Por exemplo, em um sistema transa-

cional, os gerenciadores de réplica podem concordar coletivamente com o cancelamento ou com a confirmação da transação nesse estágio.

Resposta: um ou mais gerenciadores de réplica respondem ao *front-end*. Em alguns sistemas, apenas um gerenciador de réplica envia a resposta. Em outros, o *front-end* recebe respostas de um conjunto de gerenciadores de réplica e seleciona, ou sintetiza, uma única resposta para enviar ao cliente. Por exemplo, ele poderia enviar a primeira resposta a chegar, se o objetivo fosse a alta disponibilidade. Se o objetivo fosse tolerância a falhas bizantinas, ele poderia dar ao cliente a resposta fornecida pela maioria dos gerenciadores de réplica.

Diferentes sistemas podem fazer escolhas distintas sobre a ordem das fases, assim como sobre seu conteúdo. Por exemplo, em um sistema que suporta operação desconectada, é importante fornecer ao cliente (digamos, o aplicativo no *notebook* do usuário) uma resposta o mais cedo possível. O usuário não quer esperar até que o gerenciador de réplica no *notebook* e o gerenciador de réplica no escritório possam fazer a coordenação. Em contraste, em um sistema tolerante a falhas, o cliente não recebe a resposta até o fim, quando a correção do resultado pode ser garantida.

18.2.2 O papel da comunicação em grupo

O Capítulo 6 apresentou o conceito de comunicação em grupo e a Seção 15.4 expandiu essa discussão, abordando algoritmos para confiabilidade e ordenação de entrega de mensagens em sistemas de comunicação em grupo. Neste capítulo, examinaremos o papel dos grupos no gerenciamento de dados replicados. A discussão da Seção 15.4 considerou a participação como membro de grupos como sendo definida estaticamente, embora os membros do grupo pudessem falhar. Na replicação e, na verdade, em muitas outras circunstâncias práticas, há o forte requisito de participação dinâmica, na qual os processos entram e saem do grupo à medida que o sistema executa. Em um serviço que gerencia dados replicados, por exemplo, os usuários podem adicionar ou retirar um gerenciador de réplica, ou um gerenciador de réplica pode falhar e, assim, precisar ser retirado da operação do sistema. Portanto, o gerenciamento da participação como membro do grupo, apresentado na Seção 6.2.2, é particularmente importante nesse contexto.

Os sistemas que podem se adaptar à medida que os processos entram, saem e falham – sistemas tolerantes a falhas, em particular – exigem os recursos mais avançados de detecção e notificação de falhas nas alterações da participação como membro. Um serviço de participação como membro do grupo completo mantém *modos de visualização do grupo*, que são listas dos membros correntes do grupo, identificados pelos seus identificadores de processo exclusivos. A lista é ordenada, por exemplo, de acordo com a sequência na qual os membros entraram no grupo. Um novo modo de visualização de grupo é gerado sempre que um processo é adicionado ou excluído.

É importante entender que um serviço de participação como membro do grupo pode excluir um processo de um grupo porque ele é *Suspeito*, mesmo que possa não ter falhado. Uma falha de comunicação pode ter tornado o processo inacessível, enquanto ele continua a ser executado normalmente. Um serviço de participação como membro sempre está livre para excluir tal processo. O efeito da exclusão é que, daí em diante, nenhuma mensagem será enviada para esse processo. Além disso, no caso de um grupo fechado, se esse processo se conectar novamente, todas as mensagens que ele tentar enviar não serão enviadas para os membros do grupo. Esse processo terá que entrar novamente no grupo (como uma “reencarnação” de si mesmo, com um novo identificador) ou cancelar suas operações.

Uma falsa suspeita de um processo, e sua consequente exclusão do grupo, pode reduzir a eficácia do grupo. O grupo precisa sair-se bem sem a confiabilidade, ou o desempenho extra, que o processo retirado possa ter fornecido. O desafio de projeto, além de projetar detectores de falhas mais precisos possível, é garantir que um sistema baseado na comunicação em grupo não se comporte *incorrectamente* se um processo for falsamente suspeito.

Uma consideração importante é como o serviço de gerenciamento de grupo trata o particionamento da rede. A desconexão, ou a falha de componentes, como um roteador em uma rede, podem dividir um grupo de processos em dois ou mais subgrupos, de modo que a comunicação entre os subgrupos seja impossível. Os serviços de gerenciamento de grupo diferem no fato de serem de *particionamento primário* ou *particionáveis*. No primeiro caso, o serviço de gerenciamento permite que no máximo um subgrupo (uma maioria) sobreviva a um particionamento; os processos restantes são informados de que devem suspender as operações. Esse arranjo é apropriado para os casos em que os processos gerenciam dados importantes e os custos das inconsistências entre dois ou mais subgrupos superam qualquer vantagem do trabalho desconectado.

Por outro lado, em algumas circunstâncias é aceitável que dois ou mais subgrupos continuem a operar, e um serviço de gerenciamento de grupo do tipo particionável permite isso. Por exemplo, em uma aplicação em que os usuários participam de uma conferência por áudio ou vídeo para discutir alguns problemas, pode ser aceitável que dois ou mais subgrupos de usuários continuem suas discussões independentemente, a despeito de uma divisão. Eles podem reunir seus resultados quando o particionamento for sanado e os subgrupos forem novamente conectados.

Entrega de modo de visualização • Considere a tarefa de um programador escrevendo uma aplicação que é executada em um grupo que deve aceitar membros novos e perdidos. O programador precisa saber se o sistema trata cada membro de uma maneira consistente quando a participação como membro muda. Seria complicado se o programador tivesse que consultar o estado de todos os outros membros e chegar a uma decisão global quando ocorresse uma alteração da participação dos membros, em vez de poder tomar uma decisão local sobre como responder à alteração. A vida do programador se torna mais difícil, ou mais fácil, de acordo com as garantias que se aplicam quando o sistema repassa modos de visualização para os membros do grupo.

Para cada grupo g , o serviço de gerenciamento de grupo repassa para qualquer processo membro $p \in g$, uma série de modos de visualização $v_0(g)$, $v_1(g)$, $v_2(g)$, etc. Por exemplo, uma série de modos de visualização poderia ser $v_0(g) = (p)$, $v_1(g) = (p, p')$ e $v_2(g) = (p) - p$ entra em um grupo vazio, depois p' entra no grupo, em seguida p' sai dele. Embora várias alterações da participação de membros possam ocorrer paralelamente, como quando um processo entra no grupo exatamente quando outro sai, o sistema impõe uma ordem na sequência de modos de visualização fornecida para cada processo.

Falamos sobre um membro *repassando um modo de visualização* quando ocorre uma alteração na participação de membros e o aplicativo é notificado da nova participação – exatamente como falamos sobre um processo entregando uma mensagem *multicast*. Assim como na entrega por *multicast*, a entrega de um modo de visualização é diferente do seu recebimento. Os protocolos de participação como membro de grupo mantêm os modos de visualização propostos em uma fila de retenção até que todos os membros existentes possam concordar com sua entrega.

Também falamos sobre um evento ocorrendo em um *modo de visualização* $v(g)$ no processo p , se no momento da ocorrência do evento, p tiver entregado $v(g)$, mas ainda não tiver entregado o próximo modo de visualização $v'(g)$.

Alguns requisitos básicos da entrega de modos de visualização são os seguintes:

Ordem: se um processo p entrega o modo de visualização $v(g)$ e depois o modo de visualização $v'(g)$, então nenhum outro processo $q \neq p$ entrega $v'(g)$ antes de $v(g)$.

Integridade: se o processo p entrega o modo de visualização $v(g)$, então $p \in v(g)$.

Não trivialidade: se o processo q entra em um grupo e é, ou se torna acessível, a partir de um processo $p \neq q$, então q estará sempre nos modos de visualização entregues por p . Analogamente, se o grupo sofre um particionamento e assim permanece, então os modos de visualização entregues por uma partição da rede excluirão os processos de outra partição.

O primeiro desses requisitos está relacionado ao fato de dar ao programador uma garantia de consistência, certificando-se de que as alterações no modo de visualização sempre ocorrem na mesma ordem em diferentes processos. O segundo requisito é uma “verificação de sanidade mental”. O terceiro previne-se contra soluções triviais. Por exemplo, um serviço de participação de membro que informa a cada processo, independentemente de sua conectividade, que ele está sozinho em um grupo, não tem muito interesse. A condição de não trivialidade diz que, se dois processos que entraram no mesmo grupo podem se comunicar, então cada um deles deve ser considerado membro desse mesmo grupo. Analogamente, ela exige que, quando ocorre um particionamento, o serviço de participação como membro deve refletir essa divisão. A condição não diz como o serviço de participação como membro do grupo deve se comportar no caso problemático da conectividade intermitente.

Comunicação em grupo com modo de visualização síncrono • Um sistema de comunicação em grupo com *modo de visualização síncrono* dá garantias adicionais, em relação às anteriores, sobre a ordem de distribuição das notificações de modo de visualização, a respeito da entrega de mensagens *multicast*. A comunicação com modo de visualização síncrono amplia a semântica de *multicast* confiável descrita no Capítulo 15 para levar em conta as mudanças nos modos de visualização de grupo. Por simplicidade, restringiremos nossa discussão ao caso no qual não podem ocorrer particionamentos. As garantias oferecidas pela comunicação em grupo com modo de visualização síncrono são as seguintes.

Acordo: os processos corretos entregam a mesma sequência de modos de visualização (começando a partir do modo de visualização em que eles entram no grupo) e o mesmo conjunto de mensagens em qualquer modo de visualização dado. Isto é, se um processo correto entrega a mensagem m no modo de visualização $v(g)$, então todos os outros processos corretos que entregam m também fazem isso no modo de visualização $v(g)$.

Integridade: se um processo correto p entrega a mensagem m , então ele não entregará m novamente. Além disso, $p \in \text{grupo}(m)$ e o processo que enviou m está no modo de visualização em que p entrega m .

Validade (grupos fechados): os processos corretos sempre entregam as mensagens que enviam. Se o sistema deixar de entregar uma mensagem para qualquer processo q , ele notificará os processos sobreviventes entregando um novo modo de visualização com q excluído. Isto é, seja p qualquer processo correto que entrega a mensagem m no modo de visualização $v(g)$. Se algum processo $q \in v(g)$ não entregar m no modo de visualização $v(g)$, então o próximo modo de visualização $v'(g)$ entregue por p terá $q \notin v'(g)$.

Considere um grupo com três processos p, q e r (veja a Figura 18.2). Suponha que p envie uma mensagem m enquanto está no modo de visualização (p, q, r) , mas que p falhe logo após enviar m , enquanto q e r estão corretos. Uma possibilidade é a de que p falhe antes

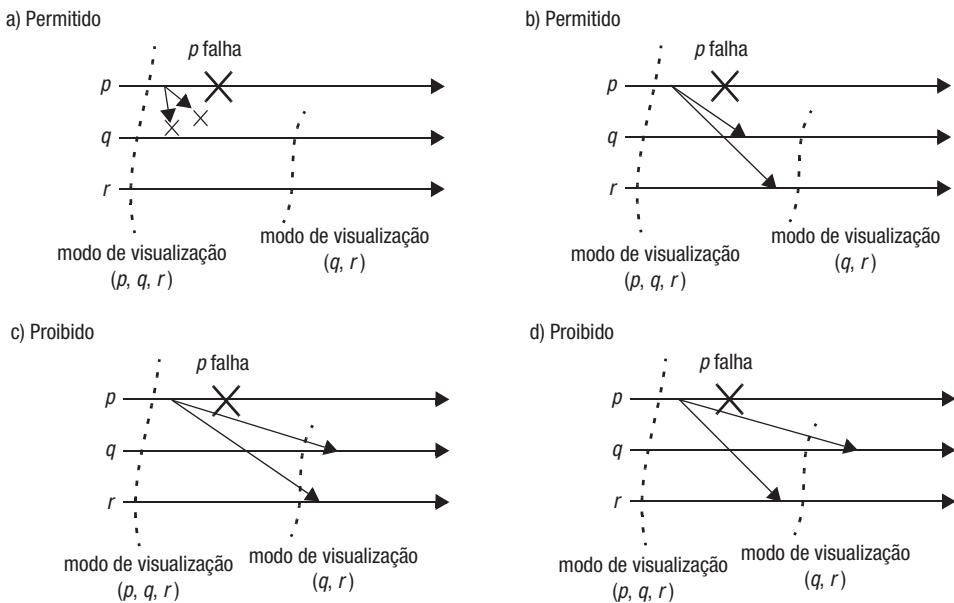


Figura 18.2 Comunicação em grupo com modo de visualização síncrono.

de m ter chegado a qualquer outro processo. Nesse caso, q e r entregam, cada um, o novo modo de visualização (q, r) , mas nenhum entregará m (Figura 18.2a). A outra possibilidade é a de que m tenha chegado a pelo menos um dos dois processos sobreviventes, quando p falha. Então, q e r entregam primeiro m e depois o modo de visualização (q, r) (Figura 18.2b). Não é permitido que q e r entreguem primeiro o modo de visualização (q, r) e depois m (Figura 18.2c), porque senão entregariam uma mensagem de um processo sobre o qual foram informados de que falhou; os dois também não podem entregar a mensagem e o novo modo de visualização em ordens opostas (Figura 18.2d).

Em um sistema com modo de visualização síncrono, a entrega de um novo modo de visualização define uma linha imaginária no sistema e toda mensagem entregue é consistentemente distribuída em um lado ou no outro dessa linha. Isso permite ao programador tirar conclusões úteis a respeito do conjunto de mensagens que outros processos corretos entregaram ao distribuir um novo modo de visualização, baseado apenas na ordem local dos eventos de entrega de mensagem e de entrega de modo de visualização.

Uma ilustração da utilidade da comunicação de modo de visualização síncrono é como ela pode ser usada para obter *transferência de estado* – a transferência do estado de trabalho de um membro corrente de um grupo de processos, para um novo membro do grupo. Por exemplo, se os processos são gerenciadores de réplica, cada um contendo o estado de uma agenda, então um gerenciador de réplica que venha a participar do grupo dessa agenda precisará adquirir o estado corrente da agenda ao entrar. Contudo, a agenda pode ser atualizada enquanto o estado está sendo capturado. É importante que o gerenciador de réplica não perca as mensagens de atualização não refletidas no estado adquirido e que não reaplique as mensagens de atualização já refletidas no estado (a não ser as atualizações idempotentes).

Para obter essa transferência de estado, podemos usar comunicação de modo de visualização síncrono em um esquema simples, como o seguinte. Na entrega do primeiro

modo de visualização contendo o novo processo, algum processo distinto dos membros previamente existentes – digamos, o mais antigo – captura seu estado, o envia para o novo membro no modo um para um e suspende sua execução. Todos os outros processos previamente existentes suspendem sua execução. Note que, precisamente, o conjunto de atualizações refletidas nesse estado foi, por definição, aplicado em todos os outros membros. Ao receber o estado, o novo processo o integra e envia uma mensagem *multicast* de “começar!” para o grupo, no ponto em que tudo prossegue mais uma vez.

Discussão • A noção de comunicação em grupo com modo de visualização síncrono apresentada é uma formulação do paradigma da comunicação virtualmente síncrona, desenvolvida originalmente no sistema ISIS [Birman 1993, Birman *et al.* 1991, Birman e Joseph 1987b]. Schiper e Sandoz [1993] descrevem um protocolo para obter comunicação de modo de visualização síncrono (ou, conforme eles o chamam, *modo de visualização atômico*). Note que um serviço de participação como membro do grupo chega ao consenso, mas faz isso considerando o resultado da impossibilidade de Fischer *et al.* [1985]. Conforme discutimos na Seção 15.5.4, um sistema pode contornar esse resultado usando um detector de falha apropriado.

Schiper e Sandoz também fornecem uma versão uniforme da comunicação de modo de visualização síncrono, na qual a condição de acordo abrange o caso dos processos que falham. Isso é semelhante ao acordo uniforme da comunicação *multicast*, que descrevemos na Seção 15.4.2. Na versão uniforme da comunicação de modo de visualização síncrono, mesmo que um processo falhe após entregar uma mensagem, todos os processos corretos são obrigados a entregar a mensagem no mesmo modo de visualização. Às vezes, essa garantia mais forte é necessária em aplicativos tolerantes a falhas, pois um processo que entregou uma mensagem pode ter afetado o mundo exterior antes de falhar. Pelo mesmo motivo, Hadzilacos e Toueg [1994] consideraram versões uniformes dos protocolos de *multicast* confiáveis e ordenados, descritos no Capítulo 15.

O sistema V [Cheriton e Zwaenepoel 1985] foi o primeiro a incluir suporte para grupos de processos. Após o ISIS, foram desenvolvidos grupos de processos com algum tipo de serviço de participação como membro do grupo em vários outros sistemas, incluindo o Horus [van Renesse *et al.* 1996], o Totem [Moser *et al.* 1996] e o Transis [Dolev e Malki 1996].

Variações do sincronismo de modo de visualização foram propostas para serviços de participação como membro de grupo divisível, incluindo suporte para aplicativos com reconhecimento de particionamentos [Babaoglu *et al.* 1998] e sincronismo virtual estendido [Moser *et al.* 1994].

Finalmente, Cristian [1991b] discute um serviço de participação de membro de grupo para sistemas distribuídos síncronos.

18.3 Serviços tolerantes a falhas

Nesta seção, examinaremos como se faz para fornecer um serviço correto a despeito de até f falhas de processo por meio da replicação de dados e da funcionalidade dos gerenciadores de réplica. Por simplicidade, supomos que a comunicação permanece confiável e que não ocorrem particionamentos na rede.

Supõe-se que cada gerenciador de réplica se comporta de acordo com uma especificação da semântica dos objetos que gerencia, quando não tiver falhado. Por exemplo, uma especificação de contas bancárias incluiria a garantia de que os fundos transferidos entre as contas nunca poderia desaparecer, e que apenas os depósitos e os saques afetariam o saldo de uma conta em particular.

Intuitivamente, um serviço baseado em replicação é correto se continua respondendo a despeito das falhas e se os clientes não conseguem identificar a diferença entre o serviço que obtém de uma implementação com dados replicados e de um fornecido por um único gerenciador de réplica correto. É preciso cuidado para satisfazer esse critério. Se não forem tomadas precauções, poderão surgir anomalias quando houver vários gerenciadores de réplica – mesmo lembrando que estamos considerando os efeitos de operações individuais e não de transações.

Considere um sistema de replicação simples, no qual dois gerenciadores de réplica nos computadores A e B mantêm, cada um, réplicas de duas contas bancárias x e y . Os clientes leem e atualizam as contas em seus gerenciadores de réplica locais, mas tentam usar outro gerenciador de réplica, se o local falha. Os gerenciadores de réplica propagam em segundo plano (*background*) as atualizações entre si, após responderem ao cliente. As duas contas têm inicialmente um saldo de \$0.

O cliente 1 atualiza para \$1 o saldo de x em seu gerenciador de réplica local B e, depois, tenta atualizar para \$2 o saldo de y , mas descobre que B falhou. Portanto, em vez disso, o cliente 1 aplica a atualização em A . Agora, o cliente 2 lê os saldos em seu gerenciador de réplica local A e descobre primeiro que y tem \$2 e depois que x tem \$0 – a atualização de B na conta bancária x não chegou, pois B falhou. A situação está mostrada a seguir, e as operações são rotuladas de acordo com o computador em que elas ocorreram primeiro:

Cliente 1:	Cliente 2:
$setBalance_B(x, 1)$	
$setBalance_A(y, 2)$	
	$getBalance_A(y) \rightarrow 2$
	$getBalance_A(x) \rightarrow 0$

Essa execução não corresponde a uma especificação sensata para o comportamento de contas bancárias: o cliente 2 deveria ter lido um saldo de \$1 para x , dado que ele lê o saldo de \$2 para y e o saldo de y foi atualizado após o de x . O comportamento anômalo no caso replicado não poderia ter ocorrido se as contas bancárias tivessem sido implementadas por um único servidor. Podemos construir sistemas que gerenciam objetos replicados sem o comportamento anômalo produzido pelo protocolo simples de nosso exemplo. Primeiramente, precisamos entender o que é considerado um comportamento correto para um sistema replicado.

Capacidade de linearização e consistência sequencial • Existem vários critérios de correção para objetos replicados. Os sistemas mais rigorosamente corretos podem ser *linearizados*, e essa propriedade é chamada de *capacidade de linearização*. Para entender a capacidade de linearização, considere uma implementação de serviço replicado com dois clientes. Seja $o_{i0}, o_{i1}, o_{i2}, \dots$ a sequência de operações de leitura e atualização executadas pelo cliente i em alguma execução. Cada operação o_{ij} é especificada pelo tipo de operação, pelos argumentos e pelos valores de retorno, conforme eles ocorreram em tempo de execução. Supomos que toda operação é síncrona. Isto é, os clientes esperam que uma operação termine antes de solicitar a próxima.

Um único servidor gerenciando uma única cópia dos objetos serializaria as operações dos clientes. No caso de uma execução apenas com cliente 1 e cliente 2, essa interposição das operações poderia ser, $o_{20}, o_{21}, o_{10}, o_{22}, o_{11}, o_{12}, \dots$. Definimos nossos critérios de correção para objetos replicados fazendo referência a uma interposição *virtual* das

operações dos clientes, as quais não ocorrem necessariamente em algum gerenciador de réplica em particular, mas que estabelecem a correção da execução.

Diz-se que um serviço de objetos compartilhados replicado pode ser linearizado se, para *qualquer execução*, existe uma interposição da série de operações executadas por todos os clientes que satisfaz os dois critérios a seguir:

- A sequência interposta de operações satisfaz a especificação de uma (única) cópia correta dos objetos.
- A ordem das operações na interposição é consistente com os tempos reais em que as operações ocorreram na execução real.

Essa definição capta a ideia de que para qualquer conjunto de operações de cliente existe uma execução canônica virtual – as operações interpostas a que a definição se refere – em preparação a uma imagem virtual única dos objetos compartilhados. E cada cliente enxerga um modo de visualização dos objetos compartilhados consistente com essa imagem única: isto é, os resultados das operações do cliente fazem sentido quando ocorrem dentro da interposição.

O serviço que deu origem à execução dos clientes de conta bancária do exemplo não pode ser linearizado. Mesmo ignorando o tempo real em que as operações ocorreram, não há nenhuma interposição das operações dos dois clientes que satisfaça qualquer especificação de conta bancária correta: para propósitos de auditoria, se a atualização de uma conta ocorresse após a outra, então a primeira deveria ser observada, caso a segunda também tivesse sido observada.

Note que a capacidade de linearização diz respeito à interposição de operações individuais e não se destina a ser transacional. Uma execução que pode ser linearizada pode violar as noções de consistência específicas da aplicação, caso não seja aplicado controle de concorrência.

O requisito de tempo real na capacidade de linearização é desejável em um mundo ideal, pois ele captura nossa noção de que os clientes devem receber informações atualizadas. Contudo, igualmente, a presença do tempo real na definição levanta o problema da equilíbrio da capacidade de linearização, pois nem sempre podemos sincronizar os relógios com o grau de precisão exigido. Uma condição de correção menos rigorosa é a *consistência sequencial*, que captura um requisito fundamental relacionado à ordem em que as requisições são processadas, sem apelar para o tempo real. A definição mantém o primeiro critério da definição de capacidade de linearização, mas modifica o segundo.

Diz-se que um serviço de objetos compartilhados replicado tem consistência sequencial se, para qualquer execução, existe alguma interposição da série de operações executadas por todos os clientes que satisfaça os dois critérios a seguir:

- A sequência interposta de operações satisfaz a especificação de uma (única) cópia correta dos objetos.
- A ordem das operações na interposição é consistente com a ordem do programa no qual cada cliente individual as executou.

Note que o tempo absoluto não aparece nessa definição. Nem nenhuma outra ordem *total* em todas as operações. A única noção de ordem relevante é a ordem dos eventos em cada cliente separado – a ordem do programa. A interposição de operações pode embaralhar, em qualquer ordem, a sequência de operações de um conjunto de clientes, desde que a ordem de cada cliente não seja violada e que o resultado de cada operação seja consistente, em termos da especificação dos objetos, com as operações que a precederam. Isso é semelhante a embaralhar vários maços de cartas para que elas se misturem de maneira tal que preservem a ordem original de cada maço.

Todo serviço que pode ser linearizado também tem consistência sequencial, pois a ordem do tempo real reflete a ordem do programa de cada cliente. O inverso não é verdadeiro. Um exemplo de execução de um serviço que tem consistência sequencial, mas não pode ser linearizado, aparece a seguir:

Cliente 1:	Cliente 2:
$setBalance_B(x, 1)$	
	$getBalance_A(y) \rightarrow 0$
	$getBalance_A(x) \rightarrow 0$
$setBalance_A(y, 2)$	

Essa execução é possível sob uma estratégia de replicação simples, mesmo que nenhum dos computadores A ou B falhe, mas se a atualização de x feita pelo cliente 1 em B não tiver chegado a A quando o cliente 2 a ler. O critério do tempo real para a capacidade de linearização não é satisfeito, pois $getBalance_A(x) \rightarrow 0$ ocorre depois de $setBalance_B(x, 1)$; mas a interposição a seguir satisfaz os dois critérios da consistência sequencial: $getBalance_A(y) \rightarrow 0$, $getBalance_A(x) \rightarrow 0$, $setBalance_B(x, 1)$, $setBalance_A(y, 2)$.

Lamport concebeu a consistência sequencial [1979] e a capacidade de linearização [1986] em relação aos registros de memória compartilhada (embora tenha usado o termo *atomicidade*, em vez de capacidade de linearização). Herlihy e Wing [1990] generalizaram a ideia para incluir objetos compartilhados arbitrários. Os sistemas de memória compartilhada distribuída também apresentam modelos de consistência menos rigorosos, conforme discutido no site que acompanha o livro [www.cdk5.net/dsm] (em inglês).

18.3.1 Replicação passiva (backup primário)

No modelo *passivo*, ou de *backup primário*, de replicação para tolerância a falhas (Figura 18.3), existe, em determinado momento, um único gerenciador de réplica (GR) primário e um ou mais gerenciadores de réplica secundários – *backups* ou *escravos*. Na forma pura do modelo, os *front-ends* (FE) se comunicam somente com o gerenciador de réplica primário para obterem o serviço. O gerenciador de réplica primário executa as operações e envia cópias dos dados atualizados para os *backups*. Se o primário falhar, um dos *backups* será promovido para atuar como primário.

A sequência de eventos, quando um cliente solicita que uma operação seja executada, é a seguinte:

1. *Requisição*: o *front-end* emite a requisição, contendo um identificador exclusivo, para o gerenciador de réplica primário.
2. *Coordenação*: o gerenciador primário trata cada requisição atomicamente, na ordem em que a recebe. Ele verifica o identificador exclusivo, para o caso de já ter executado e, se assim for, simplesmente envia a resposta novamente.
3. *Execução*: o gerenciador primário executa a requisição e armazena a resposta.
4. *Acordo*: se a requisição é uma atualização, o gerenciador primário envia o estado atualizado, a resposta e o identificador exclusivo para todos os gerenciadores de *backup*. Os gerenciadores de *backup* enviam uma confirmação.
5. *Resposta*: o gerenciador primário responde para o *front-end*, o qual envia a resposta de volta para o cliente.

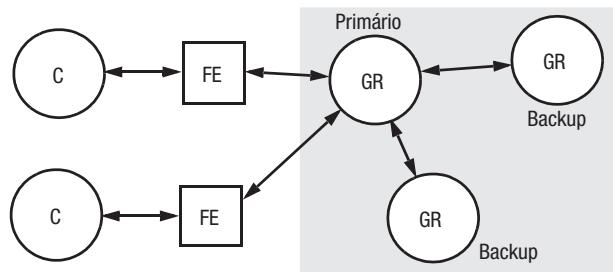


Figura 18.3 O modelo passivo (*backup* primário) de tolerância a falhas.

Obviamente, esse sistema implementa a capacidade de linearização se o gerenciador primário estiver correto, pois este coloca em sequência todas as operações sobre os objetos compartilhados. Se o gerenciador primário falhar, o sistema manterá a capacidade de linearização caso um único gerenciador de *backup* se torne o novo gerenciador primário e se a nova configuração do sistema partir exatamente de onde a última estava:

- o gerenciador primário é substituído por um único gerenciador de *backup* (se dois clientes começarem a usar dois gerenciadores de *backup*, o sistema funcionaria incorretamente); e
- os gerenciadores de réplica sobreviventes concordam a respeito das operações que tinham sido executadas no ponto em que o gerenciador primário substituto assume o controle.

Esses dois requisitos são satisfeitos se os gerenciadores de réplica (primário e *backups*) estão organizados como um grupo, e se o gerenciador primário usa comunicação em grupo com modo de visualização síncrono para enviar atualizações para os gerenciadores de *backup*. Então, o primeiro dos dois requisitos anteriores é facilmente satisfeito. Quando o gerenciador primário falha, o sistema de comunicação envia um novo modo de visualização para os gerenciadores de *backup* sobreviventes, excluindo o gerenciador primário antigo. O gerenciador de *backup* que substitui o gerenciador primário pode ser escolhido por qualquer função desse modo de visualização. Por exemplo, os gerenciadores de *backup* podem escolher o primeiro membro desse modo de visualização como substituto. Esse gerenciador de *backup* pode se registrar como gerenciador primário em um serviço de nomes que os clientes consultam quando suspeitam que o gerenciador primário falhou (ou assim que exigem o serviço).

O segundo requisito também é satisfeito pela propriedade de ordenação do sincronismo de modo de visualização e pelo uso de identificadores armazenados para detectar requisições repetidas. A semântica do modo de visualização síncrono garante que todos os gerenciadores de *backup*, ou nenhum deles, enviarão qualquer atualização dada, antes de enviar o novo modo de visualização. Assim, o novo gerenciador primário e os gerenciadores de *backup* sobreviventes concordam sobre se a atualização de qualquer cliente em particular foi processada ou não.

Considere um *front-end* que não recebeu uma resposta. O *front-end* retransmite a requisição para o gerenciador de *backup* que tiver assumido como gerenciador primário. O gerenciador primário pode ter falhado em qualquer ponto, durante a operação. Se ele falhou antes do estágio de acordo (4), os gerenciadores de réplica sobreviventes não podem ter processado a requisição. Se ele falhou durante o estágio de acordo, eles podem ter processado a requisição. Se ele falhou depois desse estágio, então eles definitivamente o processaram. Contudo, o novo gerenciador primário não precisa saber em que estágio o

gerenciador primário antigo estava quando falhou. Ao receber uma requisição, ele prossegue a partir do estágio 2 anterior. Pelo sincronismo do modo de visualização, nenhuma consulta com os gerenciadores de *backup* é necessária, pois todos processaram o mesmo conjunto de mensagens.

Discussão sobre a replicação passiva • O modelo *backup* primário pode ser usado mesmo onde o gerenciador de réplica primário se comporta de maneira não determinística por ser, por exemplo, *multithreaded*. Como o gerenciador primário comunica o estado atualizado das operações, em vez de uma especificação das operações em si, os gerenciadores de *backup* registram cegamente o estado determinado apenas pelas ações do gerenciador primário.

Para sobreviver a até f falhas de processo, um sistema de replicação passiva exige $f + 1$ gerenciadores de réplica (tal sistema não pode tolerar falhas bizantinas). O *front-end* exige pouca funcionalidade para obter tolerância a falhas. Ele precisará apenas pesquisar o novo gerenciador primário, quando o gerenciador primário corrente não responder.

A replicação passiva tem a desvantagem de acarretar sobrecargas relativamente grandes. A comunicação de modo de visualização síncrono exige várias rodadas de comunicação por *multicast* e, se o gerenciador primário falhar, ainda mais latência será acarretada, enquanto o sistema de comunicação em grupo concorda sobre o novo modo de visualização e o distribui.

Em uma variação do modelo apresentado aqui, os clientes podem enviar requisições de leitura para os gerenciadores de *backup*, diminuindo, assim, o trabalho do gerenciador primário. Com isso, a garantia da capacidade de linearização é perdida, mas os clientes recebem um serviço com consistência sequencial.

A replicação passiva é usada no sistema de arquivos replicado Harp [Liskov *et al.* 1991]. O *Network Information Service* da Sun (NIS, anteriormente *Yellow Pages*) usa replicação passiva para obter alta disponibilidade e bom desempenho, embora com menos garantias do que a consistência sequencial. As garantias de consistência mais fracas ainda são satisfatórias para muitos propósitos, como o armazenamento de certos tipos de registros de administração de sistema. Os dados replicados são atualizados em um servidor mestre e propagados de lá para servidores escravos, usando comunicação de um para um (em vez de grupo). Os clientes podem se comunicar com um servidor mestre, ou com um escravo, para recuperar informações. Entretanto, no NIS, os clientes não podem solicitar atualizações: elas são feitas nos arquivos do mestre.

18.3.2 Replicação ativa

No modelo *ativo* de replicação para tolerância a falhas (veja a Figura 18.4), os gerenciadores de réplica são máquinas de estado que desempenham papéis equivalentes e são organizados como um grupo. Os *front-ends* enviam suas requisições por *multicast* para o grupo de gerenciadores de réplica, e todos os gerenciadores de réplica processam a requisição independentemente, mas de forma idêntica, e respondem. Se qualquer gerenciador de réplica falhar, isso não tem nenhum impacto sobre o desempenho do serviço, pois os gerenciadores de réplica restantes continuam a responder normalmente. Veremos que a replicação ativa pode tolerar falhas bizantinas, pois o *front-end* pode reunir e comparar as respostas que recebe.

Sob a replicação ativa, a sequência de eventos quando um cliente solicita a execução de uma operação é a seguinte:

1. *Requisição*: o *front-end* anexa um identificador exclusivo à requisição e a envia por *multicast* para o grupo de gerenciadores de réplica, usando uma primitiva de *multicast* totalmente ordenada e confiável. Supõe-se que o *front-end* pode falhar por

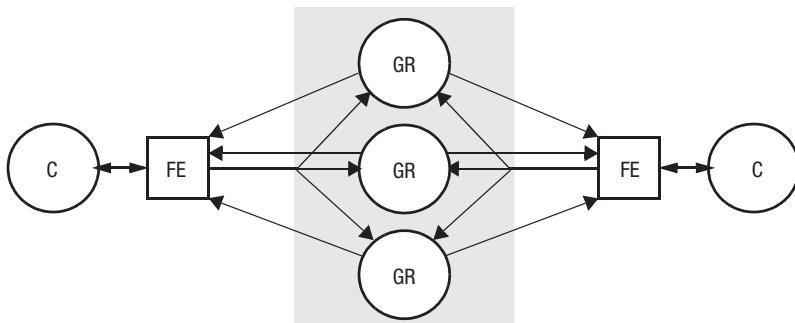


Figura 18.4 Replicação ativa.

colapso, na pior das hipóteses. Ele não emite a próxima requisição até que tenha recebido uma resposta.

2. *Coordenação*: o sistema de comunicação em grupo envia a requisição para cada gerenciador de réplica correto na mesma ordem (total).
3. *Execução*: todos os gerenciadores de réplica executam a requisição. Como eles são máquinas de estado, e como as requisições são distribuídas na mesma ordem total, todos os gerenciadores de réplica corretos processam a requisição de forma idêntica. A resposta contém o identificador de requisição exclusiva do cliente.
4. *Acordo*: nenhuma fase de acordo é necessária, devido à semântica da distribuição por *multicast*.
5. *Resposta*: cada gerenciador de réplica envia sua resposta para o *front-end*. O número de respostas reunidas pelo *front-end* depende das suposições de falha e do algoritmo de *multicast*. Se, por exemplo, o objetivo for tolerar apenas falhas por colapso e o *multicast* satisfizer o acordo uniforme e as propriedades de ordenação, o *front-end* passará ao cliente a primeira resposta que chegar e descartará as restantes (ele pode distingui-las das respostas de outras requisições examinando o identificador presente na resposta).

Esse sistema obtém consistência sequencial. Todos os gerenciadores de réplica corretos processam a mesma sequência de requisições. A confiabilidade do *multicast* garante que todos os gerenciadores de réplica corretos processam o mesmo conjunto de requisições e a ordem total garante que eles as processam na mesma ordem. Como são máquinas de estado, todos acabam no mesmo estado, após cada requisição. As requisições de cada *front-end* são atendidas na ordem FIFO (pois o *front-end* espera uma resposta antes de fazer a próxima requisição), que é igual à ordem do programa. Isso garante a consistência sequencial.

Se os clientes não se comunicam com outros clientes enquanto esperam pelas respostas de suas requisições, então suas requisições são processadas na ordem que acontece antes. Se os clientes são *multithreadeds*, e podem se comunicar uns com os outros enquanto esperam respostas do serviço, então, para garantir o processamento da requisição na ordem que acontece antes, teríamos que substituir o *multicast* por um que tivesse ordenação causal e total.

O sistema de replicação ativa não obtém capacidade de linearização. Isso porque a ordem total na qual os gerenciadores de réplica processam as requisições não é necessariamente igual à ordem de tempo real na qual os clientes as fizeram. Schneider [1990]

descreve como, em um sistema síncrono com relógios aproximadamente sincronizados, a ordem total na qual os gerenciadores de réplica processam requisições pode ser baseada na ordem dos carimbos de tempo físicos fornecidos pelos *front-ends* em suas requisições. Isso não garante a capacidade de linearização, pois os carimbos de tempo não são perfeitamente precisos; mas se aproxima disso.

Discussão sobre a replicação ativa • Admitimos uma solução para o *multicast* totalmente ordenado e confiável. Conforme o Capítulo 15, solucionar o *multicast* confiável e totalmente ordenado é equivalente a solucionar o consenso. Por sua vez, solucionar o consenso exige que o sistema seja síncrono, ou, em um sistema assíncrono, se faça uso de uma técnica como o emprego de detectores de falhas para contornar o resultado da impossibilidade de Fischer *et al.* [1985].

Algumas soluções para o consenso, como a de Canetti e Rabin [1993], funcionam mesmo com a suposição de falhas bizantinas. Dada tal solução e, portanto, uma solução para o *multicast* totalmente ordenado e confiável, o sistema de replicação ativa pode mascarar até f falhas bizantinas, desde que o serviço incorpore pelo menos $2f + 1$ gerenciadores de réplica. Cada *front-end* espera até ter reunido $f + 1$ respostas idênticas e passa essa resposta para o cliente. Ele descarta as outras respostas para a mesma requisição. Para estarmos rigorosamente seguros de qual resposta é realmente associada a qual requisição (dado o comportamento bizantino), exigimos que os gerenciadores de réplica acrescentem uma assinatura digital em suas respostas.

É possível flexibilizar o sistema que descrevemos. Primeiramente, admitimos que todas as atualizações nos objetos compartilhados replicados devem ocorrer na mesma ordem. Entretanto, na prática, algumas operações podem ser comutativas: isto é, o efeito de duas operações executadas na ordem $o_1; o_2$ é o mesmo da ordem inversa $o_2; o_1$. Por exemplo, quaisquer duas operações somente de leitura (de clientes diferentes) são comutativas; e quaisquer duas operações que não realizam leituras, mas atualizam objetos distintos, são comutativas. Um sistema de replicação ativa pode explorar o conhecimento da propriedade comutativa para evitar o custo de ordenar todas as requisições. Mencionamos, no Capítulo 15, que alguns têm proposto semânticas de ordenação por *multicast* específica ao aplicativo [Cheriton e Skeen 1993, Pedone e Schiper 1999].

Finalmente, os *front-ends* podem enviar requisições somente de leitura apenas para gerenciadores de réplica individuais. Ao fazerem isso, eles perdem a tolerância a falhas que acompanha o *multicast* das requisições, mas o serviço continua tendo consistência sequencial. Além disso, o *front-end* pode mascarar facilmente a falha de um gerenciador de réplica; nesse caso, simplesmente enviando a requisição somente de leitura para outro gerenciador de réplica.

18.4 Estudos de caso de serviços de alta disponibilidade: Gossip, Bayou e Coda

Nesta seção, consideraremos como se faz para aplicar técnicas de replicação para tornar os serviços de alta disponibilidade. Nossa ênfase, agora, é o fornecimento de acesso ao serviço para os clientes – com tempo de resposta razoável – pelo máximo de tempo possível, mesmo que alguns resultados não estejam de acordo com a consistência sequencial. Por exemplo, o usuário no trem, do início deste capítulo, pode querer aceitar as inconsistências temporárias entre cópias de dados, como as agendas, se puder continuar trabalhando enquanto estiver desconectado e corrigir os problemas posteriormente.

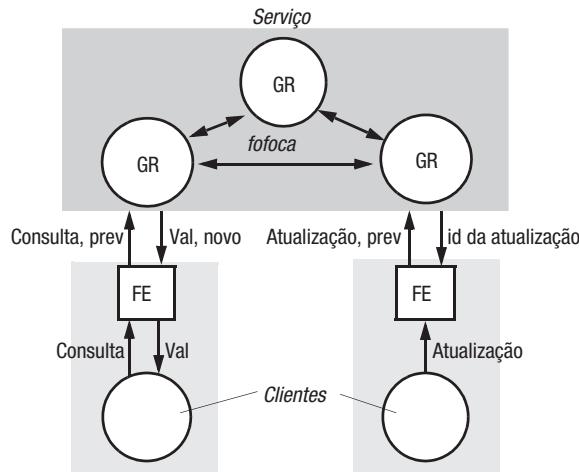


Figura 18.5 Operações de consulta e atualização em um serviço de fofoca.

Na Seção 18.3, vimos que os sistemas tolerantes a falhas transmitem as atualizações para os gerenciadores de réplica de maneira “ávida”: todos os gerenciadores de réplica corretos recebem as atualizações assim que possível e chegam a um acordo coletivo antes de devolverem o controle para o cliente. Esse comportamento é indesejável para a operação de alta disponibilidade. Em vez disso, o sistema deve fornecer um nível de serviço aceitável, usando um conjunto mínimo de gerenciadores de réplica conectados ao cliente. E ele deve minimizar o tempo durante o qual o cliente fica bloqueado, enquanto os gerenciadores de réplica coordenam suas atividades. Graus de consistência mais fracos geralmente exigem menos acordo e, portanto, permitem que dados compartilhados estejam disponíveis mais facilmente.

Examinaremos agora o projeto de três sistemas que fornecem serviços de alta disponibilidade: a arquitetura Gossip, Bayou e Coda.

18.4.1 A arquitetura Gossip

Ladin *et al.* [1992] desenvolveram o que chamamos de *arquitetura Gossip (fofoca)*, como uma estrutura para implementar serviços de alta disponibilidade por meio da replicação de dados próximo aos pontos onde os grupos de clientes precisam deles. O nome reflete o fato de que os gerenciadores de réplica trocam mensagens de “fofoca” periodicamente, para transmitir as atualizações que cada um recebeu dos clientes (veja a Figura 18.5). A arquitetura é baseada no trabalho anterior sobre bancos de dados feito por Fischer e Michael [1982] e por Wuu e Bernstein [1984]. Ela pode ser usada, por exemplo, para criar uma lista de discussão eletrônica ou um serviço de agenda de alta disponibilidade.

Um serviço de fofoca fornece dois tipos básicos de operação: as consultas são operações somente de leitura e as atualizações modificam, mas não leem o estado (esta última é uma definição mais restrita do que aquela que estivemos usando). Uma característica importante é que os *front-ends* enviam consultas e atualizações para o gerenciador de réplica que escolherem – um que esteja disponível e que possa fornecer um tempo de resposta razoável. O sistema faz duas garantias, mesmo que os gerenciadores de réplica possam estar temporariamente incapazes de se comunicar uns com os outros:

Cada cliente obtém um serviço consistente com o passar do tempo: na resposta a uma consulta, os gerenciadores de réplica só fornecem dados que refletem pelo me-

nos as atualizações observadas pelo cliente até o momento. Isso acontece mesmo que os clientes possam se comunicar com diferentes gerenciadores de réplica em diferentes momentos – e, portanto, em princípio, poderiam se comunicar com um gerenciador de réplica “menos avançado” do que o que usaram antes.

Consistência relaxada entre réplicas: todos os gerenciadores de réplica acabam por receber todas as atualizações e as aplicam com garantias de ordenação que tornam as réplicas suficientemente semelhantes para atenderem às necessidades do aplicativo. É importante perceber que, embora a arquitetura de fofoca possa ser usada para se obter consistência sequencial, ela se destina principalmente a apresentar garantias de consistência mais fracas. Dois clientes podem observar réplicas diferentes, mesmo que as réplicas incluam o mesmo conjunto de atualizações; e um cliente pode observar dados antigos.

Para suportar consistência relaxada, a arquitetura de fofoca suporta ordem de atualização causal, conforme definimos na Seção 15.2.1. Ela também suporta garantias de ordenação mais fortes, na forma de ordenação *forçada* (total e causal) e *imediata*. As atualizações com ordenação imediata são aplicadas em uma ordem consistente em relação a *qualquer* outra atualização, em todos os gerenciadores de réplica, seja a ordem destas atualizações especificadas como causal, forçada ou imediata. Além da ordenação forçada, é fornecida a ordenação imediata, pois uma atualização de ordenação forçada, ou uma atualização de ordem causal, que não possui uma relação *acontece antes* pode ser aplicada em diferentes ordens em diferentes gerenciadores de réplica.

A escolha da ordenação a ser usada é deixada para o projetista da aplicação e reflete um compromisso entre consistência e custos da operação. As atualizações causais são consideravelmente menos dispendiosas do que as outras e, quando possível, espera-se que sejam utilizadas. Note que as consultas que podem ser satisfeitas por um único gerenciador de réplica são sempre executadas na ordem causal com relação às outras operações.

Considere um aplicativo de lista de discussão eletrônica, no qual um programa cliente (que incorpora o *front-end*) é executado no computador do usuário e se comunica com um gerenciador de réplica local. O cliente envia as postagens do usuário para o gerenciador de réplica local e o gerenciador de réplica envia novas postagens em mensagens de fofoca (*gossip*) para outros gerenciadores de réplica. Os leitores veem as listas ligeiramente desatualizadas dos itens postados, mas isso normalmente não importa, caso o atraso seja na ordem de alguns minutos ou horas, em vez de dias. A ordenação causal poderia ser usada para postar itens. Isso significaria que, em geral, as postagens poderiam aparecer em ordens diferentes em diferentes gerenciadores de réplica, mas que, por exemplo, uma postagem cujo assunto fosse “Re: laranjas” sempre seria postada após a mensagem sobre “laranjas” à qual ela se refere. A ordenação forçada poderia ser usada para adicionar um novo assinante em uma lista de discussão, para que houvesse um registro inequívoco da ordem na qual os usuários entraram. A ordenação imediata poderia ser usada para remover um usuário da lista de assinaturas da lista de discussão, para que as mensagens não pudessem ser recuperadas por esse usuário por intermédio de algum gerenciador de réplica lento, uma vez que a operação de exclusão tivesse retornado.

O *front-end* de um serviço de fofoca manipula operações executadas pelo cliente usando uma API específica do aplicativo e as transforma em operações de fofoca. Em geral, as operações do cliente podem ler o estado replicado, modificá-lo ou ambos. Como na fofoca, as atualizações simplesmente modificam o estado, o *front-end* converte uma operação que lê e modifica o estado em uma consulta e em uma atualização separadas.

Em termos de nosso modelo de replicação básico, uma descrição, em linhas gerais, de como um serviço de fofoca processa operações de consulta e atualização é a seguinte:

1. *Requisição*: normalmente, o *front-end* envia as requisições apenas para um único gerenciador de réplica por vez. Entretanto, um *front-end* se comunicará com um gerenciador de réplica diferente quando aquele que utiliza normalmente falhar ou se tornar inacessível, e ele pode tentar usar um ou mais, caso o gerenciador normal esteja muito sobrecarregado. Os *front-ends* e, portanto, os clientes, podem ser bloqueados nas operações de consulta. Por outro lado, o funcionamento padrão das operações de atualização é retornar para o cliente assim que a operação tiver sido passada para o *front-end*; então, o *front-end* propaga a operação em segundo plano (*background*). Como alternativa, para obter maior confiabilidade, os clientes podem ser impedidos de continuar até que a atualização tenha sido enviada para $f+1$ gerenciadores de réplica e, portanto, serão enviadas para todas as partes, a despeito de até f falhas.
2. *Resposta de atualização*: se a requisição for uma atualização, o gerenciador de réplica responderá assim que a tiver recebido.
3. *Coordenação*: o gerenciador de réplica que recebe uma requisição não a processa até que possa aplicá-la, de acordo com as restrições de ordenação exigidas. Isso pode envolver a recepção de atualizações de outros gerenciadores de réplica em mensagens de fofoca. Nenhuma outra coordenação entre os gerenciadores de réplica está envolvida.
4. *Execução*: o gerenciador de réplica executa a requisição.
5. *Resposta de consulta*: se a requisição for uma consulta, o gerenciador de réplica responderá nesse ponto.
6. *Acordo*: os gerenciadores de réplica atualizam-se uns aos outros, trocando *mensagens de fofoca*, as quais contêm as atualizações mais recentes que receberam. Diz-se que eles atualizam-se uns aos outros de maneira “preguiçosa”, no sentido de que essas mensagens de fofoca só podem ser trocadas ocasionalmente, após várias atualizações terem sido reunidas, ou quando um gerenciador de réplica descobre que está faltando uma atualização enviada para um de seus pares, que ele precisa para processar uma requisição.

Descreveremos agora o sistema de fofoca com mais detalhes. Começaremos considerando os carimbos de tempo e as estruturas de dados mantidas pelos *front-ends* e pelos gerenciadores de réplica para conservar as garantias da ordem das atualizações. Em seguida, explicaremos como os gerenciadores de réplica processam consultas e atualizações. Grande parte do processamento dos carimbos de tempo vetoriais necessários para manter atualizações causais é semelhante ao algoritmo de *multicast causal* da Seção 15.4.3.

O carimbo de tempo (timestamp) de versão do front-end • Para controlar a ordenação do processamento de operação, cada *front-end* mantém um carimbo de tempo vetorial (*vector timestamp*) que reflete a versão mais recente dos valores de dados acessados pelo *front-end* (e, portanto, acessados pelo cliente). Esse carimbo de tempo, denotado como *prev* na Figura 18.5, contém uma entrada para cada gerenciador de réplica. O *front-end* o envia em cada mensagem de requisição para um gerenciador de réplica, junto a uma descrição da operação de consulta ou atualização em si. Quando um gerenciador de réplica retorna um valor como resultado de uma operação de consulta, ele fornece um novo carimbo de

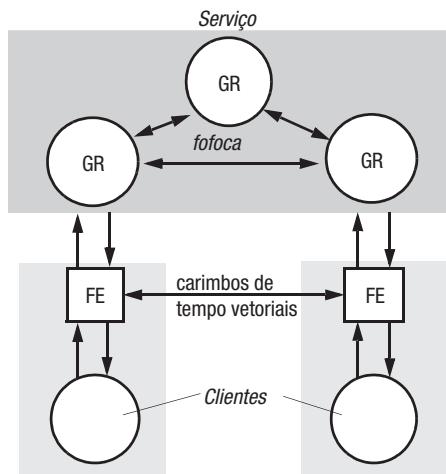


Figura 18.6 Os *front-ends* propagam seus carimbos de tempo quando os clientes se comunicam diretamente.

tempo vetorial (*novo*, na Figura 18.5), pois as réplicas podem ter sido atualizadas desde a última operação. Analogamente, uma operação de atualização retorna um carimbo de tempo vetorial (*id da atualização*, na Figura 18.5) que é exclusivo da atualização. Cada carimbo de tempo retornado é mesclado com o carimbo de tempo anterior do *front-end*, para registrar a versão dos dados duplicados que foram observados pelo cliente. (Veja na Seção 14.4 uma definição de integração de carimbos de tempo vetorial.)

Os clientes trocam dados acessando o mesmo serviço de fofoca e comunicando-se diretamente uns com os outros. Como a comunicação de cliente para cliente também pode levar a relacionamentos causais entre as operações aplicadas ao serviço, ela também ocorre por intermédio dos *front-ends* dos clientes. Desse modo, os *front-ends* podem levar seus carimbos de tempo vetoriais “de carona” nas mensagens para outros clientes. Os destinatários as integram com seus próprios carimbos de tempo, para que os relacionamentos causais possam ser deduzidos corretamente. A situação é mostrada na Figura 18.6.

Estado do gerenciador de réplica • Independentemente da aplicação, um gerenciador de réplica contém os seguintes componentes principais de estado (Figura 18.7):

Valor: é o valor do estado da aplicação, mantido pelo gerenciador de réplica. Cada gerenciador de réplica é uma máquina de estado, a qual começa com um valor inicial especificado e que, daí em diante, é apenas o resultado da aplicação de operações de atualização nesse estado.

Carimbo de tempo do valor: é o carimbo de tempo vetorial que representa as atualizações refletidas no valor. Ele contém uma entrada para cada gerenciador de réplica. É atualizado quando uma operação de atualização é aplicada no valor.

Log de atualização: todas as operações de atualização são registradas nesse *log*, assim que são recebidas. Um gerenciador de réplica mantém as atualizações em um *log* por dois motivos. O primeiro é que o gerenciador de réplica ainda não pode aplicar a atualização, porque ela ainda não é *estável*. Uma atualização estável é aquela que pode ser aplicada consistentemente com suas garantias de ordenação (causal, forçada ou imediata). Uma atualização que ainda não é estável deve ser

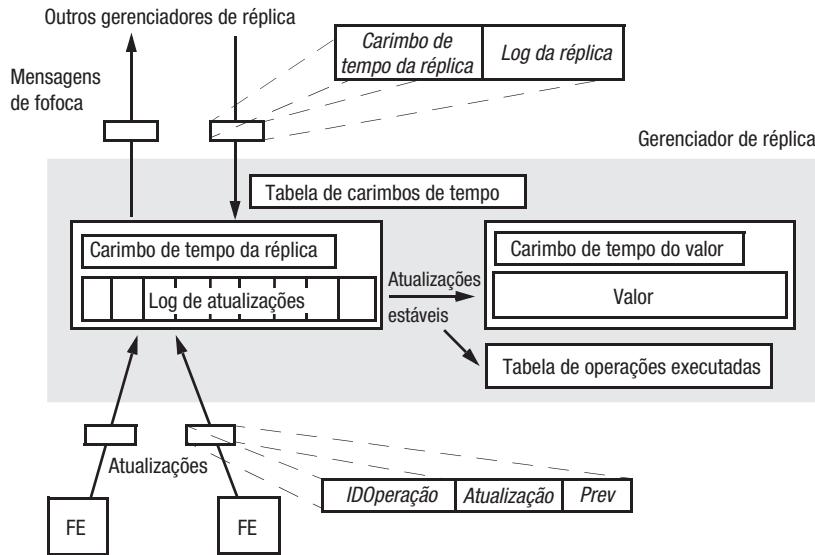


Figura 18.7 Um gerenciador de réplica fofoca mostrando seus principais componentes de estado.

retida e ainda não processada. O segundo motivo para se manter uma atualização no *log* é que, mesmo que a atualização tenha se tornado estável e aplicada ao valor, o gerenciador de réplica não recebeu confirmação de que essa atualização foi recebida em todos os outros gerenciadores de réplica. Nesse meio-tempo, ele propaga a atualização em mensagens de fofoca.

Carimbo de tempo da réplica: este carimbo de tempo vetorial representa as atualizações que foram aceitas pelo gerenciador de réplica – isto é, colocadas no *log* do gerenciador. Ele difere do valor carimbo de tempo em geral, é claro, pois nem todas as atualizações do *log* são estáveis.

Tabela de operações executadas: a mesma atualização pode chegar de um *front-end* a determinado gerenciador de réplica e em mensagens de fofoca de outros gerenciadores de réplica. Para impedir que uma atualização seja aplicada duas vezes, é mantida a tabela de operações executadas, contendo os identificadores únicos, fornecidos pelo *front-end*, das atualizações que foram aplicadas ao valor. Os gerenciadores de réplica consultam essa tabela antes de adicionar uma atualização no *log*.

Tabela de carimbos de tempo: essa tabela contém um carimbo de tempo vetorial de todos os outros gerenciadores de réplica, preenchido com carimbos de tempo que chegam deles em mensagens de fofoca. Os gerenciadores de réplica usam a tabela para estabelecer quando uma atualização foi aplicada em todos os gerenciadores de réplica.

Os gerenciadores de réplica são numerados como 0, 1, 2,..., e o i -ésimo elemento de um carimbo de tempo vetorial mantido pelo gerenciador de réplica i corresponde ao número de atualizações recebidas dos *front-ends* por i ; e o j -ésimo componente ($j \neq i$) é igual ao número de atualizações recebidas por j e propagadas para i em mensagens de fofoca. Assim, por exemplo, em um sistema de fofoca de três gerenciadores, um carimbo de tempo

de valor igual a (2,4,5) no gerenciador 0 representaria o fato de que o valor reflete as duas primeiras atualizações aceitas dos *front-ends* no gerenciador 0, as quatro primeiras no gerenciador 1 e as cinco primeiras no gerenciador 2. A seguir, veremos com mais detalhes como os carimbos de tempo são usados para impor a ordem.

Operações de consulta • A operação mais simples a considerar é a de consulta. Lembre-se de que uma requisição de consulta q contém uma descrição da operação e um carimbo de tempo $q.prev$ enviado pelo *front-end*. Esta última reflete a versão mais recente do valor lido pelo *front-end* ou enviado como atualização. Portanto, a tarefa do gerenciador de réplica é retornar um valor que seja pelo menos tão recente quanto esse. Se $valueTS$ for o valor do carimbo de tempo da réplica, então q pode ser aplicado ao valor da réplica se:

$$q.prev \leq valueTS$$

O gerenciador de réplica mantém q em uma lista de operações de consulta pendentes (isto é, uma fila de retenção) até que essa condição seja satisfeita. Ele pode esperar pelas atualizações faltantes, as quais devem chegar em mensagens de fofoca, ou pode solicitar as atualizações dos gerenciadores de réplica envolvidos. Por exemplo, se $valueTS$ for (2,5,5) e $q.prev$ for (2,4,6), pode ser visto que apenas uma atualização está faltando – do gerenciador de réplica 2. (O *front-end* que enviou q deve ter consultado um gerenciador de réplica diferente anteriormente, para ter visto essa atualização que o gerenciador de réplica não viu.)

Quando a consulta puder ser aplicada, o gerenciador de réplica retornará $valueTS$ para o *front-end*, como o carimbo de tempo *novo*, mostrado na Figura 18.5. Então, o *front-end* integra isso com seu carimbo de tempo: $frontEndTS := merge(frontEndTS, novo)$. A atualização no gerenciador de réplica 1, que o *front-end* não viu antes da consulta no exemplo que acabamos de dar ($q.prev$ tem 4 onde o gerenciador de réplica tem 5), será refletida na atualização de $frontEndTS$ (e, possivelmente, no valor retornado, dependendo da consulta).

Processamento de operações de atualização na ordem causal • Um *front-end* envia uma requisição de atualização para um ou mais gerenciadores de réplica. Cada requisição de atualização u contém uma especificação da atualização (seu tipo e seus parâmetros) $u.op$, o carimbo de tempo do *front-end* $u.prev$ e um identificador exclusivo gerado pelo *front-end*, $u.id$. Se o *front-end* enviar a mesma requisição u para vários gerenciadores de réplica, ele sempre usará o mesmo identificador em u – de modo que ele não será processado como várias requisições, porém como requisições idênticas.

Quando o gerenciador de réplica i recebe uma requisição de atualização de um *front-end*, ele verifica se já não processou essa requisição examinando seu identificador de operação na tabela de operações executadas e nos registros de seu *log*. O gerenciador de réplica descartará a atualização se já a tiver visto; caso contrário, ele incrementará por uma unidade o i -ésimo elemento em seu carimbo de tempo de réplica, para manter a contagem do número de atualizações que recebeu diretamente dos *front-ends*. Então, o gerenciador de réplica atribui à requisição de atualização u um carimbo de tempo vetorial exclusivo, cuja origem será dada em breve, e um registro da atualização é colocado no *log* do gerenciador de réplica. Se ts for o carimbo de tempo exclusivo atribuído à atualização pelo gerenciador de réplica, então o registro de atualização será construído e armazenado no *log* como a seguinte tupla:

$$logRecord := \langle i, ts, u.op, u.prev, u.id \rangle$$

O gerenciador de réplica i deriva o carimbo de tempo ts de $u.prev$ substituindo o i -ésimo elemento de $u.prev$ pelo i -ésimo elemento de seu carimbo de tempo de réplica (que ele acabou de incrementar). Essa ação torna ts única, garantindo assim que todos os componentes do sistema registrem corretamente se observaram a atualização ou não. Os

elementos restantes em ts são copiados de $u.prev$, pois esses valores são enviados pelo *front-end* que devem ser usados para determinar quando a atualização está estável. Então, o gerenciador de réplica devolve ts imediatamente para o *front-end*, que o integra com seu carimbo de tempo existente. Note que um *front-end* pode enviar sua atualização para vários gerenciadores de réplica e receber diferentes carimbos de tempo em retorno, todos os quais precisarão ser integrados em seu carimbo de tempo.

A condição de estabilidade para uma atualização u é semelhante à das consultas:

$$u.prev \leq valueTS$$

Essa condição diz que todas as atualizações das quais essa atualização depende – isto é, todas as atualizações que foram observadas pelo *front-end* que executou a atualização – já foram aplicadas ao valor. Se essa condição não for satisfeita no momento em que a atualização for enviada, ela será verificada novamente, quando as mensagens de fofoca chegarem. Quando a condição de estabilidade tiver sido satisfeita para uma atualização r , o gerenciador de réplica aplicará a atualização no valor, acertará o carimbo de tempo do valor e a tabela de operações executadas *executed*:

```
valor := apply(valor, r.u.op)
valueTS := merge(valueTS, r.ts)
executed := executed ∪ {r.u.id}
```

A primeira dessas três declarações representa a aplicação da atualização no valor. Na segunda declaração, o carimbo de tempo da atualização é integrado com o do valor. Na terceira, o identificador de operação da atualização é adicionado ao conjunto de identificadores das operações que foram executadas – o qual é usado para verificar a existência de requisições de operação repetidas.

Operações de atualização forçadas e imediatas • As atualizações forçadas e imediatas exigem tratamento especial. Lembre-se de que as atualizações forçadas são ordenadas totalmente, assim como de forma causal. O método básico para ordenar atualizações forçadas é anexar um número de sequência exclusivo nos carimbos de tempo associados a elas e processá-los na ordem desse número de sequência. Conforme o Capítulo 15, um método geral para gerar números de sequência é usar um processo sequenciador exclusivo. No entanto, a confiança em um processo exclusivo é inadequada no contexto de um serviço de alta disponibilidade. A solução é designar um *gerenciador de réplica primário* como sequenciador, em dado momento, mas garantir que outro gerenciador de réplica possa ser eleito para assumir consistentemente como sequenciador, caso o gerenciador primário venha a falhar. O que é exigido é que uma maioria de gerenciadores de réplica (incluindo o primário) registre qual atualização é a próxima na sequência, antes que a operação possa ser aplicada. Então, desde que uma maioria dos gerenciadores de réplica sobreviva à falha, essa decisão sobre a ordenação será honrada por um novo gerenciador primário eleito dentre os gerenciadores de réplica sobreviventes.

As atualizações imediatas são ordenadas com relação às atualizações forçadas, usando o gerenciador de réplica primário para ordená-las nessa sequência. O gerenciador primário também determina quais atualizações causais são consideradas como precedentes a uma atualização imediata. Ele faz isso se comunicando e sincronizando-se com os outros gerenciadores de réplica em ordem, para chegar a um acordo sobre isso. Mais detalhes são fornecidos em Ladin *et al.* [1992].

Mensagens de fofoca • Os gerenciadores de réplica enviam mensagens de fofoca (*gossip*) contendo informações a respeito de uma ou mais atualizações, para que os outros gerenciadores de réplica possam atualizar seus estados. Um gerenciador de réplica usa as en-

tradas de sua tabela de carimbo de tempo para fazer uma estimativa de quais atualizações qualquer outro gerenciador de réplica ainda não recebeu (trata-se de uma estimativa, pois agora esse gerenciador de réplica pode ter recebido mais atualizações).

Uma mensagem de fofoca m consiste em dois itens enviados pelo gerenciador de réplica de origem: seu log $m.log$ e seu carimbo de tempo de réplica $m.ts$ (veja a Figura 18.7). O gerenciador de réplica que recebe uma mensagem de fofoca tem três tarefas principais:

- Integrar o log recebido com o seu próprio (ele pode conter atualizações não vistas anteriormente pelo receptor).
- Aplicar as atualizações que se tornaram estáveis e não foram executadas antes (por sua vez, as atualizações estáveis presentes no log recebido podem tornar estáveis as atualizações pendentes).
- Eliminar registros do log e entradas na tabela de operações executadas, quando for conhecido que as atualizações foram aplicadas por toda parte e para as quais não há perigo de repetições. Eliminar as entradas redundantes do log e da tabela de operações executadas é uma tarefa importante, pois de outro modo elas cresceriam sem limite.

É simples integrar o log contido em uma mensagem de fofoca recebida com o log do receptor. Digamos que $replicaTS$ é o carimbo de tempo de réplica do destinatário. Um registro r em $m.log$ é adicionado no log do receptor, a não ser que $r.ts \leq replicaTS$ – no caso em que ele já está no log ou foi aplicado ao valor e depois descartado.

O gerenciador de réplica integra o carimbo de tempo da mensagem de fofoca recebida com seu próprio carimbo de tempo de réplica $replicaTS$, para que ela corresponda às adições feitas no log :

$$replicaTS := merge(replicaTS, m.ts)$$

Quando novos registros de atualização tiverem sido integrados no log , o gerenciador de réplica reunirá o conjunto S de todas as atualizações no log , que agora são estáveis. Elas podem ser aplicadas ao valor, mas é preciso cuidado com a ordem com que elas são aplicadas, para que a relação acontece antes seja observada. O gerenciador de réplica ordena as atualizações no conjunto, de acordo com a ordem parcial \leq entre os carimbos de tempo vetoriais. Então, ele aplica as atualizações nessa ordem, com a menor vindo primeiro. Isto é, cada $r \in S$ é aplicado apenas quando não existe nenhum $s \in S$ tal que $s.prev < r.prev$.

Então, o gerenciador de réplica procura registros no log que possam ser descartados. Se a mensagem de fofoca foi enviada pelo gerenciador de réplica j e se $tableTS$ é a tabela de carimbos de tempo de réplica dos gerenciadores de réplica, então o gerenciador de réplica configura

$$tableTS[j] := m.ts$$

Agora, o gerenciador de réplica pode descartar qualquer registro r do log para uma atualização que tenha sido recebida em toda parte. Isto é, se c é o gerenciador de réplica que criou o registro, então exigimos que todos os gerenciadores de réplica i :

$$tableTS[i][c] \geq r.ts[c]$$

A arquitetura de fofoca também define como os gerenciadores de réplica podem descartar entradas da tabela de operações executadas. É importante não descartar essas entradas cedo demais; caso contrário, uma operação muito atrasada poderia ser aplicada duas vezes por engano. Ladin *et al.* [1992] fornecem detalhes do esquema. Basicamente, os *front-ends* emitem sinais de reconhecimento nas respostas de suas atualizações, de modo que os gerenciadores de réplica sabem quando um *front-end* parará de enviar a atualização. Eles presumem um atraso de propagação de atualização máximo a partir desse ponto.

Propagação de atualização • A arquitetura de fofoca não especifica quando os gerenciadores de réplica trocam mensagens de fofoca, nem como um gerenciador de réplica escolhe outros gerenciadores de réplica para enviar uma fofoca. É necessária uma estratégia de propagação de atualização robusta, caso todos os gerenciadores de réplica devam receber todas as atualizações em um tempo aceitável.

O tempo que leva para todos os gerenciadores de réplica receberem determinada atualização depende de três fatores:

- A frequência e duração dos particionamentos da rede.
- A frequência com que os gerenciadores de réplica enviam mensagens de fofoca.
- A política de escolha de um parceiro para trocar fofoca.

O primeiro fator está fora do controle do sistema, embora os usuários possam determinar, até certo ponto, com que frequência eles trabalham desconectados.

A frequência de troca de fofoca desejada pode ser ajustada na própria aplicação. Considere um sistema de lista de discussões eletrônica compartilhada entre vários *sites*. Parece desnecessário que cada item seja enviado imediatamente para todos os *sites*. No entanto, e se a fofoca for trocada apenas após longos períodos de tempo, digamos, uma vez por dia? Se forem usadas apenas atualizações causais, então será bem possível que os clientes em cada *site* tenham seus próprios debates consistentes pela mesma lista de discussão, isolados das discussões que ocorrem em outros *sites*. Então, digamos, à meia-noite, todos os debates serão integrados; mas os debates sobre o mesmo assunto provavelmente serão incongruentes, quando teria sido preferível que eles levassem em conta uns aos outros. Um período de troca de fofoca de alguns minutos, ou mesmo horas, parece mais apropriado nesse caso.

Existem vários tipos de política de escolha de parceiro. Golding e Long [1993] consideram as políticas *aleatória*, *determinista* e *topológica* para seu protocolo antientropia com carimbo de tempo, o qual usa um esquema de propagação de atualização estilo fofoca.

As políticas aleatórias escolhem um parceiro ao acaso, mas com probabilidades ponderadas, de modo a favorecer alguns parceiros em detrimento de outros como, por exemplo, escolher os parceiros próximos. Golding e Long descobriram que tal política funciona surpreendentemente bem sob simulações. As políticas deterministas utilizam uma função simples do estado do gerenciador de réplica para fazer a escolha do parceiro. Por exemplo, um gerenciador de réplica poderia examinar sua tabela de carimbos de tempo e escolher o gerenciador de réplica que pareça estar mais atrasado nas atualizações recebidas.

As políticas topológicas organizam os gerenciadores de réplica em um grafo fixo. Uma possibilidade é uma malha: por exemplo, os gerenciadores de réplica enviam mensagens de fofoca para os quatro gerenciadores de réplica com que está diretamente conectado. Outra é organizar os gerenciadores de réplica em um anel, com cada um passando a fofoca apenas para seu vizinho (digamos, no sentido horário), de modo que as atualizações de qualquer gerenciador de réplica percorram o anel todo. Existem muitas outras topologias possíveis, incluindo as árvores.

Diferentes políticas de escolha de parceiro como essas ponderam a quantidade de comunicação em relação às latências de transmissão mais altas e a possibilidade de que uma única falha afete outros gerenciadores de réplica. Na prática, a escolha depende da importância relativa desses fatores. Por exemplo, a topologia em anel produz relativamente pouca comunicação, mas está sujeita às altas latências de transmissão, pois a fofoca geralmente precisa passar por vários gerenciadores de réplica. Além disso, se um gerenciador de réplica falhar, o anel não funcionará e precisará ser reconfigurado. Em contraste, a política de escolha aleatória não é suscetível às falhas, mas pode produzir tempos de propagação de atualização mais variáveis.

Discussão sobre a arquitetura de fofoca • A arquitetura de fofoca tem como objetivo obter alta disponibilidade para os serviços. A seu favor está o fato de que os clientes podem continuar a obter um serviço mesmo quando são separados do resto da rede, desde que pelo menos um gerenciador de réplica continue a funcionar na partição. Contudo, esse tipo de disponibilidade é obtido à custa da imposição de garantias de consistência relaxadas. Para objetos como contas bancárias, em que a consistência sequencial é obrigatória, uma arquitetura de fofoca não é melhor do que os sistemas tolerantes a falhas estudados na Seção 18.3 e fornece o serviço apenas em uma partição em que haja maioria.

A estratégia “preguiçosa” de propagação de atualização torna um sistema baseado em fofoca inadequado para atualizar réplicas em um tempo próximo ao real, como quando os usuários tomam parte de uma conferência em tempo real e atualizam um documento compartilhado. Um sistema baseado em *multicast* seria mais apropriado para esse caso.

A escalabilidade de um sistema de fofoca é outro problema. À medida que o número de gerenciadores de réplica aumenta, também aumenta o número de mensagens de fofoca que precisam ser transmitidas e o tamanho dos carimbos de tempo usados. Se um cliente faz uma consulta, normalmente isso exige duas mensagens (entre o *front-end* e o gerenciador de réplica). Se um cliente faz uma operação de atualização causal, e se cada um dos R gerenciadores de réplica normalmente reúne G atualizações em uma mensagem de fofoca, então o número de mensagens trocadas é de $2 + (R - 1)/G$. O primeiro termo representa a comunicação entre o *front-end* e o gerenciador de réplica e o segundo é a parte da atualização de uma mensagem de fofoca enviada para os outros gerenciadores de réplica. Aumentar G melhora o número de mensagens, mas piora as latências de envio, pois o gerenciador de réplica espera que mais atualizações cheguem antes de propagá-las.

Uma estratégia para fazer com que os serviços baseados em fofoca possam mudar de escala é tornar a maioria das réplicas somente de leitura. Em outras palavras, essas réplicas são atualizadas pelas mensagens de fofoca, mas não recebem atualizações diretamente dos *front-ends*. Esse arranjo é potencialmente útil onde a relação *atualização/consulta* é pequena. As réplicas somente de leitura podem estar situadas próximas aos grupos de clientes e as atualizações podem ser atendidas por relativamente poucos gerenciadores de réplica centralizados. O tráfego de fofoca é reduzido, pois as réplicas somente de leitura não têm nenhuma fofoca para propagar. E os carimbos de tempo vetoriais só precisam conter entradas para as réplicas que podem ser atualizadas.

18.4.2 Bayou e a estratégia da transformação operacional

O sistema Bayou [Terry *et al.* 1995, Petersen *et al.* 1997] fornece replicação de dados de alta disponibilidade com garantias menores do que a consistência sequencial, como a arquitetura de fofoca e o protocolo antientropia com carimbo de tempo. Assim como nesses sistemas, os gerenciadores de réplica Bayou suportam conectividade variável, trocando atualizações em pares, nas quais os projetistas também designam um protocolo antientropia. Contudo, o Bayou adota uma estratégia marcadamente diferente, pois permite a ocorrência de detecção de conflitos específica do domínio e solução de conflitos.

Considere o usuário que precisa atualizar uma agenda enquanto trabalha de forma desconectada. Se for exigida consistência restrita, então na arquitetura de fofoca as atualizações seriam realizadas usando-se uma operação forçada (totalmente ordenada). No entanto, então, apenas os usuários de uma divisão onde houvesse maioria poderia atualizar a agenda. Assim, o acesso dos usuários à agenda pode ser limitado – independentemente de precisarem, de fato, fazer atualizações que violariam a integridade da agenda. Os usuários que quiserem marcar um compromisso não conflitante serão tratados de forma igual aos usuários que podem ter marcado uma repartição de hora duas vezes, inconscientemente.

No Bayou, em contraste, os usuários que estão no trem e no trabalho podem fazer as atualizações que desejarem. Todas elas serão aplicadas e registradas no gerenciador de réplica que encontrarem. Entretanto, quando as atualizações recebidas em quaisquer dois gerenciadores de réplica são integradas, durante uma troca antientropia, os gerenciadores de réplica detectam e solucionam os conflitos. Qualquer critério de conflito específico do domínio entre operações pode ser aplicado. Por exemplo, se um executivo e sua secretária tiverem adicionado compromissos na mesma repartição de hora, um sistema Bayou detectará isso, depois que o executivo tiver reconectado seu *notebook*. Além disso, ele soluciona o conflito de acordo com uma política específica do domínio. Nesse caso, ele poderia, por exemplo, confirmar o compromisso do executivo e remover o da secretaria na repartição de hora. Tal efeito, no qual uma ou mais operações de um conjunto conflitante são desfeitas ou alteradas para solucioná-las é chamado de *transformação operacional*.

O estado que o Bayou replica é mantido na forma de um banco de dados, suportando consultas e atualizações (que podem inserir, modificar ou excluir itens no banco de dados). Embora não nos concentremos nesse aspecto aqui, uma atualização do Bayou é um caso especial de transação. Ela consiste em uma única operação, uma invocação de um procedimento, o qual afeta vários objetos dentro de cada gerenciador de réplica, mas que é executada com as garantias ACID. O Bayou pode desfazer e refazer as atualizações no banco de dados, à medida que a execução prossegue.

A garantia do Bayou é que, finalmente, cada gerenciador de réplica receberá o mesmo conjunto de atualizações e aplicará essas atualizações de maneira tal que os bancos de dados dos gerenciadores de réplica sejam idênticos. Na prática, pode haver um fluxo contínuo de atualizações, e os bancos de dados podem nunca se tornar idênticos; mas eles se tornariam idênticos se as atualizações cessassem.

Atualizações confirmadas e de tentativa • As atualizações são marcadas como *de tentativa* quando são aplicadas a um banco de dados pela primeira vez. O Bayou faz com que as atualizações de tentativa sejam colocadas em uma ordem canônica e marcadas como *confirmadas*. Enquanto as atualizações são de tentativa, o sistema pode desfazê-las e reaplicá-las, à medida que produz um estado consistente. Uma vez confirmadas, elas permanecem aplicadas em sua ordem definida. Na prática, a ordem confirmada pode ser obtida designando-se algum gerenciador de réplica como gerenciador de réplica *primário*. Como sempre, ele decide a ordem confirmada como aquela na qual recebe as atualizações de tentativa e propaga essa informação de ordenação para os outros gerenciadores de réplica. Como gerenciador primário, os usuários podem escolher, por exemplo, uma máquina rápida que normalmente esteja disponível; igualmente, ele poderia ser o gerenciador de réplica no *notebook* do executivo, caso as atualizações desse usuário tivessem prioridade.

A qualquer momento, o estado de uma réplica de banco de dados é derivado de uma sequência (possivelmente vazia) de atualizações confirmadas, seguida de uma sequência (possivelmente vazia) de atualizações de tentativa. Se a próxima atualização confirmada chegar, ou se uma das atualizações de tentativa aplicada se tornar a próxima atualização confirmada, deverá ocorrer, então, uma reordenação das atualizações. Na Figura 18.8, t_i se tornou efetivada. Todas as atualizações de tentativa após c_N precisam ser desfeitas; então, t_i é aplicada após c_N e t_0 a t_{i-1} e t_{i+1} etc., são reaplicadas após t_i .

Verificações de dependência e procedimentos de integração • Uma atualização pode entrar em conflito com alguma outra operação que já tenha sido aplicada. Devido a essa possibilidade, toda atualização do Bayou contém uma *verificação de dependência* e um *procedimento de integração*, além da especificação da operação (o tipo e os parâmetros da operação). Todos esses componentes de uma atualização são específicos do domínio.

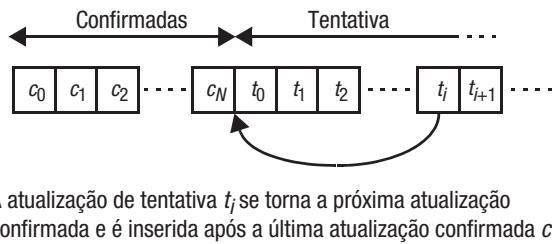


Figura 18.8 Atualizações confirmadas e de tentativa no Bayou.

Um gerenciador de réplica ativa o procedimento de verificação de dependência antes de aplicar a operação. Ele verifica se ocorreria um conflito caso a atualização fosse aplicada e, para fazer isso, pode examinar qualquer parte do banco de dados. Por exemplo, considere o caso do registro de um compromisso em uma agenda. A verificação de dependência poderia, mais simplesmente, testar a existência de um conflito de *escrita-escrita*: isto é, se outro cliente preencheu a repartição de hora exigida. No entanto, a verificação de dependência também poderia testar a existência de um conflito de *leitura-escrita*. Por exemplo, ela poderia testar se a repartição de hora desejada está vazia e se o número de compromissos nesse dia é menor do que seis.

Se a verificação de dependência indicar um conflito, então o Bayou ativará o procedimento de integração da operação. Esse procedimento altera a operação que será aplicada, de modo que ela obtenha algo semelhante ao efeito pretendido, mas evite um conflito. Por exemplo, no caso da agenda, o procedimento de integração poderia escolher outra repartição de hora próxima ou, conforme mencionado anteriormente, poderia usar um esquema de prioridade simples para decidir qual compromisso é mais importante e impô-lo. O procedimento de integração pode não encontrar uma alteração conveniente para a operação, no caso em que o sistema indicará um erro. Entretanto, o efeito de um procedimento de integração é determinista – os gerenciadores de réplica Bayou são máquinas de estado.

Discussão • O Bayou difere dos outros esquemas de replicação que consideramos, pois torna a replicação não transparente para a aplicação. Ele explora o conhecimento da semântica da aplicação para aumentar a disponibilidade dos dados, enquanto mantém um estado replicado, que é o que poderíamos chamar de *consistência sequencial final*.

As desvantagens dessa estratégia são, primeiramente, a maior complexidade para o programador da aplicação que precisa fornecer verificações de dependência e procedimentos de integração. Ambos podem ser complexos de produzir, dado o grande número de possíveis conflitos que precisam ser detectados e solucionados. A segunda desvantagem é a maior complexidade para o usuário. Não apenas se espera que os usuários lidem com dados que são lidos enquanto ainda são de tentativa, como também com o fato de que a operação especificada por um usuário pode ser alterada. Por exemplo, o usuário marcou uma repartição de hora em uma agenda, apenas para descobrir posteriormente que a marcação “pulou” para uma repartição de hora próxima. É muito importante que o usuário receba uma indicação clara de quais dados são de tentativa e quais são efetivados.

A estratégia da transformação operacional usada pelo Bayou aparece particularmente nos sistemas para suportar trabalho cooperativo apoiado por computador (CSCW, Computer-Supported Cooperative Working), em que podem ocorrer atualizações conflitantes entre usuários separados geograficamente [Kindberg *et al.* 1996, Sun e Ellis 1998]. Na prática, a estratégia é limitada às aplicações em que os conflitos são relativamente

raros, em que a semântica dos dados subjacentes é relativamente simples e em que os usuários podem aceitar informações de tentativa.

18.4.3 O sistema de arquivos Coda

O sistema de arquivos Coda é um descendente do AFS (veja a Seção 12.4) que tem como objetivo tratar de vários requisitos que o AFS não atende – particularmente, o requisito de fornecer alta disponibilidade a despeito da operação desconectada. Ele foi desenvolvido em um projeto de pesquisa empreendido por Satyanarayanan e seus colegas da Carnegie-Mellon University [Satyanarayanan *et al.* 1990; Kistler e Satyanarayanan 1992]. Os requisitos de projeto do Coda foram extraídos da experiência com o AFS na CMU e em outros lugares, envolvendo seu uso em sistemas distribuídos de larga escala, em redes de comunicação locais e de longa distância.

Embora o desempenho e a facilidade de administração do AFS sejam considerados satisfatórios sob as condições de uso na CMU, sentiu-se que a forma limitada de replicação (restrita a volumes somente de leitura) oferecida pelo AFS se tornaria um fator limitante em alguma escala, especialmente para o acesso a arquivos compartilhados em larga escala, como as listas de discussões eletrônicas e outros tipos de bancos de dados.

Além disso, havia espaço para aprimoramento na disponibilidade do serviço oferecido pelo AFS. As dificuldades mais comuns experimentadas pelos usuários do AFS surgiam da falha (ou da interrupção agendada) de servidores e componentes da rede. A escala do sistema na CMU era tal que poucas falhas de serviço ocorriam diariamente e elas podiam atrapalhar seriamente muitos usuários por períodos de tempo que variavam de alguns minutos até muitas horas.

Finalmente, estava surgindo um modo de uso do computador que o AFS não fornecia – o de computadores portáteis. Isso levou ao requisito de tornar disponíveis todos os arquivos necessários para um usuário continuar com seu trabalho, enquanto estivesse desconectado da rede, sem contar com métodos manuais para o gerenciamento da localização dos arquivos.

O Coda tem por objetivo atender a todos esses três requisitos sob o título geral de *disponibilidade constante de dados*. O objetivo era fornecer aos usuários as vantagens de um repositório de arquivos compartilhado, mas permitir que eles contassem inteiramente com recursos locais quando o repositório estivesse parcial ou totalmente inacessível. Além desses objetivos, o Coda mantém os objetivos originais do AFS com relação à escalabilidade e à simulação da semântica de arquivo do UNIX.

Em contraste com o AFS, no qual volumes de leitura e escrita são armazenados em apenas um servidor, o projeto do Coda conta com a replicação de volumes de arquivo para obter desempenho de saída (*throughput*) mais alto das operações de acesso a arquivo e um maior grau de tolerância a falhas. Além disso, o Coda conta com uma ampliação do mecanismo usado no AFS para armazenar cópias dos arquivos na cache dos computadores clientes, para permitir que esses computadores operem quando não estiverem conectados na rede.

Veremos que o Coda é como o Bayou (veja a Seção 18.4.2) no que diz respeito a seguir uma estratégia otimista. Isto é, ele permite que os clientes atualizem dados enquanto o sistema está particionado, baseado em que os conflitos são relativamente improváveis e que podem ser corrigidos, caso ocorram. Assim como o Bayou, ele detecta conflitos, mas, ao contrário do Bayou, realiza essa verificação sem considerar a semântica dos dados armazenados nos arquivos. E, ao contrário do Bayou, ele só fornece suporte de sistema limitado para solucionar réplicas conflitantes.

A arquitetura do Coda • O Coda executa o que chama de processos *Venus* nos computadores clientes e processos *Vice* nos computadores servidores de arquivo, adotando a

terminologia do AFS. Os processos Vice são o que chamamos de gerenciadores de réplica. Os processos Venus são uma mistura de *front-ends* e gerenciadores de réplica. Eles desempenham o papel do *front-end*, ocultando a implementação do serviço dos processos clientes locais; mas, como gerenciam uma cache local de arquivos, eles também são gerenciadores de réplica para os processos Vice, embora de um tipo diferente.

O conjunto de servidores contendo réplicas de um volume de arquivo é conhecido como *grupo de armazenamento de volume* (VSG, *Volume Storage Group*). A qualquer instante, um cliente que queira abrir um arquivo nesse volume pode acessar algum subconjunto do VSG, conhecido como *grupo de armazenamento de volume disponível* (AVSG, *Available Volume Storage Group*). A participação como membro do AVSG varia à medida que os servidores se tornam acessíveis ou inacessíveis por causa de falhas da rede ou do servidor.

Normalmente, o acesso a arquivo do Coda ocorre de maneira semelhante ao do AFS, com cópias dos arquivos armazenadas em cache sendo fornecidas para os computadores clientes por qualquer um dos servidores no AVSG corrente. Assim como no AFS, os clientes são notificados das alterações por meio de um mecanismo de *promessa de callback*, mas isso agora depende de um mecanismo adicional para a distribuição de atualizações para cada réplica. No *fechamento de um arquivo (close)*, as cópias dos arquivos modificados são transmitidas em paralelo para todos os servidores no AVSG.

No Coda, diz-se que a operação desconectada ocorre quando o AVSG está vazio. Isso pode ser devido a falhas de rede ou servidor ou pode ser uma consequência da desconexão deliberada do computador cliente, como no caso de um *notebook*. A operação desconectada efetiva conta com a presença, na cache do computador cliente, de *todos* os arquivos exigidos para que o trabalho do usuário prossiga. Para conseguir isso, o usuário precisa colaborar com o Coda na geração de uma lista de arquivos que devem ser colocados na cache. É fornecida uma ferramenta que regista, em uma lista, o histórico da utilização dos arquivos enquanto está na operação conectada e isso serve como base para prever a utilização enquanto está na operação desconectada.

Um princípio de projeto do Coda é o fato de que as cópias dos arquivos residentes nos servidores sejam mais confiáveis do que as que residem nas caches dos computadores clientes. Embora seja possível construir logicamente um sistema de arquivos que conte inteiramente com cópias de arquivos armazenadas em cache de computadores clientes, é improvável que seja obtida uma qualidade satisfatória de serviço. Os servidores Coda existem para fornecer a qualidade de serviço necessária. As cópias dos arquivos residentes nas caches do computadores clientes são consideradas úteis somente enquanto sua validade puder ser periodicamente revalidada em relação às cópias residentes nos servidores. No caso da operação desconectada, a revalidação ocorre quando a operação desconectada cessa e os arquivos armazenados em cache são reintegrados com os dos servidores. No pior caso, isso pode exigir alguma intervenção manual para resolver inconsistências ou conflitos.

A estratégia da replicação • A estratégia de replicação do Coda é otimista – ela permite que a modificação dos arquivos prossiga quando a rede é particionada ou durante a operação desconectada. Ela conta com a anexação, em cada versão de um arquivo, de um *vetor de versão Coda* (CVV, *Coda Version Vector*). Um CVV é um carimbo de tempo vetorial com um elemento para cada servidor no VSG relevante. Cada elemento do CVV é uma estimativa do número de modificações realizadas na versão do arquivo mantida no servidor correspondente. O objetivo dos CVVs é fornecer informações suficientes sobre o histórico de atualização de cada réplica de arquivo para permitir que conflitos em potencial possam ser detectados e submetidos à intervenção manual e para que as réplicas antigas sejam atualizadas automaticamente.

Se o CVV de um dos *sites* for maior ou igual a todos os CVVs correspondentes nos outros *sites* (a Seção 14.4 definiu o significado de $v_1 \geq v_2$ para carimbos de tempo vetoriais v_1 e v_2), então não existe conflito. As réplicas mais antigas (com carimbos de tempo rigorosamente menores) incluem todas as atualizações em uma réplica mais recente e podem ser atualizadas automaticamente com ela.

Quando isso não acontece, isto é, quando nem $v_1 \geq v_2$ nem $v_2 \geq v_1$ valem para dois CVVs, então existe um conflito: cada réplica reflete pelo menos uma atualização que a outra não reflete. Em geral, o Coda não resolve conflitos automaticamente. O arquivo é marcado como “inoperante” e o proprietário do arquivo é informado sobre o conflito.

Quando um arquivo modificado é fechado, cada *site* presente no AVSG corrente recebe uma mensagem de atualização enviada pelo processo Venus no cliente, contendo o CVV corrente e o novo conteúdo do arquivo. O processo Vice em cada *site* verifica o CVV e, se ele for maior do que o correntemente mantido, armazena o novo conteúdo do arquivo e retorna um reconhecimento positivo. Então, o processo Venus calcula um novo CVV com os contadores de modificação incrementados para os servidores que responderam positivamente à mensagem de atualização e distribui o novo CVV para os membros do AVSG.

Como a mensagem é enviada apenas para os membros do AVSG, e não do VSG, os servidores que não estão no AVSG corrente não recebem o novo CVV. Portanto, qualquer CVV sempre conterá um contador de modificação preciso do servidor local, mas os contadores dos servidores não locais, em geral, serão menores, pois só serão atualizados quando o servidor receber uma mensagem de atualização.

O quadro a seguir contém um exemplo ilustrando o uso de CVVs para gerenciar a atualização de um arquivo replicado em três *sites*. Mais detalhes sobre o uso de CVVs

Exemplo: considere uma sequência de modificações em um arquivo F , em um volume que é replicado em 3 servidores, S_1 , S_2 e S_3 . O VSG de F é $\{S_1, S_2, S_3\}$. F é modificado praticamente ao mesmo tempo por dois clientes C_1 , C_2 . Devido a uma falha da rede, C_1 só pode acessar S_1 e S_2 (o AVSG de C_1 é $\{S_1, S_2\}$) e C_2 só pode acessar S_3 (o AVSG de C_2 é $\{S_3\}$).

1. Inicialmente, os CVVs de F em todos os 3 servidores são iguais, digamos [1,1,1].
2. C_1 executa um processo que abre F , o modifica e depois fecha. O processo Venus em C_1 transmite uma mensagem de atualização para seu AVSG, $\{S_1, S_2\}$, resultando finalmente nas novas versões de F e um CVV [2,2,1] em S_1 e S_2 , mas não muda em S_3 .
3. Nesse meio-tempo, C_2 executa dois processos, cada um dos quais abre F , o modifica e depois fecha. O processo Venus em C_2 transmite uma mensagem de atualização para seu AVSG, $\{S_3\}$, após cada modificação, resultando finalmente em uma nova versão de F e um CVV [1,1,3] em S_3 .
4. Algum tempo depois, a falha da rede é reparada e C_2 realiza uma verificação de rotina para ver se os membros inacessíveis do VSG se tornaram acessíveis (o processo por meio do qual tais verificações são feitas será descrito posteriormente) e descobre que S_1 e S_2 agora estão acessíveis. Ele modifica seu AVSG para $\{S_1, S_2, S_3\}$, para o volume que contém F , e solicita os CVVs de F de todos os membros do novo AVSG. Quando eles chegam, C_2 descobre que S_1 e S_2 tem, cada um, os CVVs [2,2,1], enquanto S_3 tem [1,1,3]. Isso representa um *conflito*, exigindo intervenção manual para atualizar F , de maneira que minimize a perda de informações de atualização.

Por outro lado, considere um cenário semelhante, porém, mais simples, que segue a mesma sequência de eventos acima, mas omitindo o item (3), de modo que F não é modificado por C_2 . Portanto, o CVV em S_3 permanece inalterado como [1,1,1] e, quando a falha da rede é reparada, C_2 descobre que os CVVs em S_1 e S_2 ([2,2,1]) predominam sobre o de S_3 . A versão do arquivo em S_1 ou S_2 deve substituir a que está em S_3 .

para o gerenciamento de atualizações podem ser encontrados em Satyanarayanan *et al.* [1990]. Os CVVs são baseados nas técnicas de replicação usadas no sistema Locus [Poppek e Walker 1985].

Na operação normal, o comportamento do Coda parece semelhante ao AFS. Uma perda de cache é transparente para os usuários e apenas impõe uma penalidade no desempenho. As vantagens derivadas da replicação de alguns (ou de todos os) volumes de arquivo em vários servidores são:

- Os arquivos em um volume replicado permanecem acessíveis para qualquer cliente que possa acessar pelo menos uma das réplicas.
- O desempenho do sistema pode ser melhorado pelo compartilhamento, entre todos os servidores que contêm réplicas, de parte do trabalho do atendimento às requisições de cliente em um volume replicado.

Na operação desconectada (quando nenhum dos servidores de um volume pode ser acessado pelo cliente), uma perda de cache impede qualquer avanço, e a computação é suspensa até que a conexão seja restabelecida ou que o usuário cancele o processo. Portanto, é importante carregar a cache antes que a operação desconectada comece, para que as perdas possam ser evitadas.

Em resumo, comparado com o AFS, o Coda melhora a disponibilidade tanto pela replicação de arquivos nos servidores como pela capacidade de os clientes operarem inteiramente fora de suas caches. Os dois métodos dependem do uso de uma estratégia otimista para a detecção de conflitos de atualização na presença de particionamentos da rede. Os mecanismos são complementares e independentes entre si. Por exemplo, um usuário pode explorar os benefícios da operação desconectada, mesmo que os volumes de arquivo exigidos estejam armazenados em um único servidor.

Semântica de atualização • As garantias de vigência oferecidas pelo Coda quando um cliente abre um arquivo são menores do que as do AFS, refletindo a estratégia de atualização otimista. O servidor único \bar{S} , referenciado nas garantias de vigência do AFS, é substituído por um conjunto de servidores S (o VSG do arquivo) e o cliente C pode acessar um subconjunto de servidores s (o AVSG do arquivo visto por C).

Informalmente, a garantia oferecida por uma operação de *abertura de arquivo* (*open*) bem-sucedida no Coda é que ela fornece a cópia mais recente de F do AVSG corrente e, se nenhum servidor estiver acessível, uma cópia de F armazenada na cache local será usada, caso haja uma disponível. Uma operação de *fechamento* (*close*) bem-sucedida garante que o arquivo foi propagado para o conjunto de servidores correntemente acessíveis ou, se nenhum servidor estiver disponível, que o arquivo foi marcado para propagação na próxima oportunidade.

Uma definição mais precisa dessas garantias, levando em conta o efeito de *callbacks* perdidos, pode ser dada usando-se uma extensão da notação usada pelo AFS. Em cada definição, exceto na última, existem dois casos: o primeiro, começando com $\bar{s} \neq \emptyset$, refere-se a todas as situações em que o AVSG não está vazio; o segundo trata da operação desconectada:

após uma operação de *abertura* bem-sucedida: $(\bar{s} \neq \emptyset \text{ e } (\text{latest}(F, \bar{s}, 0)$
ou $(\text{latest}(F, \bar{s}, T) \text{ e } \text{lostCallback}(\bar{s}, T)$
e $\text{inCache}(F)))$
ou $(\bar{s} = \emptyset \text{ e } \neg \text{inCache}(F))$

após uma operação de *abertura* malsucedida: $(\bar{s} \neq \emptyset \text{ e } \text{conflict}(F, \bar{s}))$
ou $(\bar{s} = \emptyset \text{ e } \neg \text{inCache}(F))$

após uma operação de *fechamento* bem-sucedida: $(\bar{s} \neq \emptyset \text{ e } updated(F, \bar{s}))$
ou $(\bar{s} = \emptyset)$

após uma operação de *fechamento* malsucedida: $\bar{s} \neq \emptyset \text{ e } conflict(F, \bar{s})$

Esse modelo presume um sistema síncrono: T é o tempo mais longo durante o qual um cliente pode permanecer sem saber de uma atualização em qualquer parte de um arquivo que está em sua cache; $latest(F, \bar{s}, T)$ denota o fato de que o valor corrente de F , em C , era o mais recente em todos os servidores em s , em algum instante nos últimos T segundos, e que não havia nenhum conflito entre as cópias de F nesse instante; $lostCallback(\bar{s}, T)$ significa que um *callback* foi enviado por algum membro de s nos últimos T segundos e que não foi recebido em C ; e $conflict(F, \bar{s})$ significa que os valores de F em alguns servidores em s estão correntemente em conflito.

Acesso a réplicas • A estratégia usada nas operações de *abertura* e *fechamento* para acessar as réplicas de um arquivo é uma variante da estratégia *um lê/todos escrevem* (*read-one/write-all*), descrita na Seção 18.5. Na *abertura*, se uma cópia do arquivo não estiver presente na cache local, o cliente identificará um servidor preferido do AVSG do arquivo. O servidor preferido pode ser escolhido aleatoriamente, ou de acordo com critérios de desempenho, como a proximidade física ou a carga no servidor. O cliente solicita uma cópia dos atributos e do conteúdo do arquivo do servidor preferido e, ao recebê-la, consulta todos os outros membros do AVSG para verificar se a cópia é a versão mais recente disponível. Se não for, um membro do AVSG com a versão mais recente se torna o *site* preferido, o conteúdo do arquivo é novamente buscado e os membros do AVSG são notificados de que alguns membros possuem réplicas antigas. Quando a busca termina, uma promessa de *callback* é estabelecida no servidor preferido.

Quando um arquivo é fechado em um cliente, após a modificação, seu conteúdo e seus atributos são transmitidos em paralelo para todos os membros do AVSG, usando um protocolo de chamada remota de procedimento por *multicast*. Isso maximiza a probabilidade de que todo *site* de replicação de um arquivo tenha a versão corrente o tempo todo. Não há garantia disso, pois o AVSG não inclui necessariamente todos os membros do VSG. Esse processo minimiza a carga do servidor, passando aos clientes a responsabilidade de propagar as alterações nos *sites* de replicação no caso normal (os servidores são envolvidos apenas quando uma réplica antiga é descoberta na operação de *abertura*).

Como seria dispendioso manter o estado de *callback* em todos os membros de um AVSG, a promessa de *callback* é mantida apenas no servidor preferido. No entanto, isso introduz um novo problema: o servidor preferido de um cliente não precisa estar no AVSG de outro cliente. Se esse for o caso, uma atualização feita pelo segundo cliente não causará um *callback* para o primeiro cliente. A solução adotada para esse problema será discutida na próxima subseção.

Coerência de cache • As garantias de vigência do Coda mencionadas anteriormente significam que o processo Venus em cada cliente deve detectar os seguintes eventos dentro de T segundos a partir de sua ocorrência:

- ampliação de um AVSG (devido à acessibilidade de um servidor anteriormente inacessível);
- redução de um AVSG (devido ao fato de um servidor se tornar inacessível);
- um evento de *callback* perdido.

Para conseguir isso, a cada T segundos, o processo Venus envia uma mensagem de verificação (*probe*) para todos os servidores nos VSGs dos arquivos que possui em sua cache.

Serão recebidas respostas apenas dos servidores acessíveis. Se o processo Venus receber uma resposta de um servidor anteriormente inacessível, ele ampliará o AVSG correspondente e eliminará do volume relevante as promessas de *callback* em todos os arquivos que as contém. Isso é feito porque a cópia armazenada na cache pode não ser mais a versão mais recente disponível no novo AVSG.

Se deixar de receber uma resposta de um servidor anteriormente acessível, o processo Venus reduzirá o AVSG correspondente. Nenhuma alteração de *callback* é exigida, a não ser que a diminuição seja causada pela perda de um servidor preferido, no caso em que todas as promessas de *callback* desse servidor deverão ser eliminadas. Se uma resposta indicar que uma mensagem de *callback* foi enviada, mas não recebida, a promessa de *callback* no arquivo correspondente será eliminada.

Agora, ficamos com o problema mencionado anteriormente, das atualizações perdidas por um servidor por que ele não está no AVSG de um cliente diferente que realiza uma atualização. Para tratar desse caso, o processo Venus recebe um *vetor de versão de volume* (*CVV de volume*) em resposta a cada mensagem de verificação. O CVV de volume contém um resumo dos CVVs de todos os arquivos presentes no volume. Se o processo Venus detecta qualquer diferença entre os CVVs de volume, então alguns membros do AVSG devem ter algumas versões de arquivo que não estão atualizadas. Embora os arquivos desatualizados possam não ser aqueles que estão em sua cache local, o processo Venus faz uma suposição pessimista e elimina do volume relevante as promessas de *callback* em todos os arquivos que as contém.

Note que o processo Venus só verifica os servidores nos VSGs dos arquivos para os quais contém cópias na cache e que uma única mensagem de verificação serve para atualizar os AVSGs e verificar os *callbacks* de todos os arquivos de um volume. Isso, combinado com um valor relativamente grande de T (na ordem de 10 minutos na implementação experimental), significa que as verificações não são um obstáculo para a escalabilidade do Coda para grandes quantidades de servidores e de redes de longa distância.

Operação desconectada • Durante as desconexões breves, como as que podem ocorrer devido a interrupções inesperadas do serviço, a política de substituição da cache usada menos recentemente, normalmente adotada pelo processo Venus, pode ser suficiente para evitar perdas na cache nos volumes desconectados. Contudo, é improvável que um cliente possa operar no modo desconectado por longos períodos, sem gerar referências para arquivos ou diretórios que não estejam na cache, a não ser que uma política diferente seja adotada.

Portanto, o Coda permite que os usuários especifiquem uma lista de prioridade dos arquivos e diretórios que o processo Venus deve se esforçar para manter na cache. Os objetos que estão no nível mais alto são identificados como *aderentes* e devem ser mantidos na cache o tempo todo. Se o disco local for grande o suficiente para acomodar todos eles, o usuário será assegurado de que eles permanecerão acessíveis. Como frequentemente é difícil saber exatamente quais acessos de arquivo são gerados por qualquer sequência de ações do usuário, é fornecida uma ferramenta que permite ao usuário definir uma sequência de ações; o processo Venus anota as referências de arquivo geradas pela sequência e as marca com a prioridade dada.

Quando a operação desconectada termina, começa um processo de *reintegração*. Para cada arquivo ou diretório em cache que foi modificado, criado ou excluído durante a operação desconectada, o processo Venus executa uma sequência de operações de atualização para tornar as réplicas do AVSG idênticas à cópia armazenada em cache. A reintegração ocorre de cima para baixo, a partir da raiz de cada volume colocado na cache.

Podem ser detectados conflitos durante a reintegração, resultantes das atualizações nas réplicas do AVSG feitas por outros clientes. Quando isso ocorre, a cópia armazenada

na cache é posta em um local temporário no servidor, e o usuário que iniciou a reintegração é informado. Essa estratégia é baseada na filosofia de projeto adotada no Coda, que atribui prioridade maior às réplicas baseadas no servidor do que às cópias em cache. As cópias temporárias são armazenadas em um *covolume*, que é associado a cada volume em um servidor. Os covolumes são parecidos com os diretórios *lost+found* encontrados nos sistemas UNIX convencionais. Eles espelham apenas as partes da estrutura de diretório de arquivos necessárias para conter os dados temporários. Pouco armazenamento adicional é exigido, pois os covolumes são quase vazios.

Desempenho • Satyanarayanan *et al.* [1990] compararam o desempenho do Coda com o AFS com *benchmarks* projetados para simular populações de usuários variando de cinco a 50 usuários típicos do AFS.

Sem nenhuma replicação, há pouca diferença significativa entre o desempenho do AFS e do Coda. Com replicação tripla, o tempo para o Coda executar um *benchmark* que simula a carga equivalente a 5 usuários típicos ultrapassou a do AFS sem duplicação em apenas 5%. Entretanto, com replicação tripla e uma carga equivalente a 50 usuários, o tempo para o *benchmark* aumentou em 70%, enquanto o do AFS sem replicação aumentou em apenas 16%. Essa diferença é atribuída apenas em parte às sobrecargas associadas à replicação – as diferenças na otimização da implementação são responsáveis por parte da diferença no desempenho.

Discussão • Mencionamos anteriormente que o Coda é semelhante ao Bayou porque também emprega uma estratégia otimista para obter alta disponibilidade (embora eles sejam diferentes em vários outros aspectos e não apenas porque um gerencia arquivos e o outro, bancos de dados). Também descrevemos o modo como o Coda usa CVVs para verificar a existência de conflitos, sem considerar a semântica dos dados armazenados nos arquivos. A estratégia pode detectar conflitos de escrita-escrita em potencial, mas não conflitos de leitura-escrita. Existe o potencial de conflitos de escrita-escrita porque, em nível da semântica da aplicação, não pode haver nenhum conflito real: os clientes podem ter atualizado compativelmente objetos diferentes do arquivo e uma simples integração automática seria possível.

A estratégia global do Coda de detecção de conflito livre de semântica e solução manual é sensata em muitos casos, especialmente em aplicações que exigem julgamento humano ou em sistemas sem conhecimento da semântica dos dados.

Os diretórios são um caso especial do Coda. Às vezes, é possível manter automaticamente a integridade desses objetos importantes por meio da solução de conflitos, pois sua semântica é relativamente simples: as únicas alterações que podem ser feitas nos diretórios são a inserção ou a exclusão de entradas de diretório. O Coda incorpora seu próprio método para solucionar diretórios. Ele tem o mesmo efeito da estratégia de transformação operacional do Bayou, mas o Coda integra diretamente o estado de diretórios conflitantes, pois não tem nenhum registro das operações executadas pelos clientes.

Replicação no Dynamo • A Seção 16.7 apresentou o Dynamo, o serviço de armazenamento utilizado pela Amazon para aplicações, como os carrinhos de compra, que exigem somente acesso por chave/valor. No Dynamo [DeCandia *et al.* 2007], os dados são particionados e replicados; todas as atualizações chegam a todas as réplicas.

Assim como o Bayou e o Coda, o Dynamo usa técnicas de replicação otimistas; as alterações podem ser propagadas para as réplicas em segundo plano e é tolerado trabalho concorrente e desconectado. Essa estratégia pode levar a alterações conflitantes, que devem ser detectadas e resolvidas.

No Dynamo, as escritas são sempre aceitas e gravadas como versões imutáveis – para que os compradores sempre possam adicionar e remover itens de seus carrinhos de compra.

São usados carimbos de tempo vetoriais para determinar a ordenação causal entre diferentes versões do mesmo objeto. Os carimbos de tempo são comparados conforme descrito na Seção 14.4. Quando o carimbo de tempo vetorial de uma versão é menor do que a de outra, a versão mais antiga é descartada. Caso contrário, as duas versões estão em conflito e devem ser resolvidas. As duas versões dos dados são armazenadas e, então, fornecidas para um cliente como resultado de uma operação de leitura.

Esse cliente é responsável por resolver o conflito. O Dynamo fornece tanto a estratégia em nível de aplicação do Bayou como a estratégia em nível de sistema do Coda. A primeira é usada para carrinhos de compra, onde todas as operações de *adicionar item* em versões conflitantes são mescladas e, às vezes, um item excluído pode reaparecer. Quando não pode ser usada semântica de aplicação, o Dynamo usa solução baseada em carimbo de tempo simples – o objeto com o valor de carimbo de tempo físico maior é escolhido como a versão correta.

18.5 Transações em dados replicados

Até aqui, neste capítulo, consideramos sistemas nos quais os clientes solicitam uma única operação por vez em conjuntos de objetos replicados. Os Capítulos 16 e 17 explicaram que as transações são *sequências* de uma ou mais operações, aplicadas de maneira a impor as propriedades ACID. Assim como acontece nos sistemas da Seção 18.4, nos sistemas transacionais os objetos podem ser replicados para aumentar a disponibilidade e o desempenho.

Do ponto de vista do cliente, uma transação sobre objetos replicados deve ser idêntica à dos objetos não replicados. Em um sistema não replicado, as transações parecem ser executadas uma por vez, em alguma ordem. Isso é conseguido garantindo-se uma interposição serialmente equivalente das transações dos clientes. O efeito das transações executadas pelos clientes sobre objetos replicados deve ser igual ao que se eles tivessem efetuado uma por vez, sobre um único conjunto de objetos. Essa propriedade é chamada de *capacidade de serialização de uma cópia*. Ela é semelhante à consistência sequencial, mas não deve ser confundida com esta. A consistência sequencial considera execuções válidas, sem nenhuma noção de agregação das operações do cliente nas transações.

Cada gerenciador de réplica fornece controle de concorrência e recuperação de seus próprios objetos. Nesta seção, presumimos que o travamento de duas fases é usado para controle de concorrência.

A recuperação é complicada pelo fato de que um gerenciador de réplica falho é membro de um conjunto, e que os outros membros continuam a fornecer um serviço durante o tempo em que ele está indisponível. Quando um gerenciador de réplica se recupera de uma falha, ele usa informações obtidas dos outros gerenciadores de réplica para restaurar seus objetos com seus valores correntes, levando em conta todas as alterações ocorridas durante o tempo em que estava indisponível.

Esta seção apresenta primeiro a arquitetura das transações com dados replicados. As principais questões são: se a requisição de um cliente pode ser tratada em qualquer um dos gerenciadores de réplica; como muitos gerenciadores de réplica são exigidos para a conclusão bem-sucedida de uma operação; se o gerenciador de réplica contatado por um cliente pode adiar o encaminhamento das requisições até que uma transação seja confirmada; e como executar um protocolo de confirmação de duas fases.

A implementação da capacidade de serialização de uma cópia é ilustrada pela estratégia *um lê/todos escrevem* – um esquema de replicação simples no qual as operações de *leitura* são executadas por um único gerenciador de réplica e as operações de *escrita* são realizadas por todos.

Em seguida, a seção discutirá os problemas da implementação de esquemas de replicação na presença de falhas e recuperação de servidores. Será apresentada a replicação de cópias disponíveis – uma variante do esquema de replicação um lê/todos escrevem, na qual as operações *de leitura* são realizadas por um único gerenciador de réplica e as operações *de escrita* são efetuadas por todos aqueles que estiverem disponíveis.

Finalmente, a seção apresentará três esquemas de replicação que funcionam corretamente quando o conjunto de gerenciadores de réplica é dividido em subgrupos por um particionamento da rede:

- *Cópias disponíveis com validação*: a replicação de cópias disponíveis é aplicada em cada partição e, quando o particionamento for reparado, é aplicado um procedimento de validação e todas as inconsistências são tratadas.
- *Consenso de quorum*: um subgrupo deve ter *quorum* (significando que ele tem membros suficientes) para poder continuar fornecendo um serviço na ocorrência de um particionamento. Quando um particionamento é reparado (e quando um gerenciador de réplica é reiniciado, após uma falha), os gerenciadores de réplica atualizam seus objetos por intermédio de procedimentos de recuperação.
- *Partição virtual*: uma combinação do consenso de *quorum* e das cópias disponíveis. Se uma partição virtual tem *quorum*, ele pode usar replicação de cópias disponíveis.

18.5.1 Arquiteturas para transações replicadas

Assim como na variedade de sistemas que já consideramos nas seções anteriores, um *front-end* pode enviar as requisições do cliente através de *multicast* para os grupos de gerenciadores de réplica ou pode enviar cada requisição para um único gerenciador de réplica, o qual fica, então, responsável por processar a requisição e responder para o cliente. Wiesmann *et al.* [2000] e Schiper e Raynal [1996] consideram o caso do *multicast* das requisições, e não vamos tratar disso aqui. Daqui por diante, supomos que um *front-end* envia requisições do cliente para um dos gerenciadores do grupo de gerenciadores de réplica de um objeto lógico. Na estratégia da *cópia primária*, todos os *front-ends* se comunicam com um gerenciador de réplica primário determinado para executar uma operação, e esse gerenciador de réplica mantém os *backups* atualizados. Como alternativa, os *front-ends* podem se comunicar com qualquer gerenciador de réplica para executar uma operação – mas a coordenação entre os gerenciadores de réplica é, consequentemente, mais complexa.

O gerenciador de réplica que recebe uma requisição para executar uma operação sobre um objeto em particular fica responsável por obter a cooperação dos outros gerenciadores de réplica do grupo que tenham cópias desse objeto. Diferentes esquemas de replicação têm diferentes regras a respeito de quantos gerenciadores de réplica de um grupo são exigidos para a conclusão bem-sucedida de uma operação. Por exemplo, no esquema um lê/todos escrevem, uma requisição *de leitura* pode ser executada por um único gerenciador de réplica, enquanto uma requisição *de escrita* deve ser executada por todos os gerenciadores de réplica do grupo, como mostrado na Figura 18.9 (pode haver diferentes números de réplicas dos vários objetos). Os esquemas de consenso de *quorum* são projetados para reduzir o número de gerenciadores de réplica que devem efetuar operações de atualização, mas ao custo de um maior número de gerenciadores de réplica para executar operações *somente de leitura*.

Outra questão é se o gerenciador de réplica contatado por um *front-end* deve adiar o encaminhamento das requisições de atualização para outros gerenciadores de réplica do grupo, até que uma transação seja confirmada – a assim chamada estratégia *preguiçosa* de propagação de atualização – ou, inversamente, se os gerenciadores de réplica

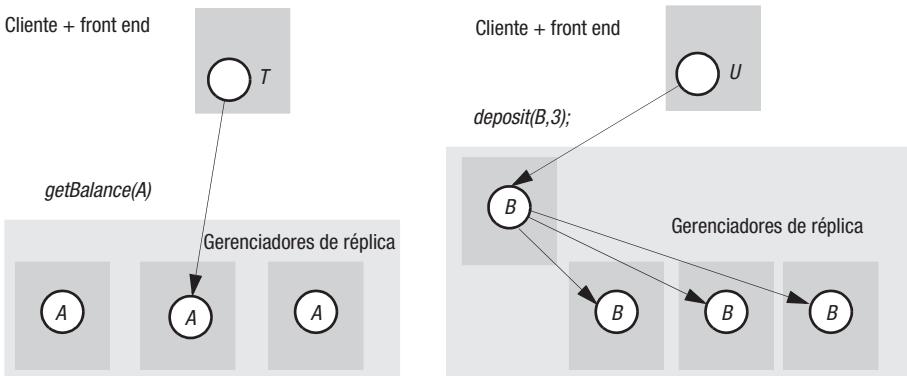


Figura 18.9 Transações sobre dados replicados.

devem encaminhar cada requisição de atualização para todos os gerenciadores de réplica necessários dentro da transação e antes de confirmar – a estratégia *ávida*. A estratégia “preguiçosa” é uma alternativa atraente, pois reduz a quantidade de comunicação entre os gerenciadores de réplica que ocorre antes da resposta ao cliente que está fazendo a atualização. Entretanto, o controle de concorrência também deve ser considerado. Às vezes, a estratégia “preguiçosa” é usada na replicação de cópia primária (veja a seguir), em que um único gerenciador de réplica primário serializa as transações. Contudo, se várias transações diferentes puderem tentar acessar os mesmos objetos em gerenciadores de réplica diferentes de um grupo, então, para garantir que as transações sejam serializadas corretamente em todos os gerenciadores de réplica do grupo, cada gerenciador de réplica precisa saber a respeito das requisições feitas pelos outros. A estratégia “ávida” é a única disponível, nesse caso.

O protocolo de confirmação de duas fases • O protocolo de confirmação de duas fases se torna um protocolo com dois níveis de aninhamento. Como antes, o coordenador de uma transação se comunica com processos operários. No entanto, se o coordenador, ou um operário, for um gerenciador de réplica, ele se comunicará com os outros gerenciadores de réplica para os quais passou requisições durante a transação.

Isto é, na primeira fase, o coordenador envia a requisição de *canCommit?* para os operários, os quais a passam para os outros gerenciadores de réplica e reúnem suas respostas, antes de responder para o coordenador. Na segunda fase, o coordenador envia a requisição de *doCommit* ou *doAbort*, a qual é passada para os membros dos grupos de gerenciadores de réplica.

Replicação da cópia primária • A replicação da cópia primária pode ser usada no contexto das transações. Nesse esquema, todas as requisições de cliente (sejam elas somente de leitura ou não) são direcionadas para um único gerenciador de réplica primário (veja a Figura 18.3). Para a replicação da cópia primária, o controle de concorrência é aplicado no gerenciador primário. Para confirmar uma transação, o gerenciador primário se comunica com os gerenciadores de réplica de *backup* e, então, na estratégia “ávida”, responde para o cliente. Essa forma de replicação permite que um gerenciador de réplica de *backup* assuma o controle consistentemente, caso o gerenciador primário venha a falhar. Na alternativa “preguiçosa”, o gerenciador primário responde para os *front-ends* antes de ter atualizado seus *backups*. Neste caso, um gerenciador de *backup* que venha a substituir um *front-end* falho não terá necessariamente o estado mais recente do banco de dados.

Um lê/todos escrevem (read-one/write-all) • Usamos esse esquema de replicação simples para ilustrar como o travamento de duas fases em cada gerenciador de réplica pode ser usado para obter capacidade de serialização de uma cópia, em que os *front-ends* podem se comunicar com qualquer gerenciador de réplica. Toda operação de *escrita* deve ser executada em todos os gerenciadores de réplica, cada um dos quais configura uma trava de escrita sobre o objeto afetado pela operação. Cada operação de *leitura* é executada por um único gerenciador de réplica, o qual configura uma trava de leitura sobre o objeto afetado pela operação.

Considere pares de operações de diferentes transações sobre o mesmo objeto: qualquer par de operações de *escrita* exigirá travas conflitantes em todos os gerenciadores de réplica; uma operação de *leitura* e uma operação de *escrita* exigirão travas conflitantes em um único gerenciador de réplica. Assim, a capacidade de serialização de uma cópia é obtida.

18.5.2 Replicação de cópias disponíveis

A replicação um lê/todos escrevem simples não é um esquema realista, pois não pode ser realizado se alguns dos gerenciadores de réplica estiverem indisponíveis, ou porque falharam, ou devido a uma falha de comunicação. O esquema das cópias disponíveis é projetado para possibilitar que alguns gerenciadores de réplica estejam temporariamente indisponíveis. A estratégia usa o fato de que a requisição de *leitura* de um cliente sobre um objeto lógico pode ser executada por qualquer gerenciador de réplica disponível, mas a requisição de atualização de um cliente deve ser executada por todos os gerenciadores de réplica disponíveis no grupo que contêm cópias do objeto. A ideia de “membros disponíveis de um grupo de gerenciadores de réplica” é semelhante ao grupo de armazenamento de volume disponível do Coda, descrito na Seção 18.4.3.

No caso normal, as requisições do cliente são recebidas e executadas por um gerenciador de réplica que esteja funcionando. As requisições de *leitura* podem ser executadas pelo gerenciador de réplica que as receber. As requisições de *escrita* são executadas pelo gerenciador de réplica receptor e por todos os outros gerenciadores de réplica disponíveis no grupo. Por exemplo, na Figura 18.10, a operação *getBalance* da transação *T* é efetuada por *X*, enquanto sua operação *deposit* é executada por *M*, *N* e *P*. O controle de concorrência em cada gerenciador de réplica afeta as operações executadas de forma local. Por exemplo, em *X*, a transação *T* leu *A* e, portanto, a transação *U* não pode atualizar *A* com a operação *deposit* até que a transação *T* tenha terminado. Desde que o conjunto de gerenciadores de réplica disponíveis não mude, o controle de concorrência local obtém a capacidade de serialização de uma cópia da mesma maneira que na replicação um lê/todos escrevem. Infelizmente, isso não acontece se um gerenciador de réplica falha ou se recupera durante o andamento das transações conflitantes.

Falha do gerenciador de réplica • Supomos que os gerenciadores de réplica falham benignamente por colapso. Entretanto, um gerenciador de réplica que apresenta falha por colapso é substituído por um novo processo, o qual recupera o estado confirmado dos objetos a partir de um arquivo de recuperação. Os *front-ends* usam tempos limites para decidir se um gerenciador de réplica não está correntemente disponível. Quando um cliente faz uma requisição para um gerenciador de réplica que falhou, o *front-end* atinge o tempo limite e tenta novamente, em outro gerenciador de réplica do grupo. Se a requisição for recebida por um gerenciador de réplica no qual o objeto está desatualizado, porque o gerenciador não se recuperou completamente da falha, ele rejeita a requisição, e o *front-end* tenta novamente em outro gerenciador de réplica do grupo.

A serialização de uma cópia exige que as falhas e recuperações sejam serializadas com relação às transações. De acordo com o fato de poder acessar um objeto ou não, uma transação observa se uma falha ocorre depois de ter terminado ou antes de ter começado.

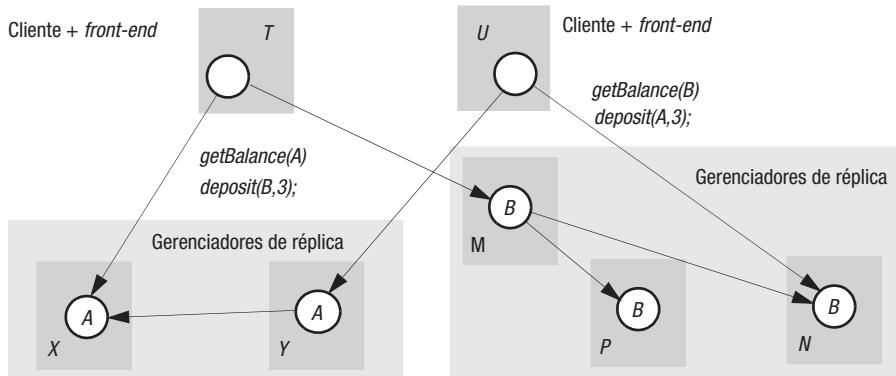


Figura 18.10 Cópias disponíveis.

A serialização de uma cópia não é obtida quando transações diferentes fazem observações de falha conflitantes.

Considere o caso da Figura 18.10, em que o gerenciador de réplica X falha imediatamente após T ter executado $getBalance$, e o gerenciador de réplica N falha imediatamente após U ter executado $getBalance$. Suponha que esses dois gerenciadores de réplica falhem antes de T e U terem executado suas operações $deposit$. Isso implica que a operação $deposit$ de T será executada nos gerenciadores de réplica M e P , e que a operação $deposit$ de U será executada no gerenciador de réplica Y . Infelizmente, o controle de concorrência de A no gerenciador de réplica X não impede que a transação U atualize A no gerenciador de réplica Y . Nem o controle de concorrência de B no gerenciador de réplica N impede que a transação T atualize B nos gerenciadores de réplica M e P .

Isso se contrapõe ao requisito da capacidade de serialização de uma cópia. Se essas operações fossem executadas em cópias únicas dos objetos, elas seriam serializadas, com a transação T antes de U , ou com a transação U antes de T . Isso garante que uma das transações lerá o valor configurado pela outra. O controle de concorrência local sobre cópias de objetos não é suficiente para garantir a capacidade de serialização de uma cópia no esquema de replicação de cópias disponíveis.

Quando as operações de *escrita* são direcionadas para todas as cópias disponíveis, o controle de concorrência local garante que as escritas conflitantes em um objeto sejam serializadas. Em contraste, uma operação de *leitura* de uma transação e uma operação de *escrita* de outra não afetam necessariamente a mesma cópia de um objeto. Portanto, o esquema exige controle de concorrência adicional para evitar que as dependências entre uma operação de *leitura* de uma transação e uma operação de *escrita* de outra transação formem um ciclo. Tais dependências não podem surgir se as falhas e recuperações de réplicas de objetos estiverem serializadas com relação às transações.

Validação local • Referimo-nos ao procedimento de controle de concorrência adicional como validação local. O procedimento de validação local é projetado para garantir que qualquer evento de falha ou recuperação não se manifeste durante o andamento de uma transação. Em nosso exemplo, como T leu de um objeto em X , a falha de X deve ser após T . Analogamente, como T observa a falha de N , quando ela tenta atualizar o objeto, a falha de N deve ser antes de T . Ou seja:

N falha $\rightarrow T$ lê o objeto A em X ; T escreve o objeto B em M e $P \rightarrow T$ é confirmada $\rightarrow X$ falha

Também pode ser argumentado, para a transação U , que:

X falha $\rightarrow U$ lê o objeto B em N ; U escreve o objeto A em $Y \rightarrow U$ é confirmada $\rightarrow N$ falha

O procedimento de validação local garante que não podem ocorrer duas dessas sequências incompatíveis. Antes que uma transação seja confirmada, ela verifica a existência de falhas (e recuperações) de gerenciadores de réplica dos objetos que acessou. No exemplo, a transação T verificaria se N ainda está indisponível e se X, M e P ainda estão disponíveis. Se esse fosse o caso, T poderia ser confirmada. Isso implica que X falha após T ser validada e antes de U ser validada. Em outras palavras, a validação de U ocorre após a validação de T . A validação de U falha porque N já falhou.

Quando uma transação tiver observado uma falha, o procedimento de validação local tentará se comunicar com os gerenciadores de réplica falhos para garantir que eles ainda não se recuperaram. A outra parte da validação local, que é testar se os gerenciadores de réplica não falharam desde que os objetos foram acessados, pode ser combinada com o protocolo de confirmação de duas fases.

Os algoritmos de cópias disponíveis não podem ser usados em ambientes nos quais gerenciadores de réplica que estão funcionando não conseguem se comunicar uns com os outros.

18.5.3 Particionamento da rede

Os esquemas de replicação precisam levar em conta a possibilidade de particionamento da rede. Um particionamento da rede separa um grupo de gerenciadores de réplica em dois ou mais subgrupos, de maneira tal que os membros de um subgrupo podem se comunicar entre si, mas não com membros de subgrupos diferentes. Por exemplo, na Figura 18.11, os gerenciadores de réplica que estão recebendo a requisição *deposit* não podem enviá-la para os gerenciadores de réplica que estão recebendo a requisição *withdraw*.

Os esquemas de replicação são projetados segundo a suposição de que o particionamento será reparado em algum momento. Portanto, os gerenciadores de réplica dentro de uma única partição devem garantir que as requisições que executarem durante o particionamento não tornarão o conjunto de réplicas inconsistente quando ele for reparado.

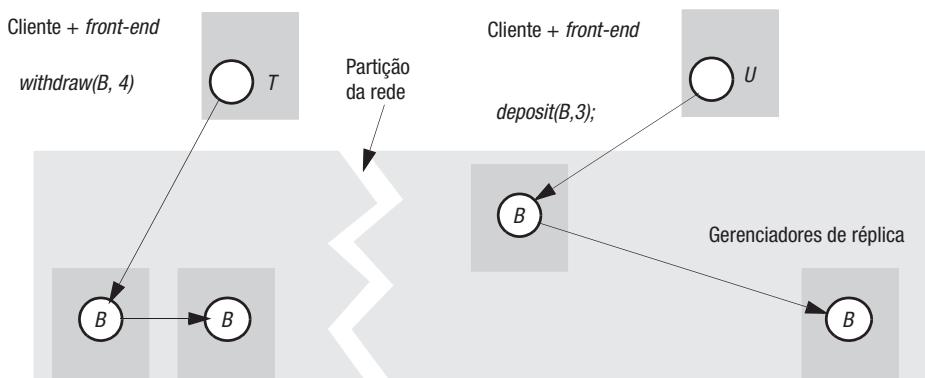


Figura 18.11 Particionamento da rede.

Davidson *et al.* [1985] discutem estratégias diferentes, as quais classificam como otimistas ou pessimistas com relação à probabilidade de ocorrência de inconsistências. Os esquemas otimistas não limitam a disponibilidade durante um particionamento, enquanto os esquemas pessimistas o fazem.

As estratégias otimistas permitem atualizações em todas as partições – isso pode levar a inconsistências entre as partições, as quais deverão ser resolvidas quando o particionamento for reparado. Um exemplo dessa estratégia é uma variante do algoritmo de cópias disponíveis na qual são permitidas atualizações nas partições e, após o particionamento ter sido reparado, as atualizações são validadas – as atualizações que violam o critério da capacidade de serialização de uma cópia são canceladas.

A estratégia pessimista limita a disponibilidade, mesmo quando não existem particionamentos, mas impede a ocorrência de inconsistências durante o particionamento. Quando um particionamento é reparado, basta atualizar as cópias dos objetos. A estratégia do consenso de *quorum* é pessimista. Ela permite atualizações em uma partição que tenha a maioria dos gerenciadores de réplica e propaga as atualizações para os outros gerenciadores de réplica quando o particionamento é reparado.

18.5.4 Cópias disponíveis com validação

O algoritmo de cópias disponíveis é aplicado dentro de cada partição. Essa estratégia otimista mantém o nível normal de disponibilidade para operações de *leitura*, mesmo durante um particionamento. Quando um particionamento é reparado, as transações possivelmente conflitantes que ocorreram nas partições separadas são validadas. Se a validação falhar, alguns passos devem ser dados para superar as inconsistências. Se não houvesse particionamento, uma de duas transações com operações conflitantes teria sido retardada ou cancelada. Infelizmente, como houve o particionamento, os pares de transações conflitantes puderam ser confirmados nas diferentes partições. A única opção após o evento é cancelar uma delas. Isso exige fazer alterações nos objetos e, em alguns casos, compensar os efeitos no mundo real, como lidar com contas bancárias sem fundo. A estratégia otimista só é possível em aplicações em que tais ações de compensação podem ser executadas.

Vetores de versão podem ser usados para validar conflitos entre pares de operações de *escrita*. Eles são usados no sistema de arquivos Coda e foram descritos na Seção 18.4.3. Essa estratégia não consegue detectar conflitos de *leitura-escrita*, mas funciona bem nos sistemas de arquivos em que as transações tendem a acessar um único arquivo e os conflitos de *leitura-escrita* não são importantes. Ela não é conveniente para aplicações como nosso exemplo de transações bancárias, no qual os conflitos de *leitura-escrita* são importantes.

Davidson [1984] usou *grafos de precedência* para detectar inconsistências entre partições. Cada divisão mantém um *log* dos objetos afetados pelas operações de *leitura* e de *escrita* das transações. Esse *log* é usado para construir um grafo de precedência cujos nós são transações e cujas arestas representam conflitos entre as operações de *leitura* e de *escrita* das transações. Tal grafo não conterá ciclos, pois foi aplicado controle de concorrência dentro da partição. O procedimento de validação pega os grafos de precedência das partições e adiciona arestas, representando conflitos entre transações nas diferentes partições. Se o grafo resultante contiver ciclos, a validação falhará.

18.5.5 Métodos de consenso de quorum

Uma maneira de evitar que transações em diferentes partições produzam resultados inconsistentes é estabelecer a regra de que as operações só podem ser executadas dentro de uma das partições. Como os gerenciadores de réplica nas diferentes partições não podem se comunicar, o subgrupo de gerenciadores de réplica dentro de cada partição deve ser capaz de

decidir independentemente se podem executar as operações. Um *quorum* é um subgrupo de gerenciadores de réplica cujo tamanho proporciona a ele o direito de executar operações. Por exemplo, se o critério fosse ter a maioria, um subgrupo que tivesse a maioria dos membros de um grupo formaria um *quorum*, pois nenhum outro subgrupo poderia ter a maioria.

Nos esquemas de replicação por consenso de *quorum*, uma operação de atualização sobre um objeto lógico pode ser concluída com êxito por um subgrupo de seu grupo de gerenciadores de réplica. Portanto, os outros membros do grupo terão cópias desatualizadas do objeto. Para determinar se as cópias estão atualizadas, podem ser usados números de versão ou carimbos de tempo. Se forem usadas versões, o estado inicial de um objeto é a primeira versão e, após cada alteração, temos uma nova versão. Cada cópia de um objeto tem um número de versão, mas apenas as cópias atualizadas têm o número de versão corrente, enquanto as cópias desatualizadas têm números de versão anteriores. As operações devem ser aplicadas apenas nas cópias com o número de versão corrente.

Gifford [1979] desenvolveu um esquema de replicação de arquivos no qual um número de votos é atribuído a cada cópia física de um único arquivo lógico em um gerenciador de réplica. Um voto pode ser considerado um peso relacionado ao desejo de usar uma cópia em particular. Cada operação de *leitura* deve primeiro obter *quorum* de leitura de R votos, antes de poder ler qualquer cópia atualizada, e cada operação de *escrita* deve obter *quorum* de escrita de W votos, antes de poder prosseguir com uma operação de atualização. R e W são configurados para um grupo de gerenciadores de réplica de modo que

$$W > \text{metade do total de votos}$$

$$R + W > \text{número total de votos do grupo}$$

Isso garante que qualquer par, consistindo em um *quorum* de leitura e um *quorum* de escrita ou dois *quora* de escrita, deve conter cópias comuns. Portanto, se houver um particionamento, não será possível executar operações conflitantes na mesma cópia, mas em partições diferentes.

Para executar uma operação de *leitura*, reúne-se um *quorum* de leitura fazendo-se perguntas sobre número de versões suficientes para encontrar um conjunto de cópias cuja soma de votos não seja menor do que R . Nem todas essas cópias precisam estar atualizadas. Como cada *quorum* de leitura se sobrepõe a cada *quorum* de escrita, é certo que todo *quorum* de leitura inclui pelo menos uma cópia corrente. A operação de leitura pode ser aplicada em qualquer cópia atualizada.

Para executar uma operação de *escrita*, reúne-se um *quorum* de escrita fazendo-se perguntas sobre número de versão suficientes para encontrar um conjunto de gerenciadores de réplica com cópias atualizadas cuja soma de votos não seja menor do que W . Se houver cópias insuficientes atualizadas, um arquivo não corrente será substituído por uma cópia do arquivo corrente para permitir que o *quorum* seja estabelecido. As atualizações especificadas na operação de *escrita* são, então, aplicadas em cada gerenciador de réplica no *quorum* de escrita, o número de versão é incrementado e a conclusão da escrita é informada ao cliente.

Então, os arquivos nos gerenciadores de réplica disponíveis restantes são atualizados, executando a operação de *escrita* como uma tarefa de segundo plano (*background*). Todo gerenciador de réplica cuja cópia do arquivo tenha um número de versão mais antigo do que aquele usado pelo *quorum* de escrita o atualiza, substituindo o arquivo inteiro por uma cópia obtida de um gerenciador de réplica que esteja atualizado.

No esquema de replicação de Gifford, pode ser usado travamento de leitura-escrita de duas fases para controle de concorrência. A pergunta preliminar sobre o número de versão para obter *quorum* de leitura, R , faz com que travas de leitura sejam configuradas em cada gerenciador de réplica contatado. Quando uma operação de *escrita* é aplicada no *qu-*

orum de escrita, W , uma trava de escrita é configurada em cada gerenciador de réplica envolvido. (As travas são aplicadas com a mesma granularidade dos números de versão.) As travas garantem a capacidade de serialização de uma cópia, pois todo *quorum* de leitura se sobrepõe a qualquer *quorum* de escrita e quaisquer dois *quora* de escrita se sobrepõem.

Capacidade de configuração de grupos de gerenciadores de réplica • Uma propriedade importante do algoritmo de votação ponderada é que grupos de gerenciadores de réplica podem ser configurados de forma a fornecer características de desempenho ou confiabilidade diferentes. Uma vez que a confiabilidade e o desempenho geral de um grupo de gerenciadores de réplica forem estabelecidos por sua configuração de votação, a confiabilidade e o desempenho das operações de *escrita* podem ser aumentados por meio da diminuição de W e, analogamente, das operações de *leitura*, pela diminuição de R .

O algoritmo também pode permitir o uso de cópias de arquivos que estão em discos locais nos computadores clientes, assim como das que estão em servidores de arquivo. As cópias dos arquivos nos computadores clientes são consideradas representantes mais fracas e sempre recebem zero votos. Isso garante que elas não sejam incluídas em nenhum *quorum*. Uma operação de *leitura* pode ser executada em qualquer cópia atualizada, uma vez que um *quorum* de leitura tenha sido obtido. Portanto, uma operação de *leitura* pode ser executada na cópia local do arquivo, caso esteja atualizada. As representantes fracas podem ser usados para acelerar operações de *leitura*.

Um exemplo de Gifford • Gifford fornece três exemplos mostrando a gama de propriedades que podem ser obtidas pela distribuição de pesos para os vários gerenciadores de réplica de um grupo e pela atribuição apropriada de R e W . Reproduziremos agora os exemplos de Gifford, os quais são baseados na tabela a seguir. As probabilidades de bloqueio fornecem uma indicação da probabilidade de que um *quorum* não possa ser obtido quando é feita uma requisição de *leitura* ou de *escrita*. Elas são calculadas supondo que existe uma probabilidade de 0,01 de que qualquer gerenciador de réplica único estará indisponível no momento de uma requisição.

O Exemplo 1 é configurado para um arquivo com uma relação leitura-escrita alta, em uma aplicação com vários representantes fracos e um único gerenciador de réplica. A replicação é usada para melhorar o desempenho do sistema e não a confiabilidade. Existe um gerenciador de réplica na rede local, que pode ser acessado em 75 milissegundos. Dois clientes optaram por terem representantes fracos em seus discos locais, os quais eles podem acessar em 65 milissegundos, resultando em uma latência mais baixa e em menos tráfego de rede.

O Exemplo 2 é configurado para um arquivo com relação leitura-escrita moderada, que é acessado a partir de uma rede local. O gerenciador de réplica na rede local recebe dois votos e os gerenciadores de réplica nas redes remotas recebem um voto cada um. As leituras podem ser atendidas a partir do gerenciador de réplica local, mas as escritas precisam acessar o gerenciador de réplica local e um gerenciador de réplica remoto. O arquivo permanecerá disponível no modo somente para leitura, caso o gerenciador de réplica local falhe. Os clientes poderiam criar representantes fracos locais para obterem uma latência de leitura menor.

O Exemplo 3 é configurado para um arquivo com uma taxa de leitura-escrita muito alta, como em um diretório de sistema em um ambiente com três gerenciadores de réplica. Os clientes podem ler a partir de qualquer gerenciador de réplica e a probabilidade de o arquivo estar indisponível é pequena. As atualizações devem ser aplicadas em todas as cópias. Mais uma vez, os clientes poderiam criar representantes fracos em suas máquinas locais para obterem latências de leitura menores.

A principal desvantagem do consenso de *quorum* é que o desempenho das operações de *leitura* é degradado pela necessidade de reunir *quorum* de leitura de R gerenciadores de réplica.

		<i>Exemplo 1</i>	<i>Exemplo 2</i>	<i>Exemplo 3</i>
<i>Latência (milissegundos)</i>	Réplica 1	75	75	75
	Réplica 2	65	100	750
	Réplica 3	65	750	750
<i>Configuração de votação</i>	Réplica 1	1	2	1
	Réplica 2	0	1	1
	Réplica 3	0	1	1
<i>Tamanhos de quorum</i>	<i>R</i>	1	2	1
	<i>W</i>	1	3	3

Desempenho derivado do conjunto de arquivos:

<i>Leitura</i>	Latência	65	75	75
	Probabilidade de bloqueio	0,01	0,0002	0,000001
<i>Escrita</i>	Latência	75	100	750
	Probabilidade de bloqueio	0,01	0,0101	0,03

Herlihy [1986] propôs uma ampliação do método do consenso de *quorum* para tipos de dados abstratos. Esse método permite que a semântica das operações seja levada em conta e, assim, aumente a disponibilidade dos objetos. O método de Herlihy usa carimbos de tempo, em vez de números de versão. Isso tem a vantagem de não haver necessidade de fazer perguntas sobre o número da versão para obter um novo número, antes de executar uma operação de escrita. A principal vantagem reivindicada por Herlihy é que o uso de conhecimento semântico pode aumentar o número de escolhas de um *quorum*.

Consenso de quorum no Dynamo • O Dynamo usa uma estratégia do tipo *quorum* para manter a consistência entre as réplicas. Assim como no esquema de Gifford, as operações de leitura e escrita devem usar nós *R* e *W*, respectivamente, e $R+W \geq N$. No Dynamo, *N* é o número de nós com réplicas. Os valores de *W* e *R* afetam a disponibilidade, a durabilidade e a consistência. DeCandia *et al.* [2007] afirmam que uma configuração comum no Dynamo tem $[N, R, W] = [3, 2, 2]$.

No caso de particionamento, o *quorum* de Gifford pode operar somente em uma partição de “maioria”. Contudo, o Dynamo usa um “*quorum relaxado*” que envolve *N* nós, nos quais as réplicas podem ser armazenadas em nós substitutos que passarão os valores quando o nó pretendido se recuperar.

18.5.6 Algoritmo de particionamento virtual

Este algoritmo, que foi proposto por El Abbadi *et al.* [1985], combina a estratégia do consenso de *quorum* com o algoritmo de cópias disponíveis. O consenso de *quorum* funciona corretamente na presença de partções, mas o algoritmo das cópias disponíveis é menos dispendioso para operações de *leitura*. Uma *partição virtual* é uma abstração de uma partição real e contém um conjunto de gerenciadores de réplica. Note que o termo *partição de rede* se refere à barreira que divide os gerenciadores de réplica em várias partes, enquanto o termo *partição virtual* se refere às partes em si. Embora não sejam

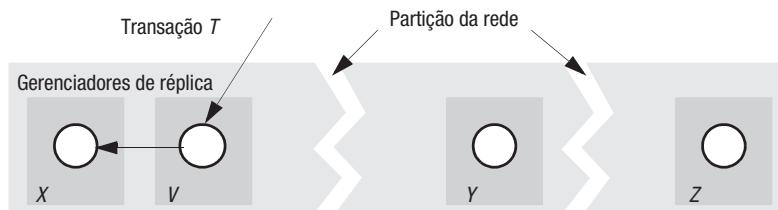


Figura 18.12 Duas partições de rede.

conectadas com comunicação *multicast*, as partições virtuais são semelhantes aos modos de visualização de grupo, que foram apresentados na Seção 18.2.2. Uma transação pode operar em uma partição virtual se contiver gerenciadores de réplica suficientes para ter *quorum* de leitura e *quorum* de escrita para os objetos acessados. Nesse caso, a transação usa o algoritmo de cópias disponíveis. Isso tem a vantagem de que as operações de *leitura* só precisam acessar uma única cópia de um objeto e podem melhorar o desempenho, escolhendo a cópia mais próxima. Se um gerenciador de réplica falhar e a partição virtual mudar durante uma transação, a transação será cancelada. Isso garante a capacidade de serialização de uma cópia das transações, pois todas as transações que sobrevivem veem as falhas e recuperações dos gerenciadores de réplica na mesma ordem.

Quando um membro de uma partição virtual detecta que não pode acessar um dos outros membros – por exemplo, quando uma operação de *escrita* não é reconhecida –, ele tenta criar uma nova partição virtual com um modo de visualização, para obter uma partição virtual com *quora* de leitura e de escrita.

Suponha, por exemplo, que tenhamos quatro gerenciadores de réplica V , X , Y e Z , cada um dos quais com um voto, e que os *quora* de leitura e de escrita sejam $R = 2$ e $W = 3$. Inicialmente, todos os gerenciadores podem contatar uns aos outros. Desde que permaneçam em contato, eles podem usar o algoritmo de cópias disponíveis. Por exemplo, uma transação T consistindo em uma operação de *leitura*, seguida de uma operação de *escrita*, executará a *leitura* em um único gerenciador de réplica (por exemplo, V) e a operação de *escrita* em todos os quatro.

Suponha que a transação T comece executando sua *leitura* em V , em um momento em que V ainda está em contato com X , Y e Z . Agora, suponha que ocorra um particionamento da rede, como na Figura 18.12, na qual V e X estão em uma partição e Y e Z estão em partições diferentes. Então, quando a transação T tentar aplicar sua *escrita*, V notará que não pode contatar Y e Z .

Quando um gerenciador de réplica não consegue contatar os gerenciadores que conseguia anteriormente, ele continua tentando até poder criar uma nova partição virtual. Por exemplo, V continuará tentando contatar Y e Z até que um ou ambos respondam, como, por exemplo, na Figura 18.13, quando Y pode ser acessado. O grupo de gerenciadores de réplica V , X e Y compreende uma partição virtual, pois eles são suficientes para formar *quora* de leitura e de escrita.

Quando uma nova partição virtual é criada, durante uma transação que executou uma operação em um dos gerenciadores de réplica (como a transação T), a transação deve ser cancelada. Além disso, as réplicas dentro de uma nova partição virtual devem ser atualizadas, copiando-as de outras réplicas. Números de versão podem ser usados, como no algoritmo de Gifford, para determinar quais cópias foram atualizadas. É fundamental que todas as réplicas sejam atualizadas, pois as operações de *leitura* são executadas em uma única réplica.

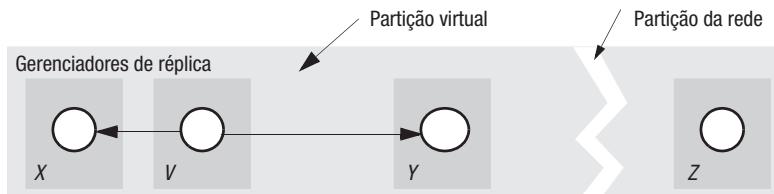


Figura 18.13 Partição virtual.

Implementação de partições virtuais • Uma partição virtual tem um tempo de criação, um conjunto de membros em potencial e um conjunto de membros reais. Os tempos de criação são carimbos de tempo lógicos. Os membros reais de uma partição virtual em particular têm a mesma ideia quanto ao seu tempo de criação e à sua participação como membro (um *modo de visualização* compartilhado dos gerenciadores de réplica com os quais eles podem se comunicar). Por exemplo, na Figura 18.13, os membros em potencial são V , X , Y e Z e os membros reais são V , X e Y .

A criação de uma nova partição virtual é obtida por um protocolo cooperativo, executado pelos membros em potencial que podem ser acessados pelos gerenciadores de réplica que a iniciaram. Vários gerenciadores de réplica podem tentar criar uma nova partição virtual simultaneamente. Por exemplo, suponha que os gerenciadores de réplica Y e Z mostrados na Figura 18.12 continuem tentando contatar os outros e, após algum tempo, a partição de rede é parcialmente reparada, de modo que Y não pode se comunicar com Z , mas os dois grupos, V, X, Y e V, X, Z , podem se comunicar entre si. Então, existe o perigo da criação de duas partições virtuais sobrepostas, como V_1 e V_2 mostradas na Figura 18.14.

Considere o efeito de executar transações diferentes nas duas partições virtuais. A operação de *leitura* da transação em V, X, Y poderia ser aplicada no gerenciador de réplica Y , no caso em que sua trava de leitura não entraria em conflito com as travas de escritas configuradas por uma operação de *escrita* e uma transação na outra partição virtual. As partições virtuais sobrepostas são contrárias à capacidade de serialização de uma cópia.

O objetivo do protocolo é criar novas partições virtuais consistentemente, mesmo que ocorram partições reais durante o processo. O protocolo para criar uma nova partição virtual tem duas fases, como mostra a Figura 18.15.

Um gerenciador de réplica que responde *Sim* na Fase 1 não pertence a uma partição virtual até receber a mensagem de *confirmação* correspondente na Fase 2.

Em nosso exemplo anterior, os gerenciadores de réplica Y e Z mostrados na Figura 18.12 tentam, cada um, criar uma partição virtual, e aquela que tiver o carimbo de tempo lógico mais alto será usado no final.

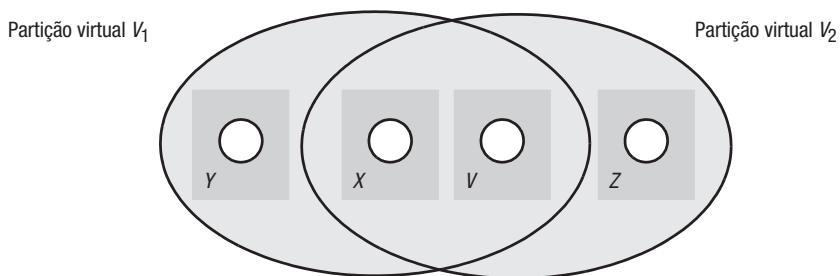


Figura 18.14 Duas partições virtuais sobrepostas.

Fase 1:

- O iniciador envia uma requisição de *Join* para cada membro em potencial. O argumento de *Join* é um carimbo de tempo lógico proposto para a nova partição virtual.
- Quando um gerenciador de réplica recebe uma requisição de *Join*, ele compara o carimbo de tempo lógico proposto com o de sua partição virtual corrente.
 - Se o carimbo de tempo lógico proposto for maior, ele concorda em juntar e responde *Sim*;
 - Se for menor, ele recusa a se juntar e responde *Não*.

Fase 2:

- Se o iniciador tiver recebido respostas *Sim* suficientes para ter *quora de leitura* e de *escrita*, ele pode completar a criação da nova partição virtual, enviando uma mensagem de *confirmação* para os *sites* que concordaram em se juntar. O carimbo de tempo da criação e a lista de membros reais são enviados como argumentos.
- Os gerenciadores de réplica que receberam a mensagem de *confirmação* entram na nova partição virtual e gravam seu carimbo de tempo de criação e sua lista de membros reais.

Figura 18.15 Criação de uma partição virtual.

Esse é um método eficiente quando o particionamento não é uma ocorrência comum. Cada transação usa o algoritmo de cópias disponíveis dentro de uma partição virtual.

18.6 Resumo

A replicação de objetos é uma maneira importante de obter serviços com bom desempenho, alta disponibilidade e tolerância a falhas em um sistema distribuído. Descrevemos arquiteturas para serviços nas quais os gerenciadores de réplica contêm réplicas dos objetos e nas quais os *front-ends* tornam essa replicação transparente. Os clientes, os *front-ends* e os gerenciadores de réplica podem ser processos separados ou não.

Este capítulo começou descrevendo um modelo de sistema no qual cada objeto lógico é implementado por um conjunto de réplicas físicas. Frequentemente, as atualizações nessas réplicas podem ser feitas convenientemente pela comunicação em grupo. Expandimos nossa narrativa sobre a comunicação em grupo para incluir modos de visualização de grupo e comunicação de modo de visualização síncrono.

Definimos a capacidade de linearização e a consistência sequencial como critérios da correção para serviços tolerantes a falhas. Esses critérios expressam como os serviços devem fornecer o equivalente a uma única imagem do conjunto de objetos lógicos, mesmo que esses objetos sejam replicados. Praticamente falando, o mais significativo dos critérios é a consistência sequencial.

Na replicação passiva (*backup* primário), a tolerância a falhas é obtida pelo direcionamento de todas as requisições por intermédio de um gerenciador de réplica distinto e fazendo-se com que um gerenciador de réplica de *backup* assuma, caso o primeiro falhe. Na replicação ativa, todos os gerenciadores de réplica processam todas as requisições independentemente. As duas formas de replicação podem ser convenientemente implementadas usando-se comunicação em grupo.

Em seguida, consideramos os serviços com alta disponibilidade. As arquiteturas de fofoca e Bayou permitem que os clientes façam atualizações em réplicas locais enquanto estão particionados. Em cada sistema, os gerenciadores de réplica trocam atualizações

entre si, quando são reconectados. A arquitetura de fofoca fornece alta disponibilidade à custa de uma consistência causal relaxada. A arquitetura Bayou fornece garantias de consistência eventual mais fortes, empregando detecção automática de conflitos e a técnica da transformação operacional para solucionar conflitos. O Coda é um sistema de arquivos de alta disponibilidade que usa vetores de versão para detectar atualizações potencialmente conflitantes.

Finalmente, consideramos o desempenho das transações com dados replicados. Para esse caso, existem tanto arquiteturas de *backup* primário como arquiteturas nas quais os *front-ends* podem se comunicar com qualquer gerenciador de réplica. Discutimos como os sistemas transacionais permitem falhas do gerenciador de réplica e partitionamentos da rede. As técnicas de cópias disponíveis, consenso de *quorum* e partições virtuais permitem o progresso das operações dentro das transações, mesmo em algumas circunstâncias nas quais nem todas as réplicas estão acessíveis.

Exercícios

- 18.1 Três computadores juntos fornecem um serviço replicado. Os fabricantes dizem que cada computador tem um tempo médio entre falhas de cinco dias; normalmente, uma falha demora quatro horas para ser corrigida. Qual é a disponibilidade do serviço replicado? *página 766*
- 18.2 Explique por que um servidor *multithreaded* não poderia ser qualificado como uma máquina de estado. *página 768*
- 18.3 Em um jogo multiusuário, os jogadores movem figuras em uma cena comum. O estado do jogo é replicado nas estações de trabalho dos jogadores e em um servidor, o qual contém serviços controlando o jogo global, como a detecção de colisão. As atualizações são enviadas por *multicast* para todas as réplicas.
 - (i) As figuras podem lançar projéteis umas nas outras e um acerto debilita o infeliz receptor por um tempo limitado. Que tipo de ordem de atualização é exigido aqui? Dica: considere os eventos de disparo, colisão e reanimação.
 - (ii) O jogo incorpora dispositivos mágicos que podem ser pegos por um jogador para ajudá-lo. Que tipo de ordenação deve ser aplicado na operação “pegar dispositivo”? *página 770*
- 18.4 Um roteador que separa o processo *p* de dois outros, *q* e *r*, falha imediatamente após *p* iniciar o *multicast* da mensagem *m*. Se o sistema de comunicação em grupo é de modo de visualização síncrono, explique o que acontece com *p* em seguida. *página 773*
- 18.5 Você recebe um sistema de comunicação em grupo com uma operação de *multicast* totalmente ordenada e um detector de falha. É possível construir comunicação em grupo com modo de visualização síncrono a partir apenas desses componentes? *página 773*
- 18.6 Uma operação de *multicast* com *ordenação de sincronismo* é aquela cuja semântica de ordem de distribuição é igual a da distribuição de modos de visualização em um sistema de comunicação em grupo com modo de visualização síncrono. Em um serviço *thingumajig*, as operações sobre *thingumajigs* têm ordem causal. O serviço suporta listas de usuários capazes de efetuar operações sobre cada *thingumajig* em particular. Explique por que a remoção de um usuário de uma lista deve ser uma operação com ordenação de sincronismo. *página 773*
- 18.7 Qual é o problema de consistência levantado pela transferência de estado? *página 774*
- 18.8 Uma operação *X* sobre um objeto *o* faz com que *o* invoque uma operação sobre outro objeto *o'*. Agora, é proposto que ela replique *o*, mas não *o'*. Explique a dificuldade que isso acarreta com relação às invocações sobre *o'* e sugira uma solução. *página 773*

- 18.9 Explique a diferença entre capacidade de linearização e consistência sequencial, e por que, em geral, é mais prático implementar esta última. *página 777*
- 18.10 Explique por que permitir que gerenciadores de *backup* processem operações de *leitura* leva a execuções com consistência sequencial, em vez de execuções com capacidade de linearização em um sistema de replicação passiva. *página 780*
- 18.11 A arquitetura de fofoca poderia ser usada para um jogo de computador distribuído, conforme descrito no Exercício 18.3? *página 783*
- 18.12 Na arquitetura de fofoca, por que um gerenciador de réplica precisa manter uma indicação de tempo da réplica e uma indicação de tempo do “valor”? *página 786*
- 18.13 Em um sistema de fofoca, um *front-end* tem o carimbo de tempo vetorial (3, 5, 7), representando os dados que recebeu dos membros de um grupo de três gerenciadores de réplica. Os três gerenciadores de réplica têm os carimbos de tempo vetoriais (5, 2, 8), (4, 5, 6) e (4, 5, 8) respectivamente. Qual gerenciador (ou gerenciadores) de réplica poderia atender imediatamente a uma consulta do *front-end* e qual é o carimbo de tempo resultante do *front-end*? Qual poderia incorporar uma atualização do *front-end* imediatamente? *página 788*
- 18.14 Explique por que tornar alguns gerenciadores de réplica somente de leitura pode melhorar o desempenho de um sistema de fofoca. *página 792*
- 18.15 Escreva um pseudocódigo para verificações de dependência e procedimentos de integração (conforme os usados no Bayou), convenientes para uma aplicação simples de marcação de quartos. *página 793*
- 18.16 No sistema de arquivos Coda, por que às vezes é necessário que os usuários intervenham manualmente no processo de atualização das cópias de um arquivo em vários servidores? *página 800*
- 18.17 Projete um esquema para integrar duas réplicas de um diretório de sistema de arquivos que passaram por atualizações separadas durante a operação desconectada. Use a estratégia da transformação operacional do Bayou ou forneça uma solução para o Coda. *página 801*
- 18.18 A replicação de cópias disponíveis é aplicada nos itens de dados *A* e *B*, com réplicas A_x, A_y e B_m, B_n . As transações *T* e *U* são definidas como:
T: *Read(A); Write(B, 44)*. *U*: *Read(B); Write(A, 55)*.
Mostre uma interposição de *T* e *U*, supondo que são aplicadas travas de duas fases nas réplicas. Explique por que apenas as travas não podem garantir capacidade de serialização de uma cópia, caso uma das réplicas falhe durante o andamento de *T* e *U*. Explique, com referência a este exemplo, como a validação local garante a capacidade de serialização de uma cópia. *página 805*
- 18.19 A replicação de consenso de *quorum* de Gifford está em uso nos servidores *X*, *Y* e *Z*, todos os quais contêm réplicas dos itens de dados *A* e *B*. Os valores iniciais de todas as réplicas de *A* e *B* são 100 e os votos de *A* e *B* são 1 cada, em *X*, *Y* e *Z*. Além disso, $R = W = 2$ para *A* e para *B*. Um cliente lê o valor de *A* e depois o escreve em *B*.
- No momento em que o cliente executa essas operações, um particionamento separa os servidores *X* e *Y* do servidor *Z*. Descreva os *quora* obtidos e as operações que ocorrerão se o cliente puder acessar os servidores *X* e *Y*.
 - Descreva os *quora* obtidos e as operações que ocorrerão se o cliente puder acessar apenas o servidor *Z*.
 - O particionamento é reparado e, em seguida, outro particionamento ocorre, de modo que *X* e *Z* ficam separados de *Y*. Descreva os *quora* obtidos e as operações que ocorrerão se o cliente puder acessar os servidores *X* e *Z*. *página 810*

19

Computação Móvel e Ubíqua

- 19.1 Introdução
- 19.2 Associação
- 19.3 Interoperabilidade
- 19.4 Percepção e reconhecimento de contexto
- 19.5 Segurança e privacidade
- 19.6 Adaptabilidade
- 19.7 Estudo de caso: Cooltown
- 19.8 Resumo

Este capítulo estuda as áreas de computação móvel e ubíqua, as quais surgiram devido à miniaturização dos dispositivos e à conectividade sem fio. De modo geral, a computação móvel ocupa-se da exploração da conexão de equipamentos portáteis; a computação ubíqua diz respeito à exploração da integração cada vez maior dos dispositivos de computação com nosso mundo físico cotidiano.

Este capítulo apresenta um modelo de sistema comum que dá ênfase à volatilidade dos sistemas móveis e ubíquos: o conjunto de usuários, dispositivos e componentes de *software*, em qualquer ambiente, está sujeito a mudar frequentemente. Em seguida, o capítulo examina algumas das principais áreas de pesquisa que surgiram por causa da volatilidade e de suas bases físicas, incluindo como os componentes de *software* podem associar-se e interagir quando as entidades mudam, falham ou aparecem espontaneamente; como os sistemas são integrados no mundo físico, por meio da percepção e do reconhecimento de contexto; os problemas de segurança e privacidade que surgem nos sistemas voláteis e fisicamente integrados; e as técnicas para se adaptar à falta de recursos computacionais e de E/S dos dispositivos portáteis. O capítulo termina com um estudo de caso do projeto Cooltown, uma arquitetura orientada para usuários humanos, baseada na Web, para computação móvel e ubíqua.

19.1 Introdução

A computação móvel e ubíqua surgiu devido à miniaturização dos dispositivos e da conectividade sem fio. De modo geral, a computação móvel ocupa-se da exploração da conexão de dispositivos que se movimentam no mundo físico cotidiano; a computação ubíqua diz respeito à exploração da integração cada vez maior dos dispositivos de computação com nosso mundo físico. À medida que os equipamentos se tornam menores, fica mais fácil levá-los conosco ou vesti-los, e podemos incorporá-los em muitas partes do mundo físico – e não apenas no já comum *desktop* ou no *rack* de um servidor. E, à medida que a conectividade sem fio se torna predominante, podemos conectar melhor esses novos e pequenos dispositivos uns com os outros, com computadores pessoais e com servidores convencionais.

Este capítulo examina os aspectos da computação móvel (um assunto já mencionado no tratamento da operação desconectada do Capítulo 18) e da computação ubíqua. Ele aborda as propriedades comuns e as diferenças que compartilham com os sistemas distribuídos mais convencionais. Dado o progresso feito até agora, o capítulo fala mais sobre problemas em aberto do que sobre soluções.

Primeiramente, este capítulo descreve, em linhas gerais, os princípios da computação móvel e ubíqua e apresenta as subáreas conhecidas como computação acoplada ao corpo (*wearable*), de mão (*handheld*) e com reconhecimento de contexto (*context-aware computing*). Depois disso, ele descreve um modelo de sistema que abrange todas essas áreas e subáreas por meio de sua volatilidade – o conjunto de usuários, dispositivos e componentes de *software*, em determinado ambiente, é sujeito a mudanças frequentes. Em seguida, o capítulo examina algumas das principais áreas de pesquisa que surgiram por causa da volatilidade e de suas bases físicas, incluindo: como os componentes de *software* se associam e interagem uns com os outros quando as entidades se movem, falham ou aparecem espontaneamente nos ambientes; como os sistemas se tornam integrados com o mundo físico, por meio da percepção e do reconhecimento de contexto; os problemas da segurança e privacidade que surgem nos sistemas voláteis e fisicamente integrados; e as técnicas para se adaptar à falta de recursos computacionais e de E/S dos dispositivos portáteis. O capítulo termina com um estudo de caso do projeto Cooltown, uma arquitetura orientada para usuários humanos, baseada na Web, para computação móvel e ubíqua.

Computação móvel e de mão • A computação móvel surgiu como um paradigma no qual os usuários poderiam carregar seus computadores pessoais e manter certa conectividade com outras máquinas. Por volta de 1980, tornou-se possível construir computadores pessoais leves o suficiente para serem carregados e que podiam ser conectados a outros computadores por meio de linhas telefônicas, usando um modem. A evolução tecnológica levou a mais ou menos à mesma ideia, mas com funcionalidade e desempenho muito melhores: o equivalente atual é um *notebook*, um *netbook* ou, ainda, um *tablet*, cada um com várias formas de conectividade sem fio, incluindo as redes de celulares, WiFi e Bluetooth.

Um caminho diferente da evolução tecnológica levou à *computação de mão (handheld computing)*: o uso de aparelhos que cabem na mão, incluindo os telefones móveis “inteligentes” (*smartphones*), os assistentes digitais pessoais (PDAs, Personal Digital Assistants), e outros equipamentos mais especializados operados manualmente. Os *smartphones* e os PDAs são capazes de executar muitos tipos diferentes de aplicações; mas, comparados aos *notebooks*, têm menor tamanho e bateria de maior duração, possuem um poder de processamento correspondentemente limitado, uma tela menor e outras restrições de recursos. Cada vez mais, os fabricantes fornecem aos equipamentos manuais a mesma variedade de conectividade sem fio que os *notebooks* e seus equivalentes bem menores possuem.

Uma tendência interessante na computação de mão tem sido a pouca distinção entre PDAs, telefones móveis e equipamentos de mão de finalidade específica, como câmeras digitais e unidades de navegação baseadas em GPS. Os *smartphones* têm funcionalidade de computação do tipo do PDA, em virtude de executarem sistemas operacionais como o Symbian da Nokia e de outros fabricantes, o Android do Google, o iOS da Apple ou o Windows Phone 7 da Microsoft. Eles têm câmeras embutidas e incorporam ou podem ser equipados com outros tipos de acessórios especializados, tornado-se uma alternativa aos equipamentos de mão de finalidade específica. Por exemplo, um usuário pode ler um código de barras com a câmera de um *smartphone* a fim de obter informações para comparação de preços. Frequentemente, os *smartphones* possuem unidades de GPS internas para navegação e outros propósitos de localização específicos.

Stojmenovic [2002] aborda os princípios e protocolos da comunicação sem fio, incluindo uma abordagem dos dois maiores problemas da camada de rede que precisam ser resolvidos para os sistemas estudados neste capítulo. O primeiro problema é como fornecer conectividade contínua para dispositivos móveis que entram e saem do alcance das *estações de base*, que são os componentes de infraestrutura que fornecem regiões de cobertura sem fio. O segundo problema é como permitir que conjuntos de dispositivos se comuniquem sem fio uns com os outros, em lugares onde não existe infraestrutura (veja o breve tratamento das *redes ad hoc*, na Seção 19.4.2). Os dois problemas surgem porque, frequentemente, a conectividade sem fio direta não está disponível entre quaisquer dois dispositivos. Então, a comunicação precisa ser obtida por meio de vários segmentos de rede com ou sem fio. Dois fatores principais levam a essa cobertura sem fio subdividida. Primeiro, quanto maior o alcance de uma rede sem fio, mais dispositivos competirão por sua largura de banda limitada. Segundo, considerações sobre a energia se aplicam: a energia necessária para transmitir um sinal sem fio é proporcional ao quadrado de seu alcance; mas muitos dos dispositivos que consideraremos têm capacidade de energia limitada.

Computação ubíqua • Mark Weiser cunhou o termo *computação ubíqua* em 1988 [Weiser 1991]. Às vezes, a computação ubíqua também é conhecida como *computação pervasiva* e os dois termos normalmente são considerados sinônimos. “Ubíquo” significa “em toda parte”. Weiser percebeu a predominância cada vez maior dos dispositivos de computação levando a mudanças revolucionárias na maneira como usávamos os computadores.

Primeiro, cada pessoa no mundo utilizaria muitos computadores. Podemos comparar isso à revolução da computação pessoal anterior a essa, que viu um computador para cada pessoa. Embora pareça simples, essa mudança teve um efeito dramático sobre a maneira como usamos os computadores, em comparação à era anterior dos computadores de grande porte, quando havia apenas um computador para muitas pessoas. A ideia de Weiser de “uma pessoa, muitos computadores” significa algo muito diferente da situação comum, na qual cada um de nós tem vários computadores mais ou menos parecidos – um no trabalho, um em casa, um *notebook* e, talvez, um *smartphone* que levamos conosco. Em vez disso, na computação ubíqua, os computadores se multiplicam na forma e na função e não apenas no número, para atender a diferentes tarefas.

Por exemplo, suponha que todos os meios de exibição e escrita fixos de uma sala – quadros, livros, folhas de papel, notas adesivas, etc. – fossem substituídos por dezenas ou centenas de computadores individuais com telas eletrônicas. Livros já aparecem em forma eletrônica, visíveis em equipamentos que permitem aos leitores ler e pesquisar seu texto, procurar o significado de palavras, buscar ideias relacionadas na Web e experimentar conteúdo multimídia vinculado. Agora, suponha que incorporemos funcionalidade de computação em todas as ferramentas de escrita. Por exemplo, as canetas e os marcadores se tornariam capazes de armazenar o que o usuário escreveu e desenhou e de reunir, copiar e mover conteúdo

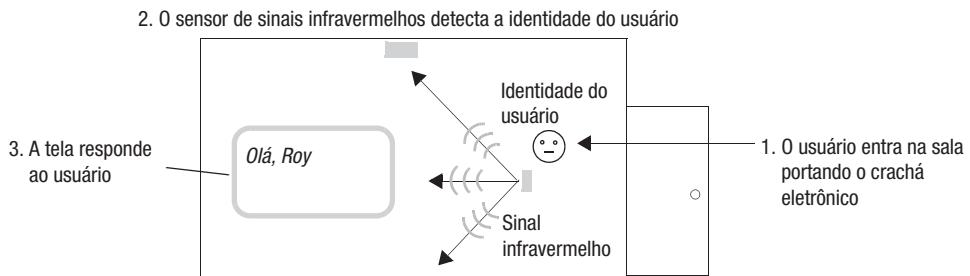


Figura 19.1 Uma sala respondendo a um usuário portando um crachá ativo.

multimídia entre os muitos computadores situados nas imediações. Esse cenário levanta questões de capacidade de utilização e econômicas e toca apenas em uma pequena parte de nossas vidas, mas nos dá uma ideia do que a “computação por toda parte” poderia ser.

A segunda mudança que Weiser previu foi que os computadores “desapareceriam” – que eles “se incorporariam em utensílios e objetos do dia a dia, até se tornarem indistinguíveis”. Essa é principalmente uma noção psicológica, comparável ao modo como as pessoas aceitam normalmente a mobília de uma casa e mal a notam. Ela reflete a ideia de que a computação estará incorporada ao que consideramos itens do cotidiano – aqueles que normalmente não achamos que tenham recursos computacionais, nada diferente do que pensamos a respeito das máquinas de lavar, ou dos veículos como “equipamentos de computação”, mesmo tendo o controle de microprocessadores incorporados – cerca de 100 microprocessadores, no caso de alguns carros.

Embora a invisibilidade de certos dispositivos seja apropriada – nos casos como os sistemas de computadores incorporados em um carro – isso não vale para todos os equipamentos que vamos considerar, particularmente para os dispositivos que os usuários móveis normalmente carregam. Por exemplo, na época da escrita deste livro, os telefones celulares eram os dispositivos móveis de maior penetração no mercado, mas sua capacidade computacional não era visível à maioria dos usuários.

Computação acoplada ao corpo (wearable computing) • Os usuários levam equipamentos de computação acoplados a si mesmos, presos no tecido de suas roupas ou dentro dele, ou transportados em seus próprios corpos, como relógios, jóias ou óculos. Ao contrário dos equipamentos de mão mencionados anteriormente, esses dispositivos frequentemente funcionam sem que o usuário precise manipulá-los. Normalmente, eles possuem funcionalidade especializada. Um exemplo simples é o “crachá ativo”, um pequeno equipamento de computação preso ao usuário, que transmite regularmente a identidade do crachá (associada a um usuário) por intermédio de um transmissor de sinal infravermelho [Want *et al.* 1992; Harter e Hopper 1994]. A ideia do crachá é que os dispositivos presentes no ambiente respondam às suas transmissões e, assim, respondam à presença de um usuário; as transmissões de sinal infravermelho têm um alcance limitado e, portanto, serão captadas apenas se o usuário estiver nas proximidades. Por exemplo, uma tela eletrônica poderia ser adaptada à presença de um usuário, personalizando-se seu comportamento de acordo com as preferências desse usuário, como a cor do desenho e a espessura de linha padrão (Figura 19.1). Uma sala poderia ser adaptada para ajustes de ar condicionado e iluminação, de acordo com a pessoa que estivesse dentro dela.

Computação com reconhecimento de contexto (context-aware computing) • O crachá ativo – ou melhor, as reações de outros dispositivos à sua presença – exemplifica a computação

com reconhecimento de contexto, que é uma subárea importante da computação móvel e ubíqua. É onde os sistemas de computadores adaptam seu comportamento automaticamente, de acordo com as circunstâncias físicas. Essas circunstâncias podem, em princípio, ser algo fisicamente medido ou detectado, como a presença de um usuário, a hora do dia ou as condições atmosféricas. Algumas das condições dependentes são relativamente simples de determinar, como o fato de ser noite (a partir da hora, do dia do ano e da posição geográfica). No entanto, outras exigem processamento sofisticado para sua detecção. Por exemplo, considere um telefone celular com reconhecimento de contexto, que só deve tocar quando for apropriado. Em particular, ele deve trocar automaticamente para o modo “vibrar”, em vez de “tocar”, quando estiver no cinema. Porém, não é simples detectar que o usuário está assistindo a um filme dentro de um cinema e não parado no saguão, dadas às imprecisões das medidas do sensor de posição. A Seção 19.4 examinará o contexto com mais detalhes.

19.1.1 Sistemas voláteis

Do ponto de vista dos sistemas distribuídos, não há uma diferença básica entre computação móvel e ubíqua, ou as subáreas que apresentamos (ou, na verdade, as subáreas que omitimos, como a computação tangível [Ishii e Ullmer 1997] e a realidade ampliada, exemplificada pela escrivaninha digital de Wellner [Wellner 1991]). Nesta subseção, mostraremos um modelo do que chamamos de *sistemas voláteis*, que abrangem os recursos dos sistemas distribuídos básicos de todos eles.

Chamamos os sistemas descritos neste capítulo de voláteis porque, ao contrário da maioria dos sistemas descritos nas outras partes deste livro, as mudanças são comuns, em vez de serem exceção. O conjunto de usuários, *hardware* e *software* nos sistemas móveis e ubíquos é altamente dinâmico e muda de maneira imprevisível. Outra palavra que às vezes usaremos para esses sistemas é *espontâneo*, que aparece na literatura, na frase *interligação em rede espontânea*. As formas relevantes de volatilidade incluem:

- falhas de dispositivos e enlaces de comunicação;
- mudanças nas características da comunicação, como a largura de banda;
- a criação e destruição de *associações* – relacionamentos de comunicação lógicos – entre os componentes de *software* residentes nos dispositivos.

Aqui, o termo “componente” abrange qualquer unidade de *software*, como objetos ou processos, independentemente de interagir como cliente, servidor ou *peer*.

O Capítulo 18 já mostrou as maneiras de lidar com algumas dessas mudanças, a saber, falhas de processamento e operação desconectada. Contudo, as soluções mostradas foram dadas considerando-se as falhas de processamento e comunicação como sendo a exceção, e não a regra, e levando em conta a existência de recursos de processamento redundantes. Os sistemas voláteis não apenas violam essas suposições, como também acrescentam ainda mais fenômenos de alteração, notadamente às mudanças frequentes nas associações entre os componentes.

É importante esclarecermos um possível mal-entendido, antes de prosseguirmos. A volatilidade não é uma propriedade *que define* os sistemas móveis e ubíquos: existem outros tipos de sistema que demonstram uma ou mais formas de volatilidade, mas que não são móveis, nem ubíquos. Um bom exemplo é a computação *peer-to-peer*, como os aplicativos de compartilhamento de arquivo (Capítulo 10), na qual o conjunto de processos participantes e as associações entre eles estão sujeitos a altas taxas de mudança. O que é diferente na computação móvel e ubíqua é que elas exibem *todas* as formas anteriores de volatilidade, devido à maneira como são integradas com o mundo físico. Temos muito a dizer sobre essa integração física e como ela causa volatilidade, mas a integração física

em si não é uma propriedade dos sistemas distribuídos, enquanto a volatilidade, sim. Daí vem o termo que adotamos.

No restante desta seção, descreveremos os espaços inteligentes (*smart spaces*), os ambientes dentro dos quais os sistemas voláteis subsistem; em seguida, caracterizaremos os dispositivos móveis e ubíquos, sua conectividade física e lógica e as consequências da menor confiança e privacidade.

Espaços inteligentes (smart spaces) • Os espaços físicos são importantes, pois formam a base da computação móvel e ubíqua. Ocorre mobilidade entre os espaços físicos; a computação ubíqua é incorporada em espaços físicos. Um *espaço inteligente* é qualquer local físico com serviços incorporados – isto é, serviços fornecidos apenas, ou principalmente, dentro desse espaço físico. É possível introduzir dispositivos de computação ao acaso, onde não existe nenhuma infraestrutura, para executar uma aplicação, como o monitoramento ambiental. Contudo, normalmente, os dispositivos móveis e sistemas ubíquos são integrados a um ambiente computacional já existente, como uma sala, um prédio, um quarteirão ou a um veículo, melhorando a capacidade computacional do meio. Nesses casos, o espaço inteligente normalmente contém uma infraestrutura de computação relativamente estável, a qual pode incluir computadores servidores convencionais, dispositivos como impressoras e telas, sensores e uma infraestrutura de rede sem fio, incluindo uma conexão com a Internet.

Existem vários tipos de movimentação ou “aparecimento e desaparecimento” que podem ocorrer nos espaços inteligentes. Primeiro, existe a *mobilidade física*. Os espaços inteligentes atuam como ambientes para os dispositivos que os visitam e saem. Os usuários entram e saem com equipamentos que carregam, ou vestem; dispositivos de robótica podem se mover sozinhos, entrando e saindo do espaço. Segundo, existe a *mobilidade lógica*. Um processo, ou agente móvel, pode entrar ou sair de um espaço inteligente ou do dispositivo pessoal de um usuário. Além disso, a movimentação física de um dispositivo pode causar a movimentação lógica dos componentes dentro dele. Entretanto, tenha um componente se movido ou não, devido à movimentação física de seu dispositivo, a mobilidade lógica não ocorreu de nenhuma maneira interessante, a não ser que, como resultado, o componente tenha alterado alguma de suas associações com outros componentes. Terceiro, os usuários podem adicionar dispositivos relativamente estáticos no espaço, como novos reprodutores de mídia, substituindo os dispositivos mais antigos. Considere, por exemplo, a evolução de uma *casa inteligente*, cujos ocupantes variam o conjunto de dispositivos dentro dela [Edwards e Grinter 2001] de uma maneira relativamente não planejada com o passar do tempo. Quarto e último, os dispositivos podem falhar e, assim, “desaparecer” de um espaço.

Da perspectiva dos sistemas distribuídos, alguns desses fenômenos parecem semelhantes. Em cada caso, um componente de *software aparece* em um espaço inteligente previamente existente e, se resultar algo interessante, ele torna-se integrado a esse espaço, pelo menos temporariamente; ou, então, um componente *desaparece* do espaço por causa da mobilidade ou porque é simplesmente desligado ou porque falha. Pode ou não ser possível que qualquer componente em particular faça distinção entre dispositivos da “infraestrutura” e dispositivos “visitantes”.

Entretanto, existem distinções significativas a serem deduzidas no projeto de um sistema. Uma diferença importante que pode surgir entre sistemas voláteis é a taxa de alteração. Algoritmos que precisam aceitar vários componentes aparecendo ou desaparecendo por dia (por exemplo, em uma casa inteligente) podem ser projetados de forma muito diferente daqueles para os quais há pelo menos uma dessas mudanças ocorrendo em dado momento (por exemplo, um sistema implementado usando comunicação Blue-

tooth entre telefones celulares em uma cidade densamente povoada). Além disso, embora todos os fenômenos de aparecimento e desaparecimento anteriores pareçam semelhantes em uma primeira aproximação, é claro que existem diferenças importantes. Por exemplo, do ponto de vista da segurança, uma coisa é o dispositivo de um usuário entrar em um espaço inteligente, e outra é um componente de *software* externo se mover em um dispositivo de infraestrutura pertencente ao espaço.

Modelo de dispositivo • Com o surgimento da computação móvel e ubíqua, uma nova classe de dispositivos de computação está se tornando parte dos sistemas distribuídos. Esses dispositivos são limitados em sua fonte de energia e em seus recursos de computação; e eles podem ter maneiras próprias para fazer a interface com o mundo físico: sensores, como os detectores de luz, e/ou controladores, como um meio de movimentação programável.

Energia limitada: um dispositivo portátil ou incorporado no mundo físico normalmente precisa funcionar com baterias, e quanto menor e mais leve o dispositivo for, menor será a capacidade necessária de sua bateria. Substituir ou recarregar essas baterias provavelmente será inconveniente, em termos de tempo (pode haver centenas desses dispositivos por usuário) e de acesso físico. Processamento, acesso à memória e outras formas de armazenamento, tudo isso consome energia preciosa. A comunicação sem fio é particularmente dispendiosa em termos de energia. Além disso, a energia consumida pela recepção de uma mensagem pode significar uma fração substancial daquela exigida para transmiti-la; mesmo o modo de “espera”, no qual uma interface de rede está pronta para receber uma mensagem, pode exigir um consumo de energia considerável [Shih *et al.* 2002]. Assim, se um equipamento precisa permanecer, enquanto for possível, com determinado nível de carga na bateria, os algoritmos precisam ser sensíveis à energia que eles consomem, especialmente em termos de sua complexidade de troca de mensagem. No entanto, em última análise, a probabilidade de falha de dispositivo aumenta devido à descarga da bateria.

Restrições de recurso: os dispositivos móveis e ubíquos têm recursos computacionais limitados, em termos de velocidade do processador, capacidade de armazenamento e largura de banda de rede. Isso se dá, em parte, porque o consumo de energia aumenta quando melhoramos essas características. Porém, também acontece porque tornar os dispositivos portáteis, ou incorporá-los nos objetos físicos cotidianos, significa torná-los fisicamente pequenos, o que, dadas às limitações impostas pelos processos de fabricação, restringe o número de transistores nos microprocessadores e controladores. Isso resulta em dois problemas: como projetar algoritmos que possam ser executados em um tempo razoável, apesar dessas limitações, e como aumentar os recursos escassos usando os recursos do próprio ambiente.

Sensores e controladores: para permitir sua integração com o mundo físico – em particular, para fazê-los ter reconhecimento de contexto –, os dispositivos são equipados com *sensores* e *controladores*. Os sensores são dispositivos que medem parâmetros físicos e fornecem seus valores para o *software*. Inversamente, os controladores são dispositivos controlados por *software*, que afetam o mundo físico. Existe uma grande variedade de cada tipo de componente. No lado dos sensores, por exemplo, existem os que medem posição, orientação, carga e níveis de luz e som. Os controladores incluem aqueles para ar condicionado programável e motores. Um problema importante para os sensores é a precisão, que é bastante limitada e, portanto, pode levar a um comportamento espúrio, como uma resposta inadequada para o que se revela ser o lugar errado. A imprecisão provavelmente continuará sendo uma característica dos dispositivos que são baratos o suficiente para serem distribuídos por toda parte.

Os dispositivos descritos anteriormente parecem um tanto exóticos. Entretanto, eles não apenas estão disponíveis comercialmente, como até são produzidos em massa. Dois exemplos são as partículas (*motes*, em inglês) e os *smartphones*.

Partículas (motes): as partículas [Hill *et al.* 2000; www.xbow.com] são dispositivos destinados à operação autônoma em aplicações como a percepção ambiental. Eles são projetados para serem incorporados em um ambiente, programados de modo a descobrirem um ao outro, sem fio, e a transferir entre si os valores percebidos. Se, por exemplo, houver um incêndio em uma floresta, então uma ou mais partículas espalhadas pela floresta poderão sentir temperaturas anormalmente altas e comunicá-las, por intermédio de seus pares, a um dispositivo mais potente, capaz de repassar a situação para os serviços de emergência. A forma mais básica de partícula tem um processador de baixa potência (um microcontrolador) que executa o sistema operacional TinyOS [Culler *et al.* 2001] em uma memória flash interna, uma memória para registro de dados e um transceptor de sinais de rádio bidirecional ISM (*Industrial, Scientific and Medical*) de curto alcance. Uma variedade de módulos sensores pode ser adicionada. As partículas também são conhecidas como “pó inteligente”, refletindo o tamanho minúsculo pretendido, em última análise, para esses dispositivos, embora seu tamanho, quando este livro estava sendo escrito, fosse em torno de $6 \times 3 \times 1$ cm, excluindo-se o compartimento de bateria e os sensores. A Smart-its fornece funcionalidade similar para partículas com um fator de forma semelhante [www.smart-its.org]. A Seção 19.4.2 discutirá os usos de dispositivos do tipo partícula em redes de sensores sem fio.

Smartphones: os *smartphones* são exemplos bem diferentes de dispositivos nos sistemas que estamos considerando. Suas funções principais são a comunicação humana e a geração de imagens. No entanto, executando um sistema operacional como o Symbian, o Android ou o iOS, eles podem ser programados para uma ampla variedade de aplicações. Além de sua conectividade de dados remota, eles frequentemente têm interfaces de rede sem fio infravermelho (IrDA) ou Bluetooth de curto alcance, que permite conectar-se uns aos outros, com PCs e com equipamentos auxiliares. Frequentemente, eles contêm dispositivos sensores, como o GPS, para determinar sua localização, magnetômetros para indicar sua orientação e acelerômetros para perceber seu movimento. Além disso, eles podem executar *software* para reconhecimento de símbolos, como códigos de barra, a partir das imagens de suas câmeras, tornando-os sensores de “valores codificados” em objetos físicos, como produtos, que podem ser usados para acessar serviços associados. Por exemplo, um usuário poderia usar seu telefone com câmera para saber as especificações de um produto em uma loja, a partir do código de barras existente em sua caixa [Kindberg 2002].

Conectividade volátil • Todos os dispositivos de interesse deste capítulo têm alguma forma de conectividade sem fio, e podem ter várias. As tecnologias de conexão (seja Bluetooth, WiFi, 3G, etc.) variam em sua largura de banda e latência nominais, em seus custos de energia e no fato de existirem ou não custos financeiros na comunicação. Contudo, a volatilidade da conectividade – a variabilidade, em tempo de execução, do estado da conexão ou desconexão entre os dispositivos e a qualidade do serviço entre eles – também tem forte impacto sobre as propriedades do sistema.

Desconexão: as desconexões sem fio são bem mais prováveis de acontecer do que as desconexões em redes com fio. Muitos dos dispositivos que descrevemos são móveis e, portanto, podem ultrapassar sua distância operacional em relação aos ou-

Previamente configurada	Espontânea
Orientada a serviços: <i>cliente e servidor de e-mail</i>	Orientada a usuários humanos: <i>navegador web e servidores web</i>
Orientada a dados: <i>aplicativos de compartilhamento de arquivo P2P</i>	Orientada a dados: <i>aplicativos de compartilhamento de arquivo P2P</i>
Fisicamente orientada: <i>sistemas móveis e ubíquos</i>	Fisicamente orientada: <i>sistemas móveis e ubíquos</i>

Figura 19.2 Exemplos de associação previamente configurada versus espontânea.

etros dispositivos, ou encontrar obstruções de sinal de rádio entre eles, por exemplo, por causa de prédios. Mesmo quando os dispositivos são estáticos, podem existir usuários e veículos se movendo, os quais causam desconexão por obstrução. Também há a questão do roteamento de múltiplos *hops* (saltos) sem fio entre dispositivos. No *roteamento ad hoc*, um conjunto de dispositivos se comunica sem confiar em nenhum outro dispositivo: eles cooperam para direcionar todos os pacotes entre si. Usando nosso exemplo de partículas em uma floresta, uma partícula poderia continuar a se comunicar com todas as outras partículas em uma faixa de rádio imediata, mas deixar de comunicar sua leitura de temperatura alta para os serviços de emergência, devido à falha de partículas mais distantes, pelas quais todos os pacotes teriam de passar.

Largura de banda e latência variáveis: os fatores que podem levar a uma completa desconexão também podem levar a uma largura de banda e latência altamente variáveis, pois acarretam taxas de erro variáveis. À medida que a taxa de erro aumenta, cada vez mais pacotes são perdidos. Isso leva intrinsecamente a baixas taxas de desempenho de saída (*throughput*), mas a situação pode ser agravada por tempos limites (*timeout*) nos protocolos de camadas mais altas. Os valores de tempo limite são difíceis de adaptar em condições dramaticamente variáveis. Se eles forem grandes demais, comparados com as condições de erro correntes, a latência e o desempenho de saída sofrerão. Se eles forem pequenos demais, poderão aumentar o congestionamento e desperdiçar energia.

Interação espontânea • Em um sistema volátil, os componentes rotineiramente mudam o conjunto de componentes com que se comunicam à medida que se movem ou que outros componentes aparecem em seu ambiente. Usamos o termo *associação* para o relacionamento lógico formado quando pelo menos um componente de determinado par de componentes se comunica com o outro, durante um período de tempo bem definido, e *interação* para suas atividades durante a associação. Note que associação é diferente de conectividade: dois componentes (por exemplo, um cliente de *e-mail*, em um *notebook*, e um servidor de *e-mail*) podem estar correntemente desconectados, enquanto permanecem associados.

Em um espaço inteligente, as associações mudam porque os componentes tiram proveito de oportunidades para interagir com componentes locais. Um exemplo simples de tal oportunidade é quando um dispositivo utiliza uma impressora local onde quer que esteja no momento. Analogamente, talvez um dispositivo queira oferecer serviços para os clientes em seu ambiente local – como um “servidor pessoal” [Want *et al.* 2002] que o usuário ves-

te (por exemplo, em um cinto), o qual fornece dados sobre o usuário ou pertencentes a ele, para equipamentos próximos. É claro que certas associações estáticas ainda fazem sentido, mesmo em um sistema volátil; demos o exemplo de um computador *notebook* que viaja com seu proprietário pelo mundo, mas que só se comunica com um servidor de *e-mail* fixo.

Para colocar esse tipo de associação em um contexto maior de serviços na Internet, a Figura 19.2 mostra exemplos de três tipos de associação espontânea (à direita), comparadas com associações previamente configuradas (à esquerda).

As associações previamente configuradas são orientadas a serviços; isto é, os clientes têm uma necessidade a longo prazo de usar um serviço específico e, portanto, elas são previamente configuradas para serem associadas a ele. O trabalho de configuração dos clientes (incluindo configurá-los com o endereço do serviço exigido) é pequeno, comparado com o valor a longo prazo de usar o serviço, em particular.

No lado direito da figura estão os tipos de associação que variam rotineiramente, orientados por um operador humano, pela necessidade de dados específicos ou pelas circunstâncias físicas variáveis. Podemos considerar as associações entre um navegador Web e serviços Web como espontâneos e *orientadas a usuários humanos*: o usuário faz escolhas dinâmicas e (do ponto de vista do sistema) imprevisíveis de *links* para clicar e, assim, da instância de serviço a acessar. A Web é um sistema verdadeiramente volátil, e o fato de a mudança nas associações normalmente envolver um trabalho desprezível é importante para seu sucesso – os autores das páginas Web já fizeram o trabalho de configuração.

As aplicações *peer-to-peer* na Internet, como os programas de compartilhamento de arquivo, também são sistemas voláteis, mas elas são principalmente *orientadas a dados*. Esses dados frequentemente se originam do ser humano (por exemplo, o nome do conteúdo a ser buscado), mas é o valor dos dados fornecidos que faz com que um par estabeleça associações com outro, com o qual pode nunca ter se associado e cujo endereço não foi armazenado por ele anteriormente, por meio de um algoritmo de descoberta distribuído, baseado em dados.

Os sistemas móveis e ubíquos deste capítulo são diferenciados por exibirem espontaneidade de associações *fisicamente orientadas*. As associações são estabelecidas e desfeitas – às vezes por usuários humanos –, de acordo com as circunstâncias físicas correntes dos componentes, em particular, sua proximidade.

Menor confiança e privacidade • Conforme explicado no Capítulo 11, em última análise, a segurança nos sistemas distribuídos é baseada em *hardware* e *software* confiáveis – a base da computação confiável. Contudo, nos sistemas voláteis, a confiança é problemática, devido à interação espontânea. Que base de confiança pode haver entre componentes que são capazes de se associar espontaneamente? Os componentes que se movem entre espaços inteligentes podem pertencer a indivíduos ou organizações distintos e têm pouco ou nenhum conhecimento anterior uns dos outros ou de um terceiro participante confiável.

A privacidade é um problema importante para os usuários, que podem desconfiar dos sistemas por causa de seus recursos de percepção. A presença de sensores nos espaços inteligentes significa que se torna possível rastrear os usuários eletronicamente, em uma escala potencialmente maciça e jamais vista. Tirando proveito dos serviços de reconhecimento de contexto – como no exemplo das salas que configuram o ar condicionado de acordo com as preferências dos usuários que estão dentro delas –, os usuários podem permitir que outros saibam onde eles estavam e o que estavam fazendo lá. Para piorar as coisas, eles podem nem sempre saber que estavam sendo seguidos. Mesmo que o usuário não revele sua identidade, é possível que outros a saibam e, assim, descubram o que um indivíduo específico faz – por exemplo, observando as viagens regulares entre uma casa e um local de trabalho e correlacionando-os com o uso de um cartão de crédito em algum lugar entre esses dois pontos.

19.2 Associação

Conforme explicado anteriormente, os dispositivos estão sujeitos a aparecer e desaparecer nos espaços inteligentes de maneira imprevisível. Apesar disso, os componentes voláteis precisam interagir – preferivelmente sem intervenção do usuário. Em outras palavras, um dispositivo que aparece em um espaço inteligente precisa conseguir se inicializar na rede local para possibilitar a comunicação com outros dispositivos e se associar apropriadamente no espaço inteligente:

Inicialização na rede: normalmente, a comunicação ocorre por meio de uma rede local. O dispositivo deve primeiro adquirir um endereço na rede local (ou registrar um endereço previamente existente, como um endereço IP móvel); ele também pode adquirir ou registrar um nome.

Associação: os componentes do dispositivo se associam aos serviços no espaço inteligente ou fornecem serviços para componentes em qualquer parte do espaço inteligente (ou ambos).

Inicialização na rede • Existem soluções bem estabelecidas para o problema da integração de um dispositivo na rede. Algumas dessas soluções contam com servidores acessíveis dentro do espaço inteligente. Por exemplo, um servidor de DHCP (veja a Seção 3.4.3) pode fornecer um endereço IP, outros parâmetros de interligação em rede e DNS, os quais o dispositivo obtém executando uma consulta em um endereço conhecido. No espaço inteligente, os servidores também podem atribuir um nome de domínio exclusivo ao dispositivo; ou, se houver acesso à Internet aberta, o dispositivo pode usar um serviço dinâmico de atualização de DNS para registrar seu novo endereço IP em um nome de domínio estático.

Um caso mais interessante é a atribuição de parâmetros de interligação em rede na ausência de qualquer infraestrutura de serviço no espaço inteligente, ou fora dele. Isso é desejável tanto para simplificar o espaço inteligente como para evitar dependências de serviços que poderiam falhar. O padrão IPv6 inclui um protocolo para atribuição de endereço sem servidor. O grupo de trabalho Zero Configuration Networking do IETF [www.zeroconf.org] está desenvolvendo padrões para a atribuição de endereço sem servidor, pesquisa de nome de domínio, atribuição de endereço *multicast* e descoberta de serviços (veja a próxima subseção). O Bonjour da Apple [www.apple.com] é uma implementação comercial de grande parte dessa funcionalidade. Assim como no acesso a DHCP, todos esses métodos utilizam *broadcast* ou *multicast* por meio da rede local, usando um endereço conhecido. Qualquer dispositivo pode receber ou transmitir nesses endereços.

O problema da associação e o princípio do limite • Uma vez que um dispositivo possa se comunicar no espaço inteligente, ele se depara com o *problema da associação*: a questão de como se associar adequadamente dentro dele. As soluções para o problema da associação devem tratar de dois aspectos principais: escala e abrangência. Primeiro, podem existir muitos dispositivos dentro do espaço inteligente e, talvez, muito mais componentes de *software* nesses dispositivos. Com quais deles, se for o caso, os componentes do dispositivo que está aparecendo deve interoperar e escolher de forma eficiente?

Segundo, como podemos restringir a *abrangência* ao solucionar esse problema, de modo a considerar apenas os componentes do espaço inteligente – e todos os componentes do espaço inteligente – em vez dos possíveis trilhões de componentes que estão de fora? A abrangência é parcialmente um tipo de problema de escala, mas não apenas isso. Normalmente, um espaço inteligente tem limites territoriais e administrativos, os quais podem fazer uma grande diferença para usuários e administradores. Por exemplo,

se um dispositivo serve para descobrir um serviço, como uma impressora em um quarto de hotel, ele deve encontrar uma impressora no seu quarto e não no quarto vizinho. Igualmente, se houver uma impressora apropriada no quarto do usuário, então uma solução deverá incluí-la como uma candidata para a associação.

O princípio do limite diz que os espaços inteligentes precisam ter limites de sistema que correspondam precisamente aos espaços significativos, de acordo como eles são normalmente definidos territorial e administrativamente [Kindberg e Fox 2001]. Esses “limites de sistema” são critérios definidos pelo sistema que abrangem, mas não necessariamente restringem a associação.

Uma tentativa de solução para o problema da associação é usar um serviço de descoberta, como descrito a seguir. Normalmente, os serviços de descoberta são baseados em *multicast* em uma sub-rede, o que tem a desvantagem de que o alcance da sub-rede talvez não coincida com os serviços disponíveis em um espaço inteligente – aqueles que violam o princípio do limite –, conforme vamos explicar. A Seção 19.2.2 descreve, então, algumas soluções que fornecem associações mais precisamente abrangentes, contando com parâmetros físicos e interface de entrada de dados.

19.2.1 Serviços de descoberta

Os clientes identificam os serviços fornecidos em um espaço inteligente usando um *serviço de descoberta*. Um serviço de descoberta é um serviço de diretório (veja a Seção 13.3) no qual os serviços de um espaço inteligente são registrados e pesquisados por meio de seus atributos, mas cuja implementação leva em conta as propriedades voláteis do sistema. Primeiro, os dados de diretório exigidos por um cliente em particular – isto é, o conjunto de atributos do serviço no qual as consultas devem ser executadas – são determinados em tempo de execução. Os dados de diretório são determinados dinamicamente como uma função do contexto do cliente – neste caso, o espaço inteligente em particular onde as consultas ocorrem. Segundo, pode não haver infraestrutura no espaço inteligente para conter um servidor de diretório. Terceiro, os serviços registrados no diretório podem desaparecer espontaneamente. Quarto, os protocolos usados para acessar o diretório precisam ser sensíveis ao consumo de energia e largura de banda.

Existem tanto serviços de *descoberta de dispositivo* como de *descoberta de serviço*; o Bluetooth inclui ambos. Na descoberta de dispositivo, os clientes descobrem os nomes e endereços de dispositivos presentes no mesmo local. Normalmente, eles escolhem um dispositivo individual com base em uma informação externa ao sistema (como a seleção por um ser humano) e o consultam para saber os serviços que ele oferece. Por outro lado, um serviço de descoberta de serviço é usado onde os clientes não estão preocupados com qual dispositivo fornece o serviço que precisam, mas somente com os atributos do serviço. Esta descrição se concentrará nos serviços de descoberta de serviço e, a não ser que seja declarado de outra forma, é a isso que nos referiremos daqui por diante como serviços de descoberta.

Um serviço de descoberta tem uma interface para automaticamente registrar e anular o registro dos serviços que estão disponíveis para associação, assim como uma interface para os clientes pesquisarem serviços que estão correntemente disponíveis a partir desses, para fazerem a associação com um serviço apropriado. A Figura 19.3 dá um exemplo fictício e simplificado dessas interfaces. Primeiro, existem chamadas para registrar a disponibilidade de um serviço com determinado endereço e atributos e, subsequentemente, para gerenciar seu registro. Em seguida, existe uma chamada para pesquisar os serviços que correspondem a uma especificação de atributos exigidos. Zero ou mais serviços podem corresponder à especificação; cada um é retornado com seu endereço e seus atributos. Note que, por si só, um serviço de descoberta não

Métodos para registro/cancelamento de registro de serviço	Explicação
<i>arrendamento:= registrar(endereço, atributos)</i>	Registra o serviço no endereço dado, com os atributos dados; é retornado um arrendamento (<i>/lease</i>)
<i>atualizar(arrendamento)</i>	Atualiza o arrendamento retornado no registro
<i>cancelarRegistro(arrendamento)</i>	Remove o registro do serviço registrado sob o arrendamento dado
Método invocado para pesquisar um serviço	
<i>conjuntoServiço:= consultar(especificaçãoAtributo)</i>	Retorna um conjunto de serviços registrados cujos atributos correspondem à especificação dada

Figura 19.3 Interface para um serviço de descoberta.

permite associação: também é exigida a *seleção do serviço* – a escolha de um serviço no conjunto retornado. Isso pode ocorrer por meio de um programa ou pela listagem dos serviços correspondentes, para um usuário escolher.

Os aprimoramentos nos serviços de descoberta incluem o serviço de descoberta Jini (veja a seguir), o protocolo de localização de serviço [Guttman 1999], o Intentional Naming System [Adjie-Winoto *et al.* 1999], o protocolo de descoberta de serviço simples, que é o cerne da iniciativa Universal Plug and Play [www.upnp.org] e o Secure Service Discovery Service [Czerwinski *et al.* 1999]. Também existem serviços de descoberta na camada de enlace, como o do Bluetooth.

Os problemas a serem tratados no projeto de um serviço de descoberta são os seguintes:

Pouco esforço, associação apropriada: de preferência, as associações apropriadas seriam feitas sem qualquer trabalho humano. Primeiro, o conjunto de serviços retornado pela operação *consulta* (Figura 19.3) seria apropriado – eles seriam precisamente os serviços existentes no espaço inteligente que corresponderam à consulta. Segundo, a seleção do serviço poderia ser feita por programa, ou com intervenção humana mínima, para atender às necessidades dos usuários.

Descrição do serviço e linguagem de consulta: o objetivo global é corresponder os serviços com as solicitações de serviços dos clientes. Isso pressupõe uma linguagem para descrever os serviços disponíveis e outra para expressar os requisitos do serviço. As linguagens de consulta e de descrição precisam concordar entre si (ou serem traduzíveis); e sua expressividade precisa acompanhar o ritmo do desenvolvimento de novos dispositivos e serviços.

Descoberta específica do espaço inteligente: queremos um mecanismo para os dispositivos acessarem uma instância (ou abrangência) do serviço de descoberta que seja apropriada para as circunstâncias físicas correntes – um mecanismo que não conte com o conhecimento *a priori*, por parte desse dispositivo, do nome ou endereço em particular desse serviço. Na prática, os serviços de descoberta são relacionados a um espaço inteligente em particular apenas por intermédio do alcance limitado do *multicast* em uma sub-rede que o contém, conforme explicaremos.

Implementação de diretório: logicamente, cada instância de um serviço de descoberta envolve um diretório de serviços disponíveis que pode ser consultado. Existem várias maneiras de implementar tal diretório, com diferentes implicações

para a largura de banda da rede, oportunidade de descoberta de serviço e consumo de energia.

Volatilidade do serviço: todo serviço em um sistema volátil precisa tratar eficiente e normalmente do desaparecimento de um cliente. Um serviço de descoberta tem os serviços como seus clientes e precisa tratar do desaparecimento deles adequadamente. Como exemplo de associação por descoberta, considere um visitante ocasional, ou que visita pela primeira vez uma organização ou hotel, o qual precisa imprimir um documento a partir de um *notebook*. É razoável não esperar que o usuário tenha em seu *notebook* os nomes das impressoras locais em particular configuradas ou que adivinhe seus nomes (como \\myrtle\\titus e \\lione\\frederick). Em vez de obrigar o usuário a configurar sua máquina enquanto faz a visita, seria preferível o *notebook* usar a chamada de consulta de um serviço de descoberta para encontrar o conjunto de impressoras de rede disponíveis que correspondam às suas necessidades. Uma impressora em particular pode ser selecionada por meio da interação com o usuário, ou pela consulta de um registro das preferências do usuário.

Os atributos exigidos do serviço de impressão podem, por exemplo, especificar se ela é uma impressora a laser ou de jato de tinta, se fornece impressão colorida ou não, e sua localização física com relação ao usuário (por exemplo, o número do quarto).

Correspondentemente, os serviços fornecem seus endereços e atributos para o serviço de descoberta com a chamada *registrar*. Por exemplo, uma impressora (ou um serviço que a gerencia) pode registrar seu endereço e seus atributos no serviço de descoberta, como segue:

```
serviceAddress=http://www.hotelDuLac.com/services/printer57;  
resourceClass=printer; type=laser; colour=yes, resolution=600dpi,  
location=room101
```

A maneira normal de inicializar o acesso ao serviço de descoberta local em tempo de execução, sem configuração manual, é usar o alcance da sub-rede local; especificamente, fazer as consultas para um endereço *multicast* IP conhecido, pela sub-rede local. O endereço *multicast* IP é conhecido *a priori* por todos os dispositivos que precisam acessar o serviço de descoberta. Os serviços de descoberta baseados apenas no alcance da rede às vezes são conhecidos explicitamente como *serviços de descoberta de rede*.

Note que algumas redes, como o Bluetooth, usam saltos de frequência e não podem se comunicar com todos os dispositivos da vizinhança simultaneamente em um nível físico. O Bluetooth obtém a descoberta usando um correlato do “endereço conhecido”: uma sequência conhecida de saltos de frequência. Os dispositivos que podem ser descobertos circulam pelas frequências mais lentamente do que os que estão tentando descobri-los, de modo que os remetentes (descobridores) e receptores finalmente coincidem na frequência e estabelecem a comunicação.

Existem várias escolhas de projeto ao se implementar um serviço de descoberta, as quais podem ter um efeito considerável sobre as maneiras como ele pode ser usado.

A primeira é se o serviço de descoberta deve ser implementado por um *servidor de diretório* ou sem servidor. Um servidor de diretório contém um conjunto de descrições de serviços que se registraram nele e responde aos clientes executando consultas em busca de serviços. Qualquer componente (servidor ou cliente) que queira usar o serviço de diretório local faz uma requisição em *multicast* para localizá-lo, e o servidor de diretório responde com seu endereço único (*unicast*). Então, o componente pode se comunicar com ele ponto a ponto – evitando a interrupção dos dispositivos não envolvidos, que ocorre na comunicação *multicast*. Isso funciona bem nos espaços inteligentes que fornecem infraestrutura. Frequentemente, o serviço de diretório pode ser executado ligado a uma máquina mais robusta. Contudo, nos espaços inteligentes mais simples, como as salas de

reunião, pode não haver recursos para a instalação de um servidor de diretório. Em princípio, seria possível eleger um servidor a partir dos dispositivos que estivessem presentes (Seção 15.3), mas qualquer servidor assim poderia desaparecer espontaneamente. Isso levaria à complexidade na implementação de clientes do serviço de descoberta, o qual, então, teria que se adaptar a um servidor de registro mutante. Além disso, as sobrecargas resultantes da reeleição podem ser grandes em um sistema altamente volátil.

Uma alternativa é a *descoberta sem servidor*, na qual os dispositivos participantes colaboram para implementar um serviço de descoberta distribuído, no lugar de um servidor de diretório. Assim como acontece com qualquer diretório distribuído, existem duas variantes de implementação principais. No modelo *push*, os serviços enviam por *multicast* (“anunciam”) suas descrições regularmente. Os clientes recebem os anúncios e executam nelas suas consultas, possivelmente armazenando as descrições em cache para o caso de serem necessárias posteriormente. No modelo *pull*, os clientes fazem o *multicast* de suas consultas. Os dispositivos que fornecem serviços verificam as consultas em relação às descrições que possuem e respondem apenas com as descrições que correspondem. Os clientes repetem suas consultas em intervalos, caso não haja nenhuma resposta.

Os modelos *push* e *pull* têm implicações na utilização de largura de banda e energia. Sempre que um dispositivo publica uma mensagem por *multicast*, largura de banda é consumida e todos os dispositivos que estão captando gastam energia recebendo a mensagem. Em um modelo *push* puro, os dispositivos precisam anunciar seus serviços regularmente para que os clientes que estejam aparecendo possam descobri-los. Contudo, isso será um desperdício de largura de banda e energia, caso não haja clientes precisando descobrir um serviço em particular. E o tempo que um cliente novo espera para saber da existência de serviços precisa ser ponderado em relação aos custos de largura de banda e energia, os quais aumentam com a frequência dos anúncios.

Em um modelo *pull* puro, um cliente pode descobrir os serviços disponíveis assim que surge no ambiente. E não há desperdício por *multicast*, se não houver necessidades de descoberta em determinado intervalo de tempo. Ainda assim, o cliente pode receber várias respostas, quando uma só resolveria. E não há nenhuma vantagem, por padrão, de requisições apresentando a mesma consulta – isso para serviços solicitados frequentemente.

É possível projetar protocolos mistos que tratam das deficiências anteriores, e o Exercício 19.2 fala disso.

Um serviço pode ativar a chamada *cancelarRegistro* (Figura 19.3) antes de desaparecer, mas igualmente pode desaparecer espontaneamente. A volatilidade do serviço é tratada de maneiras diferentes, de acordo com a arquitetura da implementação do diretório. Um servidor de diretório precisa saber, assim que possível, que um serviço registrado desapareceu, para não fornecer sua descrição de forma enganosa. Normalmente, isso é feito usando-se um mecanismo geral chamado *arrendamento*. Um arrendamento é uma alocação temporária de algum recurso de um servidor para um cliente, o qual só pode ser renovado por uma nova requisição do cliente antes que o prazo final do arrendamento se expire. Se o cliente deixar de renová-lo (por exemplo, com a chamada *atualizar* da Figura 19.3), o servidor retirará (e poderá realocar) o recurso. Apresentamos os arrendamentos como parte do Jini, na Seção 5.4.3; além disso, os servidores de DHCP usam arrendamentos ao alocar endereços IP. Um servidor de diretório só manterá o registro de um serviço se este se comunicar periodicamente com o servidor de diretório para renovar seu arrendamento. Vemos aqui um compromisso semelhante da oportunidade em relação ao consumo de largura de banda e energia – quanto menor o período de arrendamento, mais rapidamente o desaparecimento de um serviço será notado, mas mais recursos de interligação em rede e energia serão exigidos. Em uma arquitetura sem servidor, nada precisa

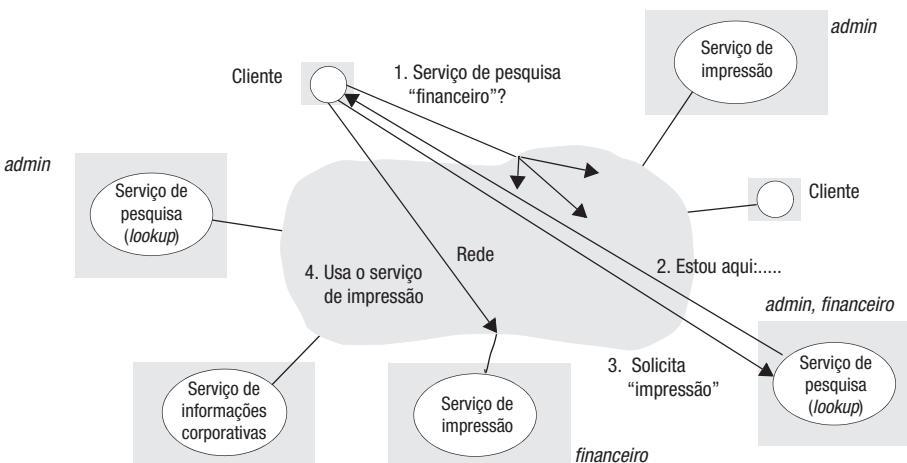


Figura 19.4 Descoberta de serviço no Jini.

ser feito (exceto eliminar as entradas antigas nos dispositivos que armazenam os serviços na cache), pois um serviço que desapareceu não se anunciará mais; e um cliente usando um protocolo baseado em *pull* só poderá descobrir os serviços presentes.

Jini • O Jini [Waldo 1999; Arnold *et al.* 1999] é um sistema projetado para ser usado por sistemas móveis e ubíquos. Ele é totalmente baseado em Java – ele presume que máquinas virtuais Java são executadas em todos os computadores, permitindo que eles se comuniquem uns com os outros por meio de RMI, ou eventos (veja os Capítulos 5 e 6), e façam o *download* de código, conforme for necessário. Descreveremos aqui o sistema de descoberta do Jini.

Os componentes relacionados à descoberta em um sistema Jini são serviços de *pesquisa (lookup)*, serviços Jini e clientes Jini (veja a Figura 19.4). O serviço de *pesquisa* implementa o que chamamos de serviço de descoberta, embora o Jini use o termo “descoberta” apenas para localizar o serviço de pesquisa em si. O serviço de pesquisa permite que os serviços Jini registrem as tarefas que oferecem, e os clientes Jini solicitam os serviços que correspondem aos seus requisitos. Um serviço Jini, como um serviço de impressão, pode ser registrado em um ou mais serviços de pesquisa. Um serviço Jini fornece (e os serviços de pesquisa armazenam) um objeto que providencia o serviço, assim como os atributos do serviço. Os clientes Jini consultam os serviços de pesquisa para encontrar serviços Jini que correspondam aos seus requisitos; caso uma correspondência seja encontrada, eles fazem o *download* de um objeto que dá acesso ao serviço, a partir do serviço de pesquisa. A correspondência que o serviço oferece às requisições dos clientes pode ser baseada em atributos ou em tipos Java, por exemplo, permitindo que um cliente solicite uma impressora colorida para a qual possui a interface Java correspondente.

Quando um cliente ou serviço Jini inicia, ele envia uma requisição para um endereço *multicast* IP conhecido. Qualquer serviço de pesquisa que receba essa requisição, que possa responder, envia seu endereço, permitindo que o solicitante realize uma invocação remota para pesquisar ou registrar um serviço nele (no Jini, o registro é chamado de *união – joining*, em inglês). Os serviços de pesquisa também anunciam sua existência em datagramas enviados para o mesmo endereço *multicast*. Os clientes e serviços Jini também podem captar o endereço *multicast* para que saibam da existência de novos serviços de pesquisa.

Pode haver várias instâncias do serviço de pesquisa acessíveis por comunicação *multicast* a partir de determinado cliente ou serviço Jini. Toda instância de tal serviço é configurada com um ou mais nomes de *grupo*, como *admin*, *financeiro* e *vendas*, os quais atuam como rótulos de escopo. A Figura 19.4 mostra um cliente Jini descobrindo e usando um serviço de impressão. O cliente solicita um serviço de pesquisa no grupo *financeiro*; portanto, ele envia em *multicast* uma requisição contendo esse nome de grupo (mensagem 1 na figura). Somente um serviço de pesquisa está ligado ao grupo *financeiro* (o serviço que também está ligado ao grupo *admin*) e esse serviço responde (2). A resposta do serviço de pesquisa inclui seu endereço, e o cliente se comunica diretamente com ele, por RMI, para localizar todos os serviços de tipo “impressão” (3). Somente um serviço de impressão se registrou nesse serviço de pesquisa sob o grupo *financeiro* e é retornado um objeto para acessar esse serviço em particular. Então, o cliente usa o serviço de impressão diretamente, utilizando o objeto retornado (4). A figura também mostra outro serviço de impressão, que está no grupo *admin*. Também há um serviço de informações corporativas, que não está ligado a nenhum grupo em particular (e que pode ser registrado em todos os serviços de pesquisa).

Discussão sobre os serviços de descoberta de rede • Os serviços de descoberta baseados apenas no alcance da rede que acabamos de descrever avançam até certo ponto na solução do problema da associação. Existem implementações de diretório eficientes, incluindo as que não contam com uma infraestrutura. Em muitos casos, o número de clientes e serviços que podem ser atingidos por meio de uma sub-rede é administrável em termos de custos de computação e rede; portanto, a escala frequentemente não é um problema. Descrevemos medidas para suportar a volatilidade dos sistemas.

No entanto, os serviços de descoberta de rede criam duas dificuldades quando vistos da perspectiva do princípio do limite: o uso de uma sub-rede e falta de adequação na maneira como os serviços são descritos.

A sub-rede pode ser uma aproximação insatisfatória de um espaço inteligente. Primeiro, a descoberta de rede pode incluir, por engano, serviços que não estão no espaço inteligente. Considere um quarto de hotel, por exemplo. As transmissões baseadas em sinais de radiofrequência (RF), como 802.11 ou Bluetooth, normalmente penetram nas paredes dos quartos de outros hóspedes. Segundo o exemplo do Jini, os serviços poderiam ser divididos logicamente por grupos – um grupo por quarto de hotel. Contudo, isso levanta a questão de como o quarto de hotel do usuário vai se tornar um parâmetro para o serviço de descoberta. Segundo, a descoberta de rede pode, por engano, não levar em conta serviços que estão “no” espaço inteligente, no sentido de serem qualificados para descoberta, mas que são colocados fora de sua sub-rede. O estudo de caso do Cooltown (veja a Seção 19.7.1) ilustra o modo como entidades não eletrônicas, como os documentos impressos em um espaço inteligente, podem ser associadas a serviços contidos fora do espaço inteligente.

Além disso, os serviços de descoberta de rede nem sempre estabelecem associações apropriadas, pois a linguagem usada para descrever os serviços pode ser inadequada em dois aspectos. Primeiro, a descoberta pode ser *frágil*: mesmo ligeiras variações no vocabulário da descrição do serviço usado por organizações distintas poderiam fazer com que ela falhasse. Por exemplo, o quarto de hotel tem um serviço chamado “Imprimir”, enquanto o *notebook* do hóspede procura “Impressão”. Variações no vocabulário da linguagem humana tendem a piorar esse problema. Segundo, pode haver *oportunidades perdidas* de acesso ao serviço. Por exemplo, existe uma “tela de fotografia digital” na parede do quarto de hotel, a qual exibirá instantâneos de férias no formato JPEG. A câmera do hóspede tem uma conexão sem fio e produz imagens nesse formato, mas ela não tem nenhuma descrição para o serviço – ela não foi atualizada com esse desenvolvimento relativamente recente. Portanto, a câmera é incapaz de tirar proveito dele.

19.2.2 Associação física

As deficiências dos sistemas de descoberta de rede podem ser resolvidas até certo ponto usando-se meios físicos, embora as soluções frequentemente exijam um grau maior de envolvimento humano. Foram desenvolvidas as técnicas a seguir.

Interação humana na descoberta de escopo • Neste caso, um ser humano fornece entrada para o dispositivo para configurar a abrangência da descoberta. Um exemplo simples disso seria digitar ou selecionar o identificador do espaço inteligente, como o número do quarto, no caso do hóspede de hotel. O dispositivo pode, então, usar o identificador como um atributo de grupo extra do serviço (como no Jini).

Percepção e canais fisicamente restritos para descoberta de escopo • Uma possibilidade menos trabalhosa é o usuário utilizar um sensor em seu dispositivo. Por exemplo, o espaço inteligente poderia ter um identificador apresentado em símbolos codificados, chamados de *glyfoss*, em documentos e superfícies no espaço – por exemplo, exibidos na tela da TV no quarto do hóspede de um hotel. O hóspede usa um telefone com câmera, ou outro dispositivo de geração de imagens, para decodificar tal símbolo, e o dispositivo usa o identificador resultante da maneira que descrevemos para entrada via intervenção humana. Outra possibilidade, para uso em espaços inteligentes ao ar livre, onde estão disponíveis sinais de navegação via satélite, é utilizar um sensor para obter a posição do espaço inteligente em coordenadas de latitude e longitude, e enviar essas coordenadas para um serviço remoto conhecido, o qual retorna o endereço do serviço de descoberta local. Entretanto, dadas às imprecisões na navegação via satélite, esse método pode ser menos preciso na identificação do espaço inteligente, caso existam outros espaços próximos.

Outra técnica que evita a entrada humana é usar um *canal fisicamente restrito* (veja também a Seção 19.5.2) – um canal de comunicação que, até certo grau de aproximação, penetra apenas a extensão física do espaço inteligente. Por exemplo, no quarto do hóspede, a TV poderia estar tocando música de fundo em volume baixo, com uma codificação digital do identificador do quarto sobreposta como uma perturbação inaudível do sinal [Madhavapeddy *et al.* 2003]; ou poderia ser um transmissor infravermelho (uma baliza, ou, em inglês, *beacon*) no quarto que propagasse o identificador [Kindberg *et al.* 2002a]. Esses dois canais são significativamente atenuados pelos materiais nos limites do quarto (supondo que as portas e janelas estejam fechadas).

Associação direta • O último conjunto de técnicas que vamos considerar aqui é o usuário humano usando um mecanismo físico para associar dois dispositivos diretamente, sem usar um serviço de descoberta. Normalmente, é aí que os dispositivos envolvidos oferecem apenas um ou um pequeno conjunto de serviços selecionados pelo usuário humano. Em cada uma das técnicas a seguir, o usuário humano ativa o dispositivo que está transportando para conhecer o endereço de rede (por exemplo, Bluetooth ou endereço IP) de um dispositivo de destino.

Percepção de endereço: usar um dispositivo para perceber (sentir) diretamente o endereço de rede do dispositivo de destino. As possibilidades incluem: ler um código de barras no dispositivo, que codifica seu endereço de rede; ou trazer um dispositivo para muito próximo do outro e usar um canal sem fio de curto alcance para ler seu endereço. Dois exemplos desses canais de curto alcance são (1) *Near Field Communication* [www.nfc-forum.org] – um padrão para comunicação bidirecional via rádio que abrange várias faixas curtas, mas tem uma variante para apenas até cerca de 3 centímetros; ou (2) transmissões de sinal infravermelho de alcance muito curto.

Estímulo físico: usar um estímulo físico para fazer o dispositivo de destino enviar seu endereço. Um exemplo aqui é irradiar um raio laser com modulação digital (outro canal fisicamente restrito) para o dispositivo de destino [Patel e Abowd 2003], transmitindo, assim, seu endereço para o destino, o qual responde com o seu endereço.

Correlação temporal ou física: usar estímulos correlacionados temporal ou fisicamente para associar dispositivos. A especificação SWAP-CA [SWAP-CA 2002] para rede sem fio em um ambiente doméstico introduziu um protocolo, às vezes referido como protocolo de dois botões, para os usuários humanos associarem dois dispositivos sem fio entre si. Cada dispositivo capta em um endereço *multicast* conhecido. Os usuários pressionam botões em seus respectivos dispositivos, mais ou menos simultaneamente, e os dispositivos enviam seus endereços de rede para o endereço *multicast*. É improvável que outra rodada desse protocolo ocorra ao mesmo tempo na mesma sub-rede. Portanto, os dispositivos são associados usando o endereço que chegar dentro de um pequeno intervalo de pressionamento dos botões. Existe um correlato físico interessante, ainda que raramente prático, para essa estratégia, no qual um usuário segura dois dispositivos na mesma mão e os sacode juntos [Holmquist *et al.* 2001]. Cada dispositivo tem um acelerômetro para captar seu estado de movimento. O dispositivo grava o padrão da vibração, calcula um identificador a partir dela e envia esse identificador por *multicast*, junto a seu endereço único, para um endereço *multicast* conhecido. Somente os dois dispositivos que experimentam exatamente esse padrão de aceleração – e dentro da faixa de comunicação direta – reconhecerão o identificador um do outro e, portanto, conhecerão o endereço um do outro.

19.2.3 Resumo e perspectiva

Esta seção descreveu o problema da associação de componentes em sistemas voláteis e algumas tentativas de resolver esse problema, variando da descoberta de rede até técnicas mais dependentes de seres humanos. Os sistemas móveis e ubíquos apresentam dificuldades únicas, pois são integrados ao nosso desorganizado mundo físico cotidiano, de espaços como ambientes doméstico e escritórios, tornando difícil encontrar soluções. Os seres humanos tendem a ter em mente fortes considerações territoriais e administrativas, quando julgam o que está em um espaço inteligente em particular e o que está fora dele. O princípio do limite diz que as soluções para o problema da associação precisam corresponder aos espaços físicos subjacentes até um grau que seja aceitável para os usuários humanos. Vimos que, frequentemente, certo grau de supervisão humana está envolvido, devido às deficiências dos sistemas de descoberta de rede. O estudo de caso do Cooltown (Seção 19.7) descreverá um modelo específico de envolvimento humano.

Ignoramos a escala como fator nas soluções do problema da associação, tendo por base que o mundo está dividido em espaços inteligentes, os quais, normalmente, têm tamanho administrável. Entretanto, existe pesquisa sendo feita sobre serviços de descoberta com escalabilidade – afinal, algumas aplicações poderiam considerar o planeta inteiro como um espaço inteligente. Um exemplo é o INS/Twine [Balazinska *et al.* 2002], que divide dados de diretório entre um conjunto de resolvedores *peer-to-peer*.

19.3 Interoperabilidade

Descrevemos as maneiras pelas quais dois ou mais componentes em um sistema volátil se associam e, agora, veremos a questão de como eles interagem. Os componentes se associam com base em certos atributos, ou dados, que um ou ambos possuem, mas isso

deixa as perguntas de qual protocolo eles usam para se comunicar e, em um nível mais alto, qual modelo de programação é mais conveniente para a interação entre eles. Esta seção trata dessas questões.

Os Capítulos 4, 5 e 6 descreveram modelos de interação, incluindo várias formas de comunicação entre processos, invocação a métodos remotos e chamada de procedimentos remotos. Uma suposição implícita a alguns desses modelos é que os componentes que estão interagindo são feitos para trabalhar juntos em um sistema ou aplicativo específico e que alterações nesses componentes são um problema de configuração a longo prazo, ou uma condição de erro em tempo de execução a ser tratada ocasionalmente. No entanto, essas suposições não são válidas nos sistemas móveis e ubíquos. Felizmente, conforme explicaremos nesta seção, alguns dos métodos de interação dos Capítulos 4, 5 e 6, além de alguns métodos novos, são mais convenientes para esses sistemas voláteis.

De preferência, um componente em um sistema móvel ou ubíquo poderia se associar com classes de serviços variadas e não apenas com um conjunto variável de instâncias da mesma classe de serviço. Isto é, é melhor evitar o problema da “oportunidade perdida”, descrito na seção anterior, no qual, por exemplo, uma câmera digital é incapaz de enviar suas imagens para uma tela de fotografia digital porque não pode interagir com o serviço de envio de fotos da tela.

Em outras palavras, um objetivo da computação móvel e ubíqua é que um componente deve ter uma chance razoável de interagir com um componente funcionalmente compatível, mesmo que este último esteja em um tipo de espaço inteligente diferente daquele para o qual foi originalmente desenvolvido. Isso exige algum acordo global entre os desenvolvedores de *software*. Dado o trabalho necessário para se chegar a um acordo, é melhor minimizar o que precisa ser concordado.

A principal dificuldade que a interação volátil enfrenta é a incompatibilidade da interface de *software*. Se, por exemplo, uma câmera digital espera invocar uma operação *pushImage*, e não existe tal operação na interface da tela de fotografia digital, então elas não podem interagir – pelo menos, não diretamente.

Existem duas estratégias principais para enfrentar esse problema. A primeira é permitir que as interfaces sejam heterogêneas, mas adaptá-las uma a outra. Por exemplo, se a tela de fotografia digital tivesse uma operação *sendImage* com os mesmos parâmetros e a mesma semântica de *pushImage*, então seria simples construir um componente que atuasse como *proxy* para a tela de fotografia digital, convertendo a ativação *pushImage* da câmera em uma ativação *sendImage* da tela.

Entretanto, é muito difícil realizar essa estratégia. Frequentemente, a semântica das operações pode variar, assim como a sintaxe; e superar a incompatibilidade semântica é uma atividade difícil e propensa a erros, de modo geral. Então, há um problema de escala: se existirem N interfaces, então potencialmente N^2 adaptadores precisam ser escritos – e cada vez mais interfaces serão criadas com o passar do tempo. Além disso, há a questão de como os componentes vão adquirir adaptadores de interface convenientes à medida que são novamente associados em um sistema volátil. Os componentes (ou os dispositivos que os contêm) não podem vir previamente carregados com todos os N^2 adaptadores possíveis; portanto, o adaptador correto precisa ser determinado e carregado em tempo de execução. Apesar de todas essas dificuldades, existem pesquisas sendo feitas sobre como tornar prática a adaptação da interface. Veja, por exemplo, Ponnekanti e Fox [2004].

Outra estratégia para a interação é obrigar as interfaces a terem sintaxe idêntica na mais ampla classe de componentes possível. Isso pode parecer irreal inicialmente, mas na verdade tem sido praticado amplamente e com êxito há várias décadas. O exemplo mais simples são os *pipes* do UNIX. Um *pipe* tem apenas duas operações, *leitura* e *escrita*, para

o transporte de dados entre componentes (processos) em suas duas extremidades. Com o passar dos anos, os programadores de UNIX criaram muitos programas que podem ler e/ou escrever dados em um *pipe*. Devido as suas interfaces padronizadas e às funcionalidades de processamento de texto genéricas, a saída de qualquer um desses programas pode alimentar a entrada de outro; usuários e programadores têm descoberto muitas maneiras úteis de combinar programas dessa forma – programas que foram escritos independentemente, sem conhecimento da funcionalidade específica dos outros programas.

Outro exemplo, de mais sucesso ainda, de sistema que obtém um alto grau de interação por meio de uma interface fixa, é a Web. O conjunto de métodos definidos pela especificação HTTP (veja a Seção 5.2) é pequeno e fixo; normalmente, um cliente Web usa apenas as operações *GET* e *POST* para acessar um servidor Web. A consequência das interfaces fixas é que um *software* relativamente estável – frequentemente, o navegador – é capaz de interagir com um conjunto de serviços em expansão. O que muda entre os serviços é o tipo e os valores do conteúdo trocado e a semântica de processamento do servidor, mas toda interação ainda é uma operação *GET* ou *POST*.

19.3.1 Programação orientada a dados para sistemas voláteis

Chamamos os sistemas que usam uma interface fixa de serviço, como os *pipes* do UNIX e a Web, de *orientados a dados* (ou, equivalentemente, *orientados a conteúdo*). O termo foi escolhido para fazer distinção de *orientados a objetos*. Um componente em um sistema orientado a dados pode ser invocado por qualquer outro componente que conheça a interface fixa. Por outro lado, um objeto, ou um conjunto de procedimentos, tem uma interface de um conjunto amplo e variado de interfaces possíveis e só pode ser invocado pelos componentes que conheçam sua interface em particular. Distribuir e explorar um número desconhecido de definições de interface especializadas é uma possibilidade muito mais problemática do que publicar e usar uma especificação de interface, como a especificação HTTP. Isso ajuda a explicar porque o sistema distribuído mais usado e heterogêneo que conhecemos é a Web, em vez de um conjunto de, digamos, objetos CORBA.

A flexibilidade dos sistemas orientados a dados tem um compromisso com a robustez. Nem sempre faz sentido dois componentes em particular interagirem, e há pouca base para os programas verificarem a compatibilidade. Em um sistema orientado a objetos, ou a procedimentos, os programas podem pelo menos verificar se suas assinaturas de interface correspondem, mas um componente orientado a dados só pode impor a compatibilidade verificando o tipo de dados enviado a ele. Ele deve fazer isso por intermédio de descritores de tipo de dados padronizados fornecidos como metadados (como os tipos MIME de conteúdo Web) ou verificando os valores de dados passados a ele; por exemplo, os dados JPEG começam com informações de cabeçalho reconhecíveis.

Examinaremos agora alguns modelos de programação que têm sido usados para sistemas voláteis por causa de seus recursos de interação orientados a dados. Começaremos com dois modelos de interação entre componentes associados indiretamente: sistemas de eventos e espaços de tuplas. Vamos descrever dois projetos de interação entre dispositivos associados diretamente: *JetSend* e *Speakeasy*.

Sistemas baseados em eventos • Apresentamos os sistemas baseados em eventos [Bates *et al.* 1996] na Seção 6.3. Os sistemas de evento fornecem instâncias de serviços de evento. Cada sistema oferece uma interface fixa e genérica por meio da qual componentes chamados *geradores de eventos* (*publishers*) divulgam dados estruturados conhecidos como eventos e, correspondentemente, componentes chamados *assinantes* recebem os eventos. Cada serviço de evento é associado a um escopo físico ou lógico de distribuição

de eventos. Os assinantes só recebem (“manipulam”) eventos que (1) são publicados no mesmo serviço de evento e (2) correspondem às suas especificações registradas a respeito de quais eventos estão interessados.

Os eventos são um paradigma de programação natural para anunciar e tratar das alterações experimentadas pelos componentes, enquanto estão em um sistema volátil ou quando se movem entre sistemas voláteis. Os eventos podem ser definidos para especificar novos estados, como as mudanças no local de um dispositivo. Um exemplo recente de sistema que usa eventos para computação ubíqua é o *one.world* [Grimm 2004]. No entanto, os eventos têm sido usados em sistemas ubíquos desde o início de seu desenvolvimento. No sistema *Active Badge* [Harter e Hopper 1994], os aplicativos podem assinar eventos de mudança de lugar que ocorrem quando os usuários se movem. Os eventos de localização também apresentam o problema da detecção de padrões de eventos que ocorrem juntos, ou em sucessão, também conhecidos como eventos compostos. Por exemplo, considere o problema de detectar quando dois usuários estão no mesmo lugar, em que se sabe apenas quando os usuários individuais entram ou saem de um local em particular. Um sistema de localização não detecta sozinho tais ocorrências: há necessidade de regras que especifiquem, em termos de eventos primitivos, como “Chegada(usuário, localização, hora)” e “Saída(usuário, localização, hora)”, quando ocorrem eventos compostos como a localização coincidente.

Embora a publicação, a assinatura e o tratamento da interface para evento sejam dados (com variações relativamente pequenas entre os sistemas de evento), os geradores de eventos (*publishers*) e assinantes só podem interagir corretamente se concordarem com o serviço de evento que utilizam (pode haver muitas instâncias) e com os tipos e atributos dos eventos – sua sintaxe e semântica. Assim, os sistemas de eventos transferem, em vez de resolver, o problema da interação ubíqua. Para que determinado componente interaja em uma ampla variedade de espaços inteligentes seriam necessários padrões para tipos de eventos e, de preferência, os eventos seriam descritos em uma linguagem de marcação independente de linguagem de programação, como a XML ou JSON.

Por outro lado, os produtores e consumidores de evento não precisam identificar-se entre si. Isso pode ser uma vantagem em um sistema volátil, no qual poderia ser difícil rastrear quais outros componentes estão presentes. Dois componentes se comunicam publicando e assinando eventos correspondentes e concordando sobre a abrangência da distribuição do evento – em outras palavras, eles se associam indiretamente.

A abrangência da distribuição do evento em si é um assunto interessante nos sistemas móveis e ubíquos. Assim como acontece com a descoberta de serviço, surge a questão de como a abrangência de um serviço de eventos está relacionada à extensão física do espaço inteligente. Esse ponto é o assunto do Exercício 19.7.

Espaços de tuplas • Assim como os sistemas baseados em eventos, os espaços de tuplas também são um paradigma de programação amadurecido que tem encontrado aplicação nos sistemas voláteis. O paradigma do espaço de tuplas foi apresentado na Seção 6.5.2 como um paradigma de comunicação indireta que suporta a adição e a recuperação de dados estruturados, chamados tuplas, em um espaço de tuplas. Os sistemas de espaço de tuplas permitem a troca de tuplas específicas do aplicativo, e a base para associação e interação é o acordo dos componentes sobre as estruturas das tuplas e os valores contidos nelas.

Como exemplo do suporte que elas podem oferecer à computação ubíqua, uma câmera digital poderia descobrir o espaço de tuplas do espaço inteligente local – um quarto de hotel, digamos – e colocar suas imagens no espaço de tuplas usando uma tupla como a seguinte:

<“A torre inclinada”, “image/jpeg”, <jpeg data>>

Os projetistas do *software* da câmara têm apenas um modelo de um espaço de tuplas no qual as imagens são colocadas em um determinado formato, e nenhum modelo em particular de como deve ser feito o processamento dessas imagens.

Correspondentemente, um dispositivo que “consome” as imagens, como a tela de fotografia digital, poderia ser programado para descobrir seu espaço de tuplas e tentar recuperar tuplas que atendem o padrão abaixo, no qual “*” representa um valor curinga:

`<*, “image/jpeg”, *>`

A tupla da câmara corresponde ao padrão exigido pela tela de fotografia – ela tem três campos e seu segundo campo contém o *string* de tipo MIME exigido. Assim, a tela de fotografia recuperará a tupla da câmara e poderá exibir a imagem e o título associado. Como outro exemplo, o usuário poderia ter invocado uma impressora para recuperar a imagem do espaço de tuplas e imprimi-la.

Vários sistemas de programação baseados em espaços de tuplas foram desenvolvidos especificamente para sistemas móveis e ubíquos, conforme discutido a seguir.

O event heap: a despeito de seu nome, o *event heap* [Johanson e Fox 2004] é um sistema de programação baseado em tuplas desenvolvido para um tipo de espaço inteligente conhecido como “iRoom”, o qual contém várias telas e outros dispositivos de infraestrutura. Para cada iRoom existe um conjunto (*heap*) de eventos correspondentes, no qual os componentes que estão no iRoom – incluindo aqueles nos dispositivos móveis trazidos para a sala – podem descobrir ou ser configurados para usar. Os componentes interagem trocando tuplas por intermédio do conjunto de eventos, e ele fornece um nível de procedimento indireto que facilita a associação dinâmica entre os dispositivos. Um exemplo é um dispositivo de controle remoto mantido no iRoom que pode ser associado dinamicamente a diferentes telas. Um vídeo pode ser apresentado em qualquer uma das diversas telas. Quando um usuário pressiona o botão de “pausa” no controle remoto, o controle coloca uma tupla “pausa” no conjunto de eventos. Qualquer que seja o dispositivo que esteja exibindo o vídeo, ele está programado para procurar e recuperar tuplas “pausa” e, assim, responder. O controle remoto poderia trabalhar com um dispositivo de saída de áudio exatamente da mesma maneira, sem reprogramação.

LIME: o sistema LIME (acrônimo de Linda in a Mobile Environment – Linda em um ambiente móvel) [Murphy *et al.* 2001] foi desenvolvido como um modelo de programação para sistemas móveis. No LIME, os dispositivos participantes contêm espaços de tuplas e não existe dependência com a infraestrutura. Cada dispositivo contém seu próprio espaço de tuplas. O LIME compartilha os espaços de tuplas individuais, quando seus dispositivos hospedeiros se associam, formando a união dos conjuntos de tuplas na agregação dos espaços compartilhados. Isso poderia, por exemplo, ser usado pelo serviço descoberta. Um componente que necessita de um serviço pode ser programado para tentar recuperar uma tupla que descreva uma instância do serviço necessário; um dispositivo que implementa um serviço correspondente seria programado para colocar uma tupla descritiva em seu espaço de tuplas. Quando os dois se conectassem, o LIME estabeleceria a correspondência e o suposto cliente obteria os detalhes do serviço.

Embora o modelo LIME seja simples de expor, não é fácil implementar uma semântica de consistência conveniente em face de conexões e desconexões arbitrárias. Os desenvolvedores do LIME fizeram suposições compreensivelmente irrealis para simplificar seu projeto, incluindo: que a conectividade se mantém uniformemente entre os dispositivos cujos espaços de tuplas são agregados e que as conexões e desconexões em um conjunto são dispostas em série e metodicamente.

TOTA: O projeto TOTA (acrônimo de Tuples On The Air – Tuplas no ar) [Mamei e Zambonelli 2009] apresenta uma modificação muito interessante na implementação padrão de espaços de tuplas. Nessa estratégia, as tuplas são injetadas na rede, sendo colocadas em um nó local, permitindo-se, então, que elas sejam clonadas e propagadas por toda a rede, de maneira semelhante aos protocolos de fofoca (*gossip*) (veja a Seção 18.4.1). O resultado final é um *campo de tuplas* representando uma disseminação espacial de determinada tupla. Então, um processo pode ler a tupla normalmente, usando correspondência associativa. Para suportar esse estilo de programação, uma tupla T é definida como $T=(C,P,M)$, onde C é o conteúdo da tupla, P é a regra de propagação para essa tupla e M é uma regra de manutenção que define como a tupla deve reagir aos eventos do ambiente ou à passagem do tempo. Por exemplo, Mamei e Zambonelli [2009] descrevem uma aplicação em um museu de arte, na qual diferentes ambientes do museu têm um dispositivo sem fio fixo e os visitantes também têm *smartphones* habilitados para conexão sem fio; juntos, eles formam uma rede *ad-hoc*. O *smartphone* de um visitante interessado em uma obra de arte em particular coloca uma tupla de consulta no sistema, cujo conteúdo é uma descrição da obra de arte, junto com um campo de distância. A regra de propagação diz que a tupla deve ser propagada para todos os nós nas proximidades, aumentando a distância em uma unidade a cada vez; a regra de manutenção diz que a tupla deve ser excluída após determinado período de duração, ou tempo de vida (*time-to-live*). Quando essa tupla chega ao local onde está a obra de arte, uma tupla de resposta é injetada no sistema, contendo uma descrição da obra, sua localização e um campo de distância; desta vez, a regra de propagação é retroceder o caminho até o visitante, aumentando a distância a cada vez; novamente, a regra de manutenção é implementar uma diretiva de tempo de vida. O resultado final é um modelo de programação altamente flexível, particularmente adequado à operação em ambientes *ad-hoc* espacialmente conectados e, de modo geral, na computação ubíqua.

L²imbo: L²imbo é uma implementação de espaço de tuplas replicado, projetado para operar em um ambiente móvel [Davies *et al.* 1998]. Embora a motivação mais comum para a replicação seja obter alta disponibilidade e tolerância à falha, o L²imbo explora a replicação para lidar com desconexão de dispositivos. O L²imbo adota uma estratégia totalmente replicada, em que cada nó mantém uma réplica. Assim, o objetivo é garantir a consistência desse conjunto de réplicas. Para conseguir isso, o L²imbo adota *multicast confiável sobre multicast IP*, conforme discutido na Seção 15.4.2 – usando especificamente a estratégia de *multicast confiável escalável*, apresentada naquela seção [Floyd *et al.* 1997]. O L²imbo suporta a criação de vários espaços de tuplas, e cada espaço de tuplas é mapeado exclusivamente em um endereço *multicast IP*. As implementações das operações *read* e *write* são relativamente simples, com *read* sendo frequentemente satisfeita localmente e *write* sendo mapeada em uma operação de *multicast* (confiável). A implementação de *take* é mais complexa, dado o requisito de retirada global. A estratégia amplia o conceito de posse no L²imbo, sendo que somente o proprietário é capaz de remover a tupla. Normalmente, o proprietário é o criador da tupla, mas ela pode ser subsequentemente transferida para outros processos.

Comparando sistemas baseados em eventos e espaços de tuplas • Se identificarmos “evento” com “tupla” e “especificação de interesse” com “padrão de correspondência de tupla”, há uma correspondência entre os dois modelos de interação. Ambos fornecem um nível de procedimento indireto que é útil para sistemas voláteis, pois a identidade dos componentes que produzem e consomem eventos, ou tuplas, é oculta uns dos outros, por padrão. Assim, o conjunto de componentes pode mudar de forma transparente. Entretanto, existem diferenças importantes. Primeiro, o modelo de evento é exclusivamente assíncrono, enquanto os sistemas de espaço de tuplas fornecem uma operação síncrona para recuperar uma tupla

correspondente. Pode ser mais fácil programar com operações síncronas. Por outro lado, é uma má ideia esperar que um componente em particular (por exemplo, um dispositivo de produção de imagem que foi encontrado dinamicamente) forneça uma tupla correspondente, pois a qualquer momento pode ocorrer uma desconexão.

A segunda diferença importante é o tempo de vida dos eventos e das tuplas. Por padrão, um evento não dura mais do que sua propagação entre geradores de eventos e assinantes. Uma tupla, em um espaço de tuplas, entretanto, pode durar mais que o componente lá colocado – e do que qualquer componente que a leia (em oposição a consumir destrutivamente). Essa persistência pode ser uma vantagem; por exemplo, a bateria da câmera de um usuário pode acabar após ele ter feito o *upload* de fotos no espaço de tuplas de um quarto de hotel, mas antes de tê-las transferido para outro dispositivo. Ao mesmo tempo, a persistência pode ser uma desvantagem: e se um conjunto de dispositivos puser tuplas em um espaço, mas elas nunca forem consumidas porque os componentes que as deveriam consumir se desconectaram? O conjunto de tuplas em tal espaço poderia crescer incontrolavelmente. Sem conhecimento global de um conjunto de componentes volátil seria impossível determinar quais tuplas seriam lixo.

Os projetistas do *event heap* reconheceram o problema da persistência dos iRooms. Eles optaram por permitir que as tuplas expirem (isto é, sejam coletadas como lixo) após estarem em um conjunto de eventos por um tempo especificado, o qual normalmente é escolhido de forma a corresponder ao período de tempo de uma interação humana. Isso evita, por exemplo, que um evento de “pausa”, não consumido de um controle remoto, cause um incômodo quando um usuário tentar ver um vídeo no dia seguinte.

Interação direta com o dispositivo • Os modelos de programação anteriores eram para a interação entre componentes associados indiretamente. O *JetSend* e o *Speakeasy* são sistemas projetados para interação por meio de associação direta.

JetSend: o protocolo *JetSend* [Williams 1998] foi projetado para interação entre aparelhos como câmeras, impressoras, scanners e TVs. O *JetSend* foi explicitamente projetado para ser orientado a dados, para que nenhum aparelho tivesse que ser carregado com *drivers* especializados, de acordo com os dispositivos específicos com os quais interagiria. Por exemplo, uma câmera *JetSend* pode enviar uma imagem para um dispositivo de consumo de imagem *JetSend*, como uma impressora ou TV, independentemente da funcionalidade específica do dispositivo consumidor. A operação genérica central entre dispositivos *JetSend* conectados é *sincronizar* o estado que um apresenta com o estado do outro. Isso significa transferir o estado em um formato sobre o qual os dispositivos negociam. Por exemplo, um dispositivo de produção de imagem, como um scanner, poderia ser sincronizado com um dispositivo de consumo de imagem, como uma tela digital, consumindo uma imagem do produtor no formato JPEG, escolhido dentre vários formatos de imagem que o produtor poderia oferecer. O mesmo scanner poderia igualmente ser sincronizado com uma TV, talvez usando um formato de imagem diferente.

Os projetistas do *JetSend* reconheceram que sua operação de sincronização se beneficiava apenas de uma interação – basicamente, a transferência de dados – entre dispositivos heterogêneos. Isso levanta a questão de como obter interações mais complexas entre dispositivos específicos. Por exemplo, como deve ser feita a escolha entre o modo monocromático e colorido ao se transferir uma imagem para ser impressa? Por suposição, o dispositivo de origem não tem nenhum *driver* para uma impressora específica. E não é possível que esse dispositivo seja programado *a priori* com a semântica de qualquer dispositivo arbitrário (incluindo dispositivos que ainda serão inventados) com que ele possa se conectar. A resposta do *JetSend* para esse problema foi contar com o ser humano para

selecionar as funções específicas do dispositivo de destino (digamos, uma impressora), usando uma interface com o usuário especificada por este, mas traduzidas em seu dispositivo de origem (por exemplo, uma câmera). É assim que a interação acontece rotineiramente na Web, quando os usuários interagem com serviços altamente heterogêneos por intermédio de seus navegadores: cada serviço com que eles interagem envia sua interface, na forma de *script* em linguagem de marcação, para o navegador, o qual o traduz para o usuário como um conjunto genérico de elementos de tela, e sem conhecimento da semântica específica do serviço. Os serviços Web (veja o Capítulo 9) são uma tentativa de substituir o usuário humano por programas, mesmo para interações complexas.

Speakeasy: o projeto *Speakeeasy* [Edwards *et al.* 2002] aplicou posteriormente os mesmos princípios de projeto do *JetSend* na interação entre dispositivos, mas com uma diferença: utilizou código móvel. Existem dois motivos para se usar código móvel. O primeiro é que um dispositivo, como uma impressora, pode enviar qualquer interface com o usuário para um usuário de outro dispositivo, como um *smartphone*. Uma implementação de código móvel de uma interface com o usuário pode realizar processamento local, como validação de dados de entradas, e pode fornecer modos de interação não disponíveis nas interfaces com o usuário que precisam ser especificadas em uma linguagem de marcação.

Entretanto, contra essa vantagem estão as implicações na segurança da execução de código móvel, a qual exige mecanismos de proteção sofisticados contra cavalos de Troia, e as implicações sobre os recursos da execução de código móvel em uma máquina virtual, em oposição ao processamento de um *script* em linguagem de marcação muito mais restrito.

O segundo motivo para ter código móvel na interação de dispositivo é a otimização da transferência de dados. Embora o código móvel do *Speakeeasy* tenha de trabalhar dentro das restrições da API do dispositivo hospedeiro, ele pode realizar interações arbitrárias com o dispositivo remoto que o enviou. Assim, por exemplo, o código móvel pode implementar um protocolo otimizado para transferir conteúdo específico para o tipo de conteúdo, por exemplo, o vídeo poderia ser compactado dinamicamente antes da transmissão. Em contraste, o *JetSend* só pode usar protocolos de transferência de conteúdo predefinidos.

19.3.2 Associações indiretas e soft state

Quando um serviço tem recursos suficientes para oferecer alta disponibilidade (como um serviço de infraestrutura), faz sentido que os componentes se associem a ele explicitamente – isto é, conheçam seu endereço. Quando, posteriormente, os componentes usarem esse endereço para interagir com o serviço – digamos, dez minutos após a associação – poderão esperar que ele ainda esteja acessível e responda. Entretanto, em geral, a volatilidade do sistema torna inapropriado contar com um serviço fornecido por um componente em particular, pois este poderia sair ou falhar a qualquer momento. Uma lição a ser aprendida dessa distinção é que é interessante os programadores saberem quais serviços são de alta disponibilidade e quais são voláteis. Além disso, para suportar a volatilidade, eles precisam conhecer técnicas de programação que não envolvam a confiança em um componente específico.

Alguns dos exemplos anteriores de sistemas de programação orientados a dados envolviam associações anônimas indiretas. Especificamente, os componentes que interagem por intermédio de um sistema de evento, ou por um espaço de tuplas, não conhecem necessariamente os nomes ou endereços uns dos outros. Desde que um serviço de evento, ou um espaço de tuplas, sejam persistentes, os componentes individuais podem entrar, sair e ser substituídos. É preciso cuidado para manter a operação correta do sistema global, mas pelo menos os programadores dos componentes não precisam gerenciar associações individuais com pares que desaparecem rotineiramente.

Um exemplo de sistema cliente-servidor que usa associação indireta é o Intentional Name System (INS) [Adjie-Winoto *et al.* 1999]. Os componentes emitem requisições que especificam os atributos do serviço exigido, a operação a ser invocada e seus parâmetros. Os componentes não precisam especificar o nome ou o endereço de uma instância do serviço solicitado, pois o INS direciona automaticamente a operação e os parâmetros para uma instância do serviço apropriada – por exemplo, local – que corresponda aos atributos solicitados. Como sucessivas operações direcionadas para a mesma especificação de atributo poderiam ser manipuladas por diferentes componentes servidores, o INS presume que esses servidores são sem estado (*stateless*), ou que replicam seu estado usando uma das técnicas descritas no Capítulo 18.

Isso leva a uma pergunta geral: como os programadores devem gerenciar o estado em um sistema volátil? As técnicas de replicação do Capítulo 18 presumem uma redundância dos recursos, os quais podem não estar disponíveis em um sistema volátil – pelo menos, não continuamente. As técnicas de replicação também exigem comunicação extra que pode não ser praticável, devido ao consumo de energia e à degradação do desempenho associados.

O algoritmo *Paxos* ou *Part-time Parliament*, de Lamport [1998], oferece uma maneira de se chegar a um acordo distribuído, a despeito da volatilidade – presume-se que os processos participantes desaparecem e reaparecem, regular e independentemente. Entretanto, o algoritmo depende de cada processo ter acesso ao seu próprio meio de armazenamento persistente.

Em contraste, algumas implementações usam o que se denomina de *soft state* para fornecer garantias de consistência mais flexíveis, porém ainda úteis, mesmo na ausência de um armazenamento persistente continuamente disponível. Clark [Clark 1988] introduziu a noção de *soft state* como uma maneira de gerenciar a configuração de roteadores da Internet a despeito de falhas. O conjunto de roteadores é um sistema volátil que precisa continuar funcionando, mesmo que nenhum roteador esteja sempre disponível. A definição de *soft state* tem sido um assunto de discussões [Raman e McCanne 1999], mas, falando de modo geral, os dados fornecem dicas sobre o estado do sistema (porém, eles podem ser antigos e, portanto, não devem ser considerados precisos); e mais importante, os dados que compõem o *soft state* são atualizados automaticamente. Alguns sistemas de descoberta (Seção 19.2) exemplificam o uso de *soft state* para gerenciar o conjunto de entradas de registro do serviço. Primeiro, as entradas são apenas palpites – pode haver uma entrada para um serviço que desapareceu. Segundo, as entradas são atualizadas automaticamente pelo anúncio dos serviços em *multicast* – para adicionar novas entradas e manter as já existentes atualizadas.

19.3.3 Resumo e perspectivas

Esta seção descreveu modelos de interação entre componentes em sistemas voláteis. Se cada espaço inteligente desenvolvesse sua própria interface de programação, as vantagens da mobilidade seriam limitadas. Se um componente não se originasse em determinado espaço inteligente, mas se movesse para lá, a única maneira pela qual ele poderia interagir com os serviços dentro do espaço inteligente seria por intermédio de um modo de adaptar sua interface espontaneamente à de sua vizinhança. Obter isso exigiria um suporte em tempo de execução muito sofisticado, que ainda não é realizado fora de uns poucos exemplos em atividades de pesquisa.

Uma estratégia diferente, descrita anteriormente por meio de vários exemplos, é a programação orientada a dados. Por um lado, a Web tem mostrado a capacidade de ampliação e aplicação em massa desse paradigma. Por outro, não há nenhuma panaceia que resolva todos os problemas da interação de sistemas voláteis. Os sistemas orientados a dados trocam

o acordo sobre o conjunto de funções em uma interface pelo acordo sobre os tipos de dados passados como argumentos para essas funções. Embora XML (veja a Seção 4.3.3) às vezes seja apregoada como uma maneira de facilitar a interação de dados, permitindo que eles sejam “autodescritivos”, na verdade ela apenas fornece uma estrutura para expressar estrutura e vocabulário. Em si mesma, XML nada tem para contribuir com o que é um problema de semântica. Alguns autores consideram a “Web semântica” [www.w3.org XX] uma maneira de obter interoperabilidade entre máquinas, sem interpretação humana.

19.4 Percepção e reconhecimento de contexto

As seções anteriores se concentraram nos aspectos de volatilidade dos sistemas móveis e ubíquos. Esta seção se concentrará em outra característica importante desses sistemas: o fato de serem integrados com o mundo físico. Especificamente, ela considerará as arquiteturas para processamento de dados coletados a partir de sensores e os sistemas de reconhecimento de contexto que podem responder às suas circunstâncias físicas (percebidas). O sensoriamento ou percepção do local, um importante parâmetro físico, será examinado com mais detalhes.

Como os usuários e os dispositivos que estamos considerando frequentemente são móveis, e como o mundo físico apresenta diferentes oportunidades de interações de locais em diferentes tempos, suas circunstâncias físicas são frequentemente relevantes como determinantes para o comportamento do sistema. O sistema de frenagem com reconhecimento de contexto de um carro poderia ajustar seu comportamento de acordo com a condição da estrada ser escorregadia ou não. Um dispositivo pessoal poderia utilizar recursos detectados automaticamente em seu ambiente, como uma tela para exibição. O sistema *Active Badge* fornece um exemplo histórico: a localização de um usuário – isto é, a localização do crachá que ele portava – foi usada, antes do surgimento dos telefones móveis, para identificar para qual telefone suas ligações deveriam ser direcionadas [Want *et al.* 1992].

O *contexto* de uma entidade (pessoa, lugar ou coisa, seja eletrônico ou não) é um aspecto de circunstâncias físicas, de relevância para o comportamento do sistema. Isso inclui valores relativamente simples, como a localização, a hora, a temperatura, a identidade de um usuário associado, por exemplo, operando um dispositivo, ou a presença e o estado de um objeto como uma tela de exibição. O contexto pode ser codificado e influenciado por meio de regras, como “Se o usuário for Fred e ele estiver em uma sala de reunião do IQ Labs, e se houver uma tela de exibição a 1 m de distância, então mostre as informações do dispositivo na tela – a não ser que um funcionário que não seja do IQ Labs esteja presente”. O contexto também é usado para incluir atributos mais complexos, como a atividade do usuário. Por exemplo, um telefone com reconhecimento de contexto que precisa decidir se vai tocar, exige respostas para perguntas como: o usuário está em um cinema assistindo a um filme ou está falando com seus amigos antes da exibição?

19.4.1 Sensores

A determinação de um valor contextual começa com sensores, que são combinações de *hardware* e/ou *software* usadas para medir valores contextuais. Alguns exemplos são:

Localização, velocidade e orientação: unidades de navegação por satélite (por exemplo, o GPS) para fornecer coordenadas e velocidades globais; acelerômetros para detectar movimento; magnetômetros e giroscópios para fornecer dados de orientação.

Condições do ambiente: termômetros; sensores que medem a intensidade da luz; microfones de intensidade sonora.

Presença: sensores que medem a carga física, por exemplo, para detectar a presença de uma pessoa em uma cadeira ou andando em um pavimento; dispositivos que leem identificadores eletrônicos em etiquetas colocadas próximas a eles, como os leitores *RFID* (*Rádio Frequency IDentification*, uma sub-rede do *NFC*) [Want 2004] ou leitores de sinais infravermelhos, como os usados para perceber etiquetas ativas; *software* usado para detectar pressionamentos de tecla em um computador.

Essas categorias são apenas exemplos de uso de sensores para propósitos em particular. Determinado sensor pode ser usado para diversos propósitos. Por exemplo, a presença de seres humanos pode ser detectada com microfones em uma sala de reunião; a localização de um objeto pode ser determinada pela detecção da presença de sua etiqueta ativa em um local conhecido.

Um aspecto importante de um sensor é o seu modelo de erro. Todos os sensores produzem valores com certo grau de erro. Alguns sensores, por exemplo, os termômetros, podem ser fabricados de modo que os erros caiam dentro de limites de tolerância bem conhecidos e com uma distribuição conhecida (por exemplo, gaussiana). Outros, como as unidades de navegação por satélite, têm modos de erro mais complicados que dependem de suas circunstâncias correntes. Primeiro, eles podem deixar de produzir um valor em certas circunstâncias. As unidades de navegação por satélite são dependentes do conjunto de satélites correntemente acessíveis sob visualização direta. Normalmente, elas não funcionam dentro de prédios, cujas paredes atenuam demais os sinais do satélite para que a unidade funcione. Segundo, o cálculo da localização da unidade depende de fatores dinâmicos, incluindo as posições do satélite, a existência de obstruções próximas e das condições da ionosfera. Mesmo fora de prédios, uma unidade normalmente fornecerá diferentes valores em diferentes ocasiões, para a mesma localização, com apenas uma estimativa de melhor esforço da precisão corrente. Perto de prédios, ou de outros objetos altos, que obstruam ou refletam sinais de rádio, satélites suficientes podem produzir uma leitura, mas a precisão pode ser baixa e a leitura pode ser espúria.

Uma maneira útil de expor o comportamento de erro de um sensor é citar a precisão que ele atinge para uma proporção de medidas especificada, por exemplo: “dentro da área determinada, a unidade de navegação por satélite se mostrou precisa dentro de um raio de 10 m para 90% das medidas”. Outra estratégia é expor um valor de confiança para uma medida em particular – um número (normalmente entre 0 e 1) escolhido de acordo com as incertezas encontradas na dedução da medida.

19.4.2 Arquiteturas de sensoriamento

Salber *et al.* [1999] identificam quatro desafios funcionais a serem superados no projeto de sistemas de reconhecimento de contexto:

Integração de sensores idiossincráticos: alguns dos sensores necessários para a computação com reconhecimento de contexto são incomuns em sua construção e em suas interfaces de programação. Pode ser necessário um conhecimento especializado para implantá-los corretamente no cenário físico de interesse (por exemplo, onde os acelerômetros devem ser colocados para medir os gestos do braço do usuário?); e pode haver problemas de sistema, como a disponibilidade de *drivers* para sistemas operacionais padrão.

Abstração dos dados do sensor: as aplicações exigem abstrações dos atributos contextuais para evitar preocupação com as peculiaridades dos sensores individuais. O

problema é que, mesmo os sensores que conseguem resultados semelhantes, normalmente fornecem dados brutos diferentes. Por exemplo, uma localização pode ser determinada como um par latitude/longitude por um sensor de navegação via satélite, ou vista como o *string* “Café do Joe”, lida de uma fonte de sinal infravermelho próxima. Há necessidade de um acordo sobre o significado dos atributos contextuais e sobre o *software* para inferir esses atributos a partir dos valores brutos do sensor.

As saídas do sensor talvez precisem ser combinadas: a percepção confiável de um fenômeno pode exigir a combinação de valores de várias fontes propensas a erro. Por exemplo, detectar a presença de uma pessoa pode exigir: um microfone (para detectar a voz – mas os sons próximos podem interferir), sensores de pressão no piso (para detectar movimento humano – mas os padrões de “pisada” de diferentes usuários são difíceis de distinguir) e vídeo (para detectar formas humanas – mas é difícil reconhecer características faciais). A combinação de diferentes sensores para reduzir erros é conhecida como *fusão de sensores*. Igualmente, uma aplicação pode necessitar de leituras de sensores de diferentes tipos, para reunir vários atributos contextuais necessários para sua operação. Por exemplo, um *smartphone* com reconhecimento de contexto, que decide se vai apresentar dados em uma tela próxima, necessita de dados de diferentes fontes sensoriais, incluindo aquelas para detectar quem e quais dispositivos estão presentes, e uma ou mais fontes para perceber a localização.

O contexto é dinâmico: normalmente, uma aplicação de reconhecimento de contexto precisa responder às mudanças no contexto e não simplesmente ler um instantâneo dele. Por exemplo, o *smartphone* com reconhecimento de contexto precisa apagar seus dados da tela da sala, se uma pessoa que não for um funcionário entrar nela, ou se Fred (o proprietário do dispositivo) for embora.

Os pesquisadores têm projetado várias arquiteturas de *software* para suportar aplicações de reconhecimento de contexto para tratar de alguns ou de todos os problemas descritos anteriormente. Demos exemplos de arquiteturas para situações nas quais o conjunto de sensores disponíveis é mais ou menos conhecido e estático, e de arquiteturas para determinar atributos contextuais de conjuntos de sensores voláteis – nos quais requisitos não funcionais, como a economia de energia, também se tornam importantes.

Sensoriamente dentro de uma infraestrutura • Os sensores de crachá ativo foram originalmente implantados no laboratório da Olivetti Research, em Cambridge, Inglaterra, em locais fixos e conhecidos do prédio. Uma das aplicações originais de reconhecimento de contexto foi como auxílio para uma telefonista. Se alguém ligasse para, digamos, Roy Want, a telefonista procurava na tela o local onde Roy se encontrava e, assim, passava a ligação para o ramal apropriado. O sistema determinava a localização de Roy, a partir das informações sobre onde o crachá que ele portava foi sentido pela última vez, e exibia essas informações para a telefonista. Os sistemas de processamento de dados de crachá ativo, e outros dados contextuais, foram refinados no Olivetti Research Labs e no Xerox PARC. Harter e Hopper [1994] descrevem um sistema inteiro para processamento de eventos de localização. Schilit *et al.* [1994] também apresentam um sistema que pode processar eventos de detecção de crachá ativo, por meio do que chamam de *ações disparadas pelo contexto*. Por exemplo, a especificação:

Máquina de café Cozinha Chegando “play -v 50 /sounds/rooster.au”

faria uma música ser tocada quando fosse percebido um crachá chegando no sensor montado próximo à máquina de café na cozinha.

Atributos (acessíveis por <i>polling</i>)	Explicação
<i>Localização</i>	Localização que o elemento de contexto está monitorando
<i>Identidade</i>	Identidade do último usuário detectado
<i>Carimbo de tempo (timestamp)</i>	Hora da última saída
Callbacks	
<i>PersonArrives (localização, identidade, carimbo de tempo)</i>	Disparado quando um usuário chega
<i>PersonLeaves (localização, identidade, carimbo de tempo)</i>	Disparado quando um usuário vai embora

Figura 19.5 A classe de elementos de janela *IdentityPresence* do Context Toolkit.

O Context Toolkit [Salber *et al.* 1999] é um exemplo de arquitetura de sistema para aplicativos de reconhecimento de contexto mais gerais do que os baseados em uma tecnologia específica, como os crachás ativos. Foram os projetistas do Context Toolkit que definiram os quatro desafios para sistemas de reconhecimento de contexto, listados anteriormente. Sua arquitetura segue o modelo de como as interfaces gráficas com o usuário são construídas a partir de bibliotecas de elementos de janela (*widgets*) reutilizáveis, as quais ocultam do desenvolvedor de aplicativo a maior parte das preocupações do tratamento com o *hardware* da placa gráfica subjacente. O *toolkit* de contexto define *elementos de contexto*. Esses componentes reutilizáveis de *software* apresentam uma abstração de algum tipo de atributo de contexto, ao mesmo tempo em que ocultam a complexidade dos sensores reais utilizados. Por exemplo, a Figura 19.5 mostra a interface para um elemento de contexto *IdentityPresence*. Ela fornece atributos contextuais para o *software* que faz *polling* nos elementos de contexto e também ativa *callbacks* quando a informação contextual muda (quando um usuário chega ou vai embora). Conforme indicado anteriormente, as informações de presença poderiam ser extraídas de qualquer uma das várias combinações de sensores em determinada implementação; a abstração permite que o programador do aplicativo ignore esses detalhes.

Os elementos de contexto são construídos a partir de componentes distribuídos. *Geradores* adquirem dados brutos de sensores, como os de pressão no piso, e fornecem esses dados para os elementos de contexto. Os elementos de contexto usam os serviços de *interpretadores*, os quais abstraem atributos contextuais dos dados de baixo nível do gerador, extraíndo valores de nível mais alto, como a identidade de uma pessoa que esteja presente, a partir de seus passos característicos. Finalmente, elementos de contexto chamados *servidores* fornecem níveis de abstração, reunindo, armazenando e interpretando atributos contextuais de outros elementos de janela. Por exemplo, um elemento de contexto *PersonFinder* de um prédio poderia ser construído a partir dos elementos de contexto *IdentityPresence* para cada sala do prédio (Figura 19.6), os quais, por sua vez, poderiam ser implementados usando a interpretação do passo, a partir das leituras da pressão no piso, ou do reconhecimento da face, a partir de captura de vídeo. O elemento de contexto *PersonFinder* encapsula a complexidade de um prédio para o programador do aplicativo.

Vista em relação aos quatro desafios apresentados anteriormente, expressos pelos projetistas do Context Toolkit, essa arquitetura acomoda uma variedade de tipos de sensores; ela se destina à produção de atributos contextuais abstratos a partir de dados brutos

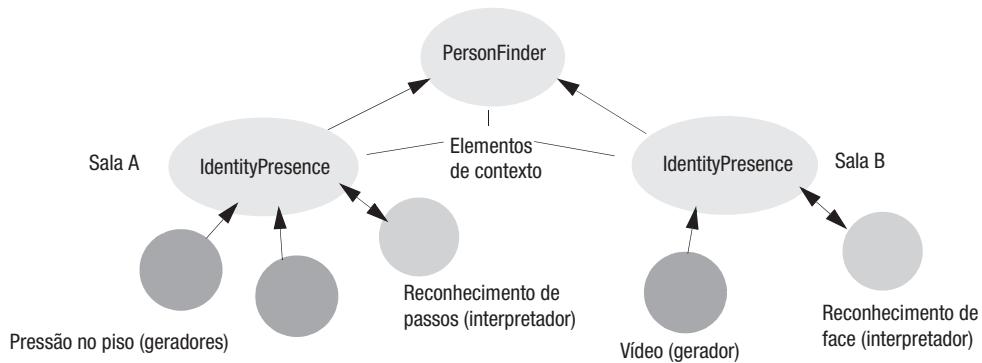


Figura 19.6 Um elemento de contexto *PersonFinder* construído usando elementos de contexto *IdentityPresence*.

dos sensores; e, por intermédio de *polling*, ou de *callbacks*, um aplicativo de reconhecimento de contexto pode ficar sabendo das alterações em seu contexto. Entretanto, para soluções práticas, o *toolkit* apresenta limitações. Ele não ajuda os usuários e programadores a integrar sensores idiossincráticos e também não resolve nenhum dos problemas difíceis inerentes aos processos de interpretação e combinação de um caso específico.

Redes de sensores sem fio • Discutimos arquiteturas para aplicações nas quais o conjunto de sensores é relativamente estável – por exemplo, os sensores são instalados em salas de um prédio, frequentemente com energia externa e com conexões a redes com fio (cabeadas). Agora, veremos os casos em que o conjunto de sensores forma um sistema volátil. Uma *rede de sensores sem fio* consiste em um número (normalmente grande) de pequenos dispositivos de baixo custo, os *nós*, cada um com recursos para sensoriamento, computação e comunicação sem fio [Culler *et al.* 2004]. Trata-se de um caso especial de *rede ad hoc*: os nós são fisicamente organizados de maneira mais ou menos aleatória, mas podem se comunicar por meio de várias passagens intermediárias (*hops*) entre seus pares. Um objetivo de projeto importante dessas redes é funcionar sem nenhum controle global; cada nó se inicializa sozinho, descobrindo seus vizinhos e comunicando-se apenas por meio deles. A Seção 3.5.2 descreveu as configurações *ad hoc* das redes 802.11, mas as tecnologias de potência mais baixa, como a ZigBee (IEEE 802.15.4), são mais relevantes aqui.

Um motivo pelo qual os nós não se comunicam em um único salto (*hop*) com todos os outros nós, mas, em vez disso, comunicam-se diretamente apenas com os nós vizinhos, é que a comunicação sem fio tem um alto consumo de energia, que aumenta com o quadrado do alcance do rádio. O outro motivo importante para restringir o alcance de rádios individuais é a redução da disputa pelo acesso a rede.

As redes de sensores sem fio são projetadas para serem adicionadas em um ambiente natural existente, ou construído, e para funcionar independentemente, isto é, sem haver uma infraestrutura. Em razão de seu alcance de rádio e percepção limitados, os nós são instalados em uma densidade suficiente para tornar provável que a comunicação por vários *hops* seja possível entre qualquer par de nós e que os fenômenos significativos possam ser capturados (percebidos).

Por exemplo, considere dispositivos colocados por toda uma floresta, cuja tarefa fosse monitorar incêndios e outras condições ambientais, como a presença de animais. Esses nós são exatamente como os dispositivos apresentados na Seção 19.1.1. Cada um possui sensores, por exemplo, de temperatura, som e luz; funcionam com baterias; e se

comunicam com outros dispositivos de maneira *peer-to-peer*, por intermédio de comunicação via rádio de curto alcance. A volatilidade deriva do fato de que esses dispositivos podem falhar devido ao esgotamento da bateria ou a acidentes, como os incêndios, e sua conectividade pode mudar devido a falhas de nó (os nós retransmitem pacotes entre outros nós) ou a condições ambientais que afetem a propagação do sinal de rádio.

Outro exemplo é quando os nós são colocados em veículos para monitorar o tráfego e as condições da estrada. Um nó que tenha observado uma condição ruim pode retransmitir informações sobre ela por meio de nós nos veículos que estão passando. Com conectividade global suficiente, esse sistema pode relatar o problema para outros motoristas próximos, que estejam indo nessa direção. Aqui, a volatilidade surge principalmente por causa do movimento dos nós, que muda rapidamente o estado da conectividade de cada nó com outros nós. Esse é um exemplo de *rede ad hoc móvel*.

Em geral, as redes de sensores sem fio são dedicadas a um propósito específico da aplicação, isto significa detectar certos *alarms* – condições de interesse, como incêndios ou condições de estrada ruins. Normalmente, pelo menos um dispositivo mais poderoso, um *nó raiz*, é incluído na rede para prover comunicação de mais longo alcance com um sistema convencional reagindo adequadamente aos alarmes como, por exemplo, chamando os serviços de emergência quando há um incêndio.

Uma estratégia das arquiteturas de *software* para redes de sensores é tratá-las de modo semelhante às redes convencionais, separando a camada de rede das camadas mais altas. Em particular, é possível adaptar algoritmos de roteamento existentes ao grafo formado pelos nós quando eles descobrem dinamicamente que estão conectados por seus enlaces de rádio diretos, com cada nó capaz de atuar como roteador para as comunicações dos outros nós. O roteamento adaptativo, que tenta acomodar a volatilidade da rede, tem sido assunto de muitos estudos, e Milanovic *et al.* [2004] fornece uma visão geral de algumas técnicas.

Entretanto, limitar a preocupação com a camada de rede levanta questões. Primeiro, os algoritmos de roteamento adaptativos não são necessariamente ajustados para baixo consumo de energia (e largura de banda). Segundo, a volatilidade abala algumas das suposições das camadas tradicionais *acima* da camada de rede. Uma alternativa é considerar a estratégia inicial para arquiteturas de *software* para redes de sensores sem fio que é estimulada por dois requisitos principais: economia de energia e operação contínua, a despeito da volatilidade. Esses dois fatores levam a três características principais: processamento na rede, interligação em rede tolerante a rompimento e modelos de programação orientados a dados.

Processamento na rede: não apenas a comunicação sem fio é absolutamente dispendiosa em termos de consumo de energia, como também é relativamente cara em comparação ao processamento. Pottie e Kaiser [2000] calcularam o consumo de energia e descobriram que um processador de propósito geral poderia executar 3 milhões de instruções pela mesma quantidade de energia (3J) usada para transmitir, via rádio, 1 Kbit de dados a 100 m. Portanto, em geral, o processamento é preferível em relação à comunicação: é melhor gastar alguns ciclos do processador para determinar se a comunicação é (ainda) necessária do que transmitir cegamente os dados. Na verdade, é por isso que os nós nas redes de sensores têm capacidade de processamento – caso contrário, eles poderiam consistir simplesmente em módulos de sensoriamento e comunicação, que enviariam os valores capturados para processamento nos nós raízes.

A frase *processamento na rede* se refere ao processamento dentro da rede de sensores; isto é, nos nós da rede. Os nós em uma rede de sensores executam tarefas como agregar ou tirar a média de valores de nós vizinhos, para examinar valores de uma área, em vez de um único sensor; filtrar dados sem interesse ou repetidos; examinar dados para detectar alarmes; e ativar ou desativar sensores, de acordo com os valores que estejam

sendo percebidos. Por exemplo, se sensores de luz de baixa potência indicassem a possível presença de animais (devido à projeção de sombras), então os nós próximos de onde as sombras foram projetadas poderiam ligar seus sensores de potência mais alta, como microfones, para tentar detectar sons do animal. Esse esquema permite que os microfones sejam desligados em outra situação, para economizar energia.

Interligação em rede tolerante a rompimento: o argumento do princípio fim-a-fim (Seção 2.3.3) tem sido um importante princípio para a arquitetura de sistemas distribuídos. Entretanto, nos sistemas voláteis, como as redes de sensores, pode ser que não exista continuamente nenhum caminho do princípio ao fim longo o suficiente para se obter uma operação como a movimentação de um volume de dados em um sistema. Os termos *Disruption Tolerant Networking* (interligação em rede tolerante a rompimento) e *Delay Tolerant Networking* (interligação em rede tolerante a atraso) são usados pelos protocolos para obter transferências de camada mais alta em redes voláteis (e normalmente heterogêneas) [www.dttrg.org]. As técnicas se destinam não apenas às redes de sensores, mas também a outras redes voláteis, como os sistemas de comunicação interplanetária, necessários para a pesquisa espacial [www.ipnsig.org]. Em vez de contar com conectividade contínua entre dois pontos extremos fixos, a comunicação se torna oportunista: os dados são transferidos como e quando puderem ser, e os nós assumem sucessivas responsabilidades de mover dados em um estilo armazenar-e-encaminhar (*store-and-forward*), até que um objetivo do princípio fim-a-fim, como o transporte de um volume de dados, tenha sido atingido. A unidade de transferência entre nós é conhecida como *pacote* [Fall 2003], o qual contém os dados do aplicativo da origem e dados descrevendo como fazer para gerenciá-los e processá-los no ponto extremo e nos nós intermediários. Por exemplo, um pacote pode ser transferido nó a nó, também denominado *hop-by-hop*, usando protocolos de transporte confiáveis; e quando um pacote tiver sido entregue, o nó destinatário assume a responsabilidade por sua distribuição subsequente para o próximo nó (*hop*) – e assim sucessivamente. Esse procedimento não conta com nenhuma rota contínua; além disso, os nós com poucos recursos são desobrigados de armazenar os dados assim que os tiverem transferido para o próximo nó. Para evitar falhas, os dados podem ser encaminhados de forma redundante para vários nós vizinhos.

Modelos de programação orientados a dados: passando para a interação nas camadas de aplicação, foram desenvolvidas técnicas orientadas a dados, incluindo a *difusão direcionada* e o *processamento de consulta distribuído*, descritos em breve, para aplicações de redes de sensores. Essas técnicas reconhecem a necessidade de processamento na rede, incorporando métodos para distribuir o processamento entre os nós. Além disso, as técnicas reconhecem a volatilidade das redes de sensores, eliminando identidades de nó – e todos os outros nomes de componentes, como processos ou objetos, associados a um nó. Conforme discutimos na Seção 19.3.2, qualquer programa que conte com a existência contínua de um nó, ou de um componente, não funcionará de forma robusta em um sistema volátil, pois existe uma chance significativa de que a comunicação com esse nó ou componente se torne impossível.

Na *difusão direcionada* [Heidemann *et al.* 2001], o programador especifica *interesses*, que são declarações de tarefas injetadas no sistema em certos nós denominados de *nós coletores (sinks nodes)*. Por exemplo, um nó poderia expressar um interesse na presença de animais. Cada interesse contém pares atributo-valor, que são os “nomes” dos nós que executarão a tarefa. Assim, os nós são referenciados não por meio de sua identidade, mas das características exigidas para executar a tarefa solicitada, como os valores que estão sendo percebidos em determinado intervalo.

O suporte de execução (*runtime*) propaga os interesses de um nó coletor pela rede, em um processo chamado de *difusão* (Figura 19.7a). O nó coletor encaminha o interesse

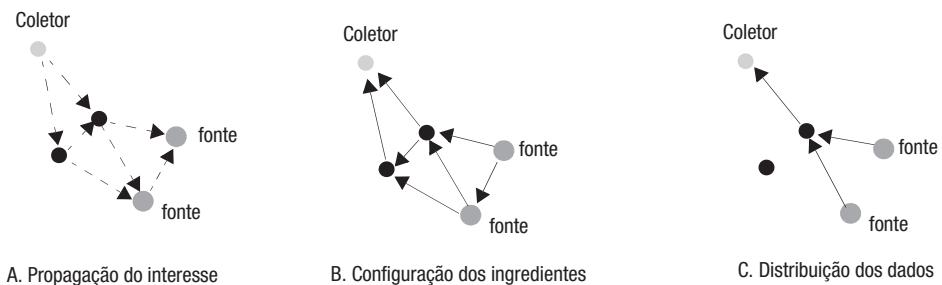


Figura 19.7 Difusão direcionada.

para os nós vizinhos. Qualquer nó que receba um interesse armazena um registro dele, junto às informações necessárias para passar dados para o nó coletor antes de propagá-los na busca de nós que correspondam ao interesse. Um nó *fonte* é aquele que corresponde às características especificadas em pares atributo-valor de acordo com um determinado interesse como, por exemplo, ser equipado com um tipo específico de sensor. Pode haver vários nós fontes para determinado interesse (assim como pode haver vários nós coletores nos quais o interesse foi injetado). Quando o sistema de tempo de execução encontra um nó fonte correspondente, ele passa o interesse para o aplicativo, o qual ativa seus sensores, conforme for exigido, e gera os dados necessários para o nó coletor. O suporte de execução transporta esses dados de volta para o nó coletor, por um caminho inverso constituído dos nós que encaminharam o interesse do nó coletor.

Como, em geral, nenhum nó tem conhecimento *a priori* de qual outro nó pode atuar como fonte, a difusão direcionada pode envolver comunicação redundante considerável. Na pior das hipóteses, a rede inteira poderá ser inundada com um interesse. Entretanto, às vezes o interesse diz respeito apenas a certa região física, como uma área específica de uma floresta. Se os nós sensores souberem suas localizações, então o interesse precisará ser propagado apenas para a área de destino. Em princípio, os nós poderiam ser equipados com receptores de navegação via satélite para esse propósito, embora a cobertura natural, como árvores densas, possa obstruir as leituras.

O fluxo de dados de volta da fonte para o coletor é controlado por *gradientes*, que são pares (*direção, valor*) entre nós, configurados para cada interesse em particular, quando ele é difundido pela rede (Figura 19.7b). A *direção* é aquela para a qual os dados fluem e o *valor* é específico da aplicação, mas pode ser usado para controlar a velocidade do fluxo. Por exemplo, o coletor poderia exigir dados sobre observações de animais apenas certo número de vezes por hora. Pode haver vários caminhos de determinada fonte para determinado coletor. O sistema pode aplicar várias estratégias para escolher um deles, incluindo o uso de caminhos de forma redundante em caso de falha, ou a aplicação de heurísticas para encontrar um caminho de comprimento mínimo (Figura 19.7c).

O programador da aplicação também pode fornecer um *software* chamado de *filtro*, que é executado em cada um dos nós para interceptar o fluxo de dados correspondentes que passam pelo nó. Por exemplo, um filtro poderia suprimir alarmes duplicados de detecção de animais que derivassem de diferentes nós percebendo o mesmo animal (possivelmente o nó entre as fontes e o coletor na Figura 19.7c).

Outra estratégia orientada a dados para programação de redes de sensores é o processamento de consulta distribuído [Gehrke e Madden 2004]. Neste caso, entretanto, é usada uma linguagem como SQL para declarar as consultas que serão feitas coletivamente pelos nós. A execução de uma consulta é normalmente processada no PC do usuário,

ou na *estação de base* fora da rede, levando em conta todos os custos conhecidos, associados ao uso de nós sensores em particular. A estação de base distribui a consulta para os nós da rede, por rotas descobertas dinamicamente, levando em conta os padrões de comunicação que o processamento da consulta impõe, como o envio de dados para tirar a média de pontos de coleta. Assim como acontece na difusão direcionada, os dados podem ser agregados na rede para amortizar os custos da comunicação. Os resultados fluem de volta para a estação de base, para mais processamento.

19.4.3 Percepção de localização

De todos os tipos de percepção usados na computação ubíqua, a percepção da localização tem recebido a maior atenção. A localização é um parâmetro óbvio da computação móvel com reconhecimento de contexto. Parece natural fazer os aplicativos e dispositivos se comportarem de uma maneira que dependa de onde o usuário se encontra, como o telefone com reconhecimento de contexto. No entanto, a percepção da localização tem muitas outras utilizações, desde ajudar usuários na navegação em áreas urbanas ou rurais até determinar rotas de rede pela geografia [Imielinski e Navas 1999].

Os sistemas de percepção da localização são projetados para obter dados sobre a posição de entidades, incluindo objetos e seres humanos, dentro de algum tipo de região de interesse. Aqui, nos concentraremos nas localizações das entidades, mas algumas tecnologias também extraem valores de sua orientação e valores de ordem mais alta, como suas velocidades.

Uma distinção importante, especialmente quando se trata da privacidade, é se um objeto, ou um usuário, determina sua própria localização ou se algo a determina. Este último caso é conhecido como *rastreamento*.

A Figura 19.8 (baseada em uma figura semelhante que aparece em [Hightower e Borriello 2001]) mostra alguns tipos de tecnologias de localização e algumas de suas características principais. Uma das características é o mecanismo usado para inferir uma localização. Às vezes, esse mecanismo impõe limitações sobre onde a tecnologia pode ser implantada, por exemplo, se ela funciona internamente ou ao ar livre, e quais instalações são exigidas na infraestrutura local. O mecanismo também é associado a uma precisão, dada na Figura 19.8 para uma ordem de grandeza. Em seguida, diferentes tecnologias produzem diferentes tipos de dados sobre a localização de um objeto. Finalmente, as tecnologias diferem nas informações, se houver, fornecidas sobre a entidade que está sendo localizada, o que é relevante para as preocupações dos usuários com a privacidade. Mais tecnologias são examinadas em Hightower e Borriello [2001].

O GPS (Global Positioning System) dos Estados Unidos é o exemplo mais conhecido de sistema de navegação via satélite – um sistema para determinar a posição aproximada de um *receptor*, ou *unidade*, a partir de sinais de satélite. Outros sistemas de navegação por satélite são o sistema russo GLONASS e o sistema europeu Galileo. O GPS, que só funciona ao ar livre devido à atenuação do sinal dentro de prédios, é usado rotineiramente em veículos e em dispositivos de mão (*handheld*) para navegação, e cada vez é mais empregado em aplicações menos convencionais, como a distribuição de mídia dependente da localização para pessoas em áreas urbanas [Hull *et al.* 2004]. A posição do receptor é calculada com relação a um subconjunto de 24 satélites que orbitam a terra em seis planos, quatro por plano. Cada satélite orbita a terra cerca de duas vezes por dia, divulgando a hora corrente de um relógio atômico a bordo e informações sobre suas localizações em uma gama de tempos (conforme julgados pelas observações de estações em terra). O receptor, cuja localização vai ser determinada, calcula sua distância a partir de cada um de vários satélites visíveis, usando a diferença entre a hora de chegada do sinal e a hora em que ele foi transmitido – isto é, a hora

<i>Tipo</i>	<i>Mecanismo</i>	<i>Limitações</i>	<i>Precisão</i>	<i>Tipo de dados de localização</i>	<i>Privacidade</i>
GPS	Triangulação de fontes de rádio do satélite	Somente ao ar livre (visibilidade do satélite)	1-10 m	Coordenadas geográficas absolutas (latitude, longitude, altitude)	Sim
Balizamento de rádio	Transmissões de estações de base sem fio (GSM, 802.11, Bluetooth)	Áreas com cobertura sem fio	10 m-1 km	Proximidade de entidade conhecida (normalmente semântica)	Sim
Active Bat	Triangulação de rádio e ultrassom	Sensores montados no teto	10 cm	Coordenadas relativas (sala)	Identidade do morcego (<i>bat</i>) revelada
Ultra Wide Band	Triangulação de recepção de pulsos de rádio	Instalações de receptor	15 cm	Coordenadas relativas (sala)	Identidade da etiqueta revelada
Active Badge	Percepção de sinal infravermelho	Luz do sol ou luz fluorescente	Tamanho da sala	Proximidade de entidade conhecida (normalmente semântica)	Identidade do crachá revelada
Etiqueta de identificação automática	RFID, Near Field Communication, etiqueta visual (por exemplo, código de barras)	Instalações do leitor	1 cm-10 m	Proximidade de entidade conhecida (normalmente semântica)	Identidade da etiqueta revelada
Easy Living	Visão, triangulação	Instalações da câmera	Variável	Coordenadas relativas (sala)	Não

Figura 19.8 Algumas tecnologias de percepção de localização.

codificada no sinal – e uma estimativa da velocidade da propagação de rádio do satélite para a Terra. O leitor calcula, então, sua posição, usando um cálculo trigonométrico conhecido como *triangulação*. Para se obter uma posição, são exigidos pelo menos três satélites visíveis para o receptor. Se apenas três satélites estiverem visíveis, o leitor só poderá calcular sua latitude e longitude; com mais satélites visíveis, a altitude também poderá ser calculada.

Outro método de posicionamento que potencialmente funciona em uma área ampla, pelo menos em regiões densamente povoadas, é identificar *balizas* (*beacons*) próximas, na forma de nós sem fio fixos, com alcance de transmissão limitado. Os dispositivos podem comparar as intensidades do sinal como uma medida de qual baliza está mais próxima. Cada uma das estações de base para telefones móveis (também conhecidas como torres de celulares) tem uma *identificação de célula*; os pontos de acesso 802.11 têm um identificador BSSID (Basic Service Set Identifier). Algumas balizas transmitem informações de identificação; outras são descobertas. Por exemplo, muitos pontos de acesso 802.11 transmitem seus identificadores, enquanto um equipamento Bluetooth fornece seu identificador ao ser descoberto por outro dispositivo.

O balizamento não determina a posição de uma entidade em si, apenas sua proximidade de outra entidade. Se a posição da baliza for conhecida, então a posição da entidade de destino será conhecida dentro do alcance do rádio da baliza. O posicionamento absoluto exige pesquisar o identificador balizado em um banco de dados de locais. Os provedores de telecomunicação revelam informações de posicionamento usando os locais de suas torres de celulares – ou diretamente, ou por meio de terceiros. Algumas empresas, como a Google,

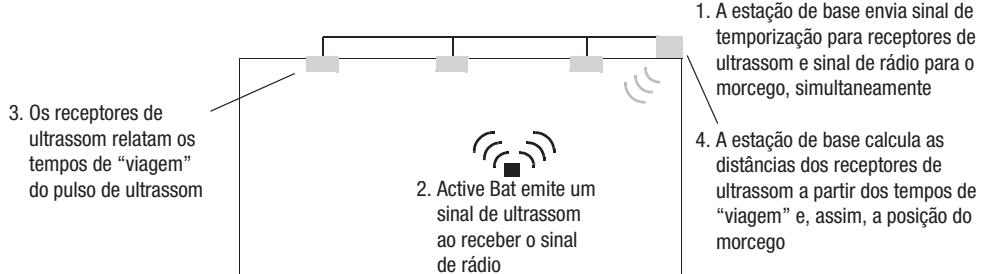


Figura 19.9 Localização de um morcego ativo (*active bat*) dentro de uma sala.

utilizam veículos para identificar sistematicamente áreas de pontos de acesso 802.11, os quais mapeiam usando posicionamento de GPS. A localização de um *smartphone* pode ser determinada dentro de dezenas de metros, quando está ao alcance de um ponto de acesso mapeado (supondo que o ponto de acesso não tenha mudado, o que às vezes ocorre).

A proximidade pode ser uma propriedade útil em si mesma. Por exemplo, usando a proximidade é possível criar aplicativos com reconhecimento de localização que são disparados pelo retorno a um local visitado anteriormente. Por exemplo, um usuário esperando em uma estação de trem poderia criar um alerta lembrando-o de comprar um novo passe mensal quando entrar nas proximidades da estação de trem (isto é, quando seu dispositivo receber o mesmo identificador balizado), no primeiro dia do mês. O Bluetooth, uma tecnologia de rádio alternativa, tem a propriedade interessante de que algumas balizas – por exemplo, aqueles integrados com telefones móveis – são eles próprios móveis. Isso pode ser útil. Por exemplo, passageiros habituais de um trem poderiam receber dados das pessoas com quem viajam frequentemente – “estranhos conhecidos” –, fornecidos pelos seus telefones móveis.

Voltando às formas mais definidas de posicionamento, o GPS extrai as coordenadas *absolutas* (isto é, globais) de um objeto ao ar livre. Em contraste, o sistema Active Bat [Harter *et al.* 2002] foi projetado para produzir a localização de um objeto ou ser humano em interiores, em coordenadas *relativas* – isto é, com relação à sala que contém o objeto (Figura 19.9). O sistema Active Bat fornece uma precisão de cerca de 10 cm. A localização interior relativamente precisa é útil para aplicações como a detecção de qual tela um usuário móvel está próximo e para “teletransportar” a área de trabalho de seu PC para ela, usando o protocolo VNC (veja “Implementações de clientes leves”, na Seção 2.3.2). Um *morcego* (*bat*) é um dispositivo ligado a um usuário ou objeto, cuja localização deve ser encontrada e que recebe sinais de rádio e emite sinais de ultrassom. O sistema conta com uma grade de receptores de ultrassom em locais conhecidos no teto, ligados com fios a uma *estação de base*. Para localizar um morcego, a estação de base emite simultaneamente um sinal de rádio para ele, contendo seu identificador, e um sinal pelo fio para os receptores de ultrassom montados no teto. Quando o morcego com o identificador dado recebe o sinal da estação de base, ele emite um pulso de ultrassom curto. Quando um receptor no teto recebe o sinal da estação de base, ele inicia um temporizador. Como a velocidade da propagação eletromagnética é muito maior do que a velocidade do som, a emissão do pulso de ultrassom e a partida do temporizador são efetivamente simultâneas. Quando um receptor no teto recebe o pulso de ultrassom correspondente (o do morcego), ele lê o tempo decorrido e o encaminha para a estação de base, a qual utiliza uma estimativa da velocidade do som para deduzir a distância do morcego até o receptor. Se a estação de base receber distâncias de pelo menos três receptores de ultrassom não colineares, poderá calcular a posição do morcego no espaço tridimensional.

A *Ultra Wide Band* (UWB) é uma técnica para propagar dados em altas taxas de transmissão (100 Mbps ou mais) em raios de ação curtos (até 10 m). Os bits são propagados em potência muito baixa, mas em um espectro de frequência muito amplo, usando pulsos estreitos – da ordem de 1ns de largura. Sendo conhecidos o tamanho e o formato do pulso, é possível medir tempos de “viagem” com grande precisão. Organizando-se receptores no ambiente e usando triangulação, como nas tecnologias anteriores, é possível determinar as coordenadas de uma etiqueta UWB com uma precisão de cerca de 15 cm. Ao contrário das tecnologias anteriores, os sinais UWB se propagam através de paredes e outros objetos encontrados em um ambiente. Seu baixo consumo de energia é outra vantagem.

Alguns pesquisadores têm experimentado o uso de nós sem fio já existentes, como os pontos de acesso 802.11, para ir além do simples balizamento e tentar inferir a localização de um cliente sem fio medindo a intensidade do sinal com relação a vários pontos de acesso. Na prática, a presença no ambiente de outras entidades que atenuam, refletem ou refratam o sinal significa que sua intensidade não é uma simples função da distância do transmissor. Uma estratégia para lidar com esse problema é o *fingerprinting* (tirar impressões digitais), que determina probabilisticamente as localizações de características da intensidade do sinal medida por todo o espaço. Como parte do projeto Place Lab, Cheng *et al.* [2005] consideram algumas técnicas para determinar localizações a partir de intensidades de sinal, da precisão que pode ser obtida e da quantidade de calibração envolvida.

As tecnologias anteriores fornecem dados sobre a localização física de um objeto: suas coordenadas em uma região física. Uma vantagem de conhecer a localização física é que, por meio de bancos de dados que incluem *sistemas de informação geográfica* (GIS) e *modelos do mundo* de espaços construídos, uma única localização pode ser relacionada com muitos tipos de informações sobre o objeto, ou seu relacionamento com outros objetos. Entretanto, a desvantagem é o trabalho exigido para produzir e manter esses bancos de dados, os quais podem experimentar altas taxas de mudança.

Em contraste, o sistema Active Badge (Seção 19.1) produz a localização semântica de um objeto: o nome ou a descrição da localização. Por exemplo, se um crachá é percebido por um receptor de sinal infravermelho na sala 101, então, a localização desse crachá é determinada como sendo a “Sala 101”. (Ao contrário da maioria dos sinais de rádio, as paredes de uma construção atenuam fortemente os sinais infravermelhos; portanto, é improvável que o crachá esteja fora da sala.) Esses dados não nos informam nada, explicitamente, sobre a localização no espaço, mas fornecem aos usuários informações relacionadas ao seu conhecimento do mundo em que vivem. Em contraste, a latitude e a longitude do mesmo lugar $51^{\circ} 27.010\ N$ $002^{\circ} 37.107\ W$ são úteis para, digamos, calcular distâncias para outros lugares; mas é difícil para seres humanos trabalharem com elas. Note que as balizas (*beacons*) – que invertem as tecnologias Active Badge, colocando o receptor no destino a ser localizado, em vez de colocar na infraestrutura – podem ser usadas para fornecer localizações semânticas ou físicas.

Os *Active Badges* são uma forma especializada de etiquetas de identificação automática: identificadores legíveis eletronicamente, normalmente projetados para aplicações industriais de massa. As etiquetas de identificação automática incluem RFID [Want 2004], Near Field Communication (NFC) [www.nfc-forum.org], glifos ou outros símbolos visuais, como os códigos de barras – especialmente aqueles projetados para serem lidos à distância por câmaras [de Ipiña *et al.* 2002]. Essas etiquetas são anexadas no objeto cuja localização deve ser determinada. Quando lida por um leitor com alcance limitado, e em um local conhecido, a localização do objeto de destino se torna conhecida.

Finalmente, técnicas de visão por computador podem ser usadas para localizar um objeto, como um ser humano observado por uma ou mais câmaras. O projeto *Easy Living*

[Krumm *et al.* 2000] usou algoritmos de visão em *feeds* a partir de várias câmaras. Um objeto de destino pode ser localizado se puder ser reconhecido por uma câmera colocada em um local conhecido. Em princípio, com várias câmaras colocadas em locais conhecidos, as diferenças entre as aparências do objeto em suas imagens podem ser usadas para determinar sua localização com mais precisão. Equipamentos mais especializados combinam imagens no espectro visível com descoberta de alvo infravermelho para determinar a presença de seres humanos e a localização de suas mãos e membros como entradas gestuais, para uso em games, por exemplo.

Conforme demonstrado no estudo de caso do Cooltown (Seção 19.7.2), algumas das tecnologias de localização anteriores – especialmente as etiquetas de identificação automática e as balizas de sinal infravermelho – também podem ser usadas para dar acesso a informações e serviços relativos às entidades nas quais estão anexadas, por meio dos identificadores que tornam disponíveis.

Comparando as tecnologias anteriores com relação à privacidade, a solução do GPS fornece privacidade absoluta: em nenhum ponto na operação do GPS as informações sobre o dispositivo receptor são transmitidas. As balizas podem fornecer privacidade absoluta, mas isso depende de como elas são usadas. Se um dispositivo simplesmente captar as balizas e nunca se comunicar com a infraestrutura de outra forma, ele manterá a privacidade. Em contraste, as outras são tecnologias de rastreamento. Os *Active Bats*, a UWB, os *Active Badges* e os métodos de identificação automática produzem um identificador para a infraestrutura em um local e momento conhecidos. Mesmo que o usuário associado não revele sua identidade, é possível inferi-la. As técnicas de visão do *Easy Living* contam com o reconhecimento de usuários para localizá-los e, assim, a identidade do usuário é revelada muito mais diretamente.

Arquiteturas para percepção de localização • Duas das principais características exigidas para os sistemas de localização são: (1) generalidade com relação aos tipos de sensores usados para a percepção da localização e (2) mudança de escala com relação ao número de objetos a serem localizados e à taxa de eventos de atualização de posicionamento que ocorrem quando os objetos móveis, como pessoas e veículos, mudam suas localizações. Pesquisadores e desenvolvedores têm produzido arquiteturas modestas para percepção da localização – em espaços inteligentes individuais, como salas, prédios ou ambientes naturais cobertos por redes de sensores; e para sistemas de informação geográficas, destinados a cobrir grandes áreas e incluir as localizações de muitos objetos.

A *pilha de localização* [Hightower *et al.* 2002, Graumann *et al.* 2003] tem como objetivo atender ao requisito da generalidade. Ela divide os sistemas de percepção da localização para espaços inteligentes individuais em camadas. A *camada de sensores* contém *drivers* para extrair dados brutos de uma variedade de sensores de localização. Então, a *camada de medidas* transforma esses dados brutos em tipos de medida comuns, incluindo distância, ângulo e velocidade. A *camada de fusão* é a mais baixa disponível para os aplicativos. Ela combina as medidas de diferentes sensores (normalmente, de tipos diferentes), para inferir a localização de um objeto e fornecê-la por meio de uma interface uniforme. Como os sensores produzem dados imprecisos, as inferências da camada de fusão são probabilísticas. Fox *et al.* [2003] examinam algumas das técnicas bayesianas disponíveis. A *camada de arranjos* deduz os relacionamentos entre os objetos, como o fato de eles estarem no mesmo lugar. Acima dessas, estão as camadas para combinar dados de localização com dados de outros tipos de sensores, para determinar atributos contextuais mais complexos, como se um grupo de pessoas localizadas em uma casa estão todas dormindo.

A escalabilidade é uma preocupação maior nos sistemas de informação geográfica. Consultas *espaço-temporais*, como “Quem esteve neste prédio nos últimos 60 dias?”, “Al-

guém está me seguindo?” ou “Quais objetos móveis nesta região correm maior perigo de colidir?”, ilustram essa necessidade. O número de objetos – em particular, o número de objetos móveis – a serem localizados e o número de consultas concorrentes pode ser grande. Além disso, no último desses exemplos de consulta, é exigida sensibilidade em tempo real. A estratégia óbvia para tornar os sistemas de localização capazes de ser escaláveis é dividir a região de interesse em sub-regiões, recursivamente, usando estruturas de dados como *quadtrees*. Tal indexação de bancos de dados espaciais e temporais é uma área de pesquisa ativa.

19.4.4 Resumo e perspectiva

Esta seção descreveu algumas das infraestruturas que foram projetadas para computação com reconhecimento de contexto. Concentramo-nos principalmente nas maneiras pelas quais os sensores são aproveitados para produzir os atributos contextuais dos quais os aplicativos dependem para definir seu comportamento. Vimos as arquiteturas para conjuntos de sensores relativamente estáticos e arquiteturas para redes de sensores altamente voláteis. Finalmente, descrevemos algumas tecnologias para o caso particularmente importante da percepção da localização, algumas das quais estão sendo amplamente utilizadas. A API de geolocalização do W3C (World Wide Web Consortium) [www.w3.org XXIV] inclui suporte para apresentar ao usuário um conteúdo Web específico por meio da percepção automática de sua localização, com um equipamento móvel usando GPS ou a proximidade de uma torre de celular ou de um ponto de acesso 802.11, conforme discutido anteriormente.

Por meio do reconhecimento de contexto, integramos o mundo físico cotidiano com sistemas de computador. Um problema importante que permanece é que, comparados ao entendimento sutil que os seres humanos têm de seu mundo físico, os sistemas que descrevemos são bastante pobres. Não apenas os sensores (pelo menos, aqueles baratos o suficiente para serem amplamente distribuídos) são inevitavelmente imprecisos, como também o estágio final da produção semântica de informações ricas e precisas, a partir dos dados brutos do sensor, é extremamente difícil. O mundo da robótica (que envolve acionamento, um assunto que ignoramos, além da percepção) vem tentando resolver essa dificuldade há muitos anos. Em domínios rigorosamente restritos, como um aspirador de pó doméstico ou a produção industrial, os robôs podem funcionar razoavelmente bem. No entanto, a generalização desses domínios permanece indefinida.

19.5 Segurança e privacidade

Os sistemas voláteis levantam muitas questões tanto sobre a segurança como sobre a privacidade. Primeiro, os usuários e administradores de sistemas voláteis exigem segurança para seus dados e recursos (confidencialidade, integridade e disponibilidade). Entretanto, conforme mencionamos ao descrevermos o modelo de sistemas voláteis, na Seção 19.1, a confiança – a base de toda a segurança – é frequentemente diminuída nos sistemas voláteis. Ela é diminuída porque os principais, cujos componentes interagem espontaneamente, podem ter pouco (se tiverem algum) conhecimento anterior uns dos outros e podem não ter um terceiro participante confiável em comum. Segundo, muitos usuários se preocupam com sua *privacidade* – grosso modo, sua capacidade de controlar a acessibilidade das informações sobre eles mesmos. Contudo, a privacidade está potencialmente mais ameaçada do que nunca, devido à percepção nos espaços inteligentes pelos quais os usuários passam.

Apesar desses fatores desafiadores, as medidas para garantir a segurança e a privacidade das pessoas devem ser leves – parcialmente para preservar a espontaneidade das interações e parcialmente devido às interfaces com o usuário restritas de muitos dispo-

sitivos. As pessoas não desejariam, por exemplo, fazer o *login* em uma caneta inteligente antes de a usarem no escritório de seus anfitriões!

Nesta seção, descreveremos, em linhas gerais, alguns dos principais problemas de segurança e privacidade dos sistemas voláteis. Stajano [2002] fornece um tratamento mais detalhado sobre alguns deles. Langheinrich [2001] examina o assunto da privacidade na computação ubíqua, começando com seu contexto histórico e jurídico.

19.5.1 Fundamentação

A segurança e a privacidade são complicadas nos sistemas voláteis, por problemas relacionados ao *hardware*, como a escassez de recursos, e porque sua espontaneidade leva a novos tipos de compartilhamento de recursos.

Problemas relacionados ao hardware • Os protocolos de segurança convencionais tendem a fazer suposições sobre dispositivos e conectividade que frequentemente não valem nos sistemas voláteis. Primeiro, os dispositivos portáteis, como os *smartphones* e nós sensores, em geral podem ser mais facilmente roubados e falsificados do que os dispositivos como os PCs em salas trancadas. Um projeto de segurança para sistemas voláteis não deve contar com a integridade de nenhum subconjunto de dispositivos que possa ser comprometida. Por exemplo, se um espaço inteligente abrange uma área física suficientemente grande, então uma maneira de ajudar a proteger a integridade global do sistema é tornar necessário que um invasor visite muitos locais dentro dele, mais ou menos ao mesmo tempo, para que seu ataque seja bem-sucedido [Anderson *et al.* 2004].

Segundo, nos sistemas voláteis, às vezes os dispositivos não têm recursos de computação suficientes para a criptografia assimétrica (de chave pública) – mesmo quando se usa criptografia de curva elíptica (Seção 11.3.2). O SPINS [Perrig *et al.* 2002] oferece garantias de segurança para os dados que os nós de baixa potência nas redes de sensores sem fio trocam em um ambiente potencialmente hostil. Seus protocolos usam apenas criptografia de chave simétrica, a qual, ao contrário da criptografia de chave assimétrica, é exequível em tais dispositivos de baixa potência. Entretanto, isso levanta a questão de quais nós em uma rede de sensores sem fio devem compartilhar a mesma chave simétrica. Em um extremo, se todos os nós compartilharem a mesma chave, então um ataque bem-sucedido em um nó comprometeria o sistema inteiro. No outro extremo, se cada nó compartilhar uma chave distinta com cada outro nó, então poderia haver chaves demais para os nós com memória limitada armazenarem. Um meio-termo é fazer com que os nós compartilhem as chaves somente com seus vizinhos mais próximos e contar com encadeamentos de nós mutuamente confiáveis que cifrem as mensagens nó-a-nó (*hop-by-hop*), em vez de usar criptografia de ponta a ponta.

Terceiro, como sempre, a energia é um problema. Não apenas os protocolos de segurança devem ser projetados de forma a minimizar as sobrecargas de comunicação para preservar o tempo de vida da bateria, como também a energia limitada é a base para um novo tipo de ataque de negação de serviço. Stajano e Anderson [1999] descrevem o “ataque de tortura de privação do sono” em nós alimentados por bateria: um invasor pode negar o serviço enviando mensagens espúrias para fazer os dispositivos ficarem sem bateria por desperdiçarem energia ao recebê-las. Martin *et al.* [2004] descrevem mais ataques de “privação do sono”, incluindo fornecer secretamente código ou dados aos dispositivos que os façam desperdiçar energia por meio de processamento. Por exemplo, um invasor poderia fornecer uma imagem GIF animada que parecesse estática para o usuário, mas que na verdade causa redesenho constante.

Finalmente, uma operação desconectada significa que é preferível evitar os protocolos de segurança que contam com o acesso *online* contínuo a um servidor. Por exem-

plo, suponha que máquinas distribuidoras localizadas em paradas para descanso na estrada forneçam certos refrescos gratuitamente, mas somente para passageiros *genuínos* de uma empresa de ônibus específica. Em vez de supor que essa máquina está sempre conectada à sede da empresa para verificar a autorização, é melhor projetar um protocolo pelo meio do qual o dispositivo do usuário (como um telefone) receba um certificado que permita à máquina distribuidora verificar a autorização usando apenas Bluetooth ou outra comunicação de curto alcance [Zhang e Kindberg 2002]. Infelizmente, a ausência de um servidor *online* também significa que um certificado não pode ser revogado e só pode ser construído de forma a expirar depois de determinado tempo – levantando a questão de como os dispositivos *offline* vão contabilizar a passagem do tempo de forma segura.

Novos tipos de compartilhamento de recursos: exemplos de problemas • Os sistemas voláteis originam novos tipos de compartilhamento de recursos que exigem novos projetos de segurança, como os exemplos a seguir.

- Os administradores de um espaço inteligente expõem um serviço acessível aos visitantes por meio de uma rede sem fio – como o envio de *slides* para o serviço de projeção em uma sala de conferência ou o uso de uma impressora em um café.
- Dois funcionários da mesma empresa que se encontram em uma conferência, trocam um documento entre seus telefones móveis, ou outros dispositivos portáteis, sem usar fio.
- Uma enfermeira pega um monitor sem fio de batimentos cardíacos, de uma caixa de dispositivos semelhantes, liga-o a um paciente e o associa ao serviço de registro de dados clínicos desse paciente.

Cada um desses casos é um exemplo de interação espontânea; cada um levanta problemas de segurança e/ou privacidade. Nenhum deles é parecido com os padrões de compartilhamento de recursos normalmente encontrados dentro de intranets protegidas por *firewalls* ou na Internet.

Os serviços de projeção e impressão se destinam apenas aos visitantes, mas a rede sem fio pode ir além dos limites do prédio, de onde invasores poderiam intrometer-se, estragar apresentações ou enviar tarefas de impressão falsificadas. Portanto, os serviços exigem proteção, semelhantemente a um servidor Web destinado apenas para membros de um clube. No entanto, o *login* – digitar um nome de usuário e uma senha – e o procedimento de registro que o precedeu seriam trabalhosos demais; além disso, os usuários podem fazer objeção quanto à privacidade.

A troca de documentos entre dois funcionários é semelhante, de certas maneiras, ao envio de um *e-mail* dentro de uma intranet corporativa. E a interação ocorre por meio de uma rede pública sem fio, em um lugar cheio, principalmente, de pessoas desconhecidas. Em princípio, existe um terceiro participante confiável (a empresa deles), mas, na prática ela pode não estar acessível (eles podem não conseguir obter um sinal de telecomunicações sem fio bom o suficiente para seus telefones no saguão de conferência) ou pode não estar configurada nos dispositivos de todos os usuários.

O que a enfermeira faz é semelhante, de algumas maneiras, ao primeiro exemplo: ela se apropria temporariamente, mas com segurança, de um dispositivo confiável, assim como um visitante poderia se apropriar de um projetor ou de uma impressora. Contudo, o exemplo se destina a mostrar mais enfaticamente a questão da reutilização. Pode haver um número confuso de sensores sem fio usados para diferentes pacientes, em diferentes momentos, e é fundamental fazer e desfazer associações com segurança entre os dispositivos e os respectivos registros de paciente.

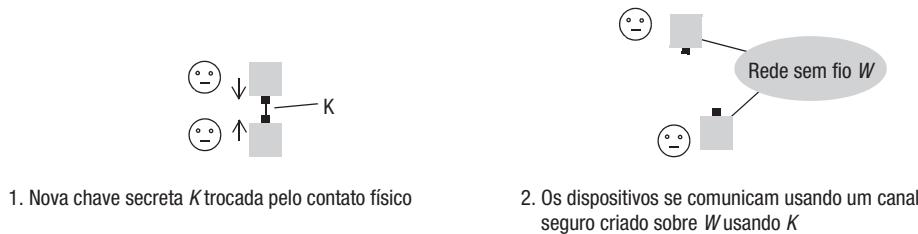


Figura 19.10 Associação segura de dispositivos usando contato físico.

19.5.2 Algumas soluções

Examinaremos agora algumas tentativas de resolver os problemas do fornecimento de segurança e privacidade em sistemas voláteis: a associação espontânea e segura de dispositivos, a autenticação baseada na localização e a proteção da privacidade. As técnicas de segurança que vamos descrever divergem significativamente das estratégias padrão nos sistemas distribuídos. Elas exploram o fato de que os sistemas que estamos considerando são integrados em nosso mundo físico cotidiano, usando evidência física em vez de evidência criptográfica, para capturar propriedades de segurança.

Associação espontânea e segura de dispositivos • Uma questão importante levantada pelos exemplos anteriores é como tornar segura uma associação espontânea entre dois dispositivos conectados por uma rede de rádio sem fio, como a Bluetooth ou a 802.11. Esse é o problema da *associação espontânea e segura de dispositivos*, também conhecido como *problema da associação transiente segura*. O objetivo é criar um canal seguro entre dois dispositivos, trocando uma chave de sessão com segurança entre eles e usando-a para cifrar sua comunicação em uma rede sem fio W . As suposições iniciais são que, como a associação é espontânea, nenhum dos dispositivos (ou seus usuários) compartilha um segredo com o outro, não possui a chave pública do outro e os dispositivos não têm acesso a um terceiro participante confiável. Mesmo que exista um terceiro participante confiável, ele pode estar *offline*. Um invasor pode tentar intrometer-se na rede W e reproduzir e forjar mensagens. Em particular, um invasor pode tentar lançar um ataque do homem no meio (Seção 11.1.1).

Uma solução para esse problema seria permitir que um visitante estabelecesse uma conexão segura com um serviço de projetor ou impressora; os colegas em uma conferência poderiam trocar um documento com segurança entre seus dispositivos portáteis; a enfermeira poderia conectar um monitor de batimentos cardíacos sem fio com segurança em uma unidade de registro de dados na cama do paciente.

Neste contexto, nenhum protocolo de comunicação executando na rede sem fio W permitirá a troca de chave segura por si só; portanto, neste caso, é necessário empregar outra forma de comunicação para negociar os parâmetros de segurança. Em particular, o método padrão para estabelecer uma chave em nível de enlace entre dois dispositivos conectados por Bluetooth conta com ações de um ou mais usuários. Um *string* de dígitos, escolhido em um dispositivo, deve ser digitado por um usuário no outro dispositivo. No entanto, esse método frequentemente não é executado com segurança, pois *strings* de dígitos simples e curtos, como “0000”, tendem a ser usados e podem ser descobertos pelos invasores por meio de busca exaustiva.

Outra estratégia para resolver o problema da associação segura é usar um canal secundário com certas propriedades físicas. Especificamente, a propagação de sinais por esse canal secundário é restrita em ângulo, alcance ou temporização (ou uma combinação

deles). Para um primeiro grau de aproximação, podemos inferir propriedades sobre o remetente ou receptor de mensagens em tais canais, que nos permitam estabelecer associação segura com um dispositivo fisicamente demonstrável, conforme mostraremos em breve. Kindberg *et al.* [2002b] os chamam de *canais fisicamente restritos*, o termo que usaremos aqui; Balfanz *et al.* [2002] se referem a *canais de localização limitada*; Stajano e Anderson [1999] exploraram pela primeira vez tal canal secundário, na forma de contato físico. Apresentamos alguns exemplos desses canais, para os propósitos da associação física de dispositivos, na Seção 19.2.2.

Em um cenário, um dos dispositivos gera uma nova chave de sessão e a envia para o outro por meio de um *canal de recepção restrita*, que fornece certo grau de sigilo – isto é, ele restringe os dispositivos que podem receber a chave. Alguns exemplos de tecnologias de canais de recepção restrita são:

Contato físico. Cada dispositivo tem terminais para conexão elétrica direta [Stajano e Anderson 1999]. Veja a Figura 19.10.

Infravermelho. Feixes de raios infravermelhos podem se tornar direcionais dentro de cerca de 60 graus e são bastante atenuados por paredes e janelas. Um usuário pode “emitir” uma chave para o dispositivo receptor exigido, por uma distância de até cerca de um metro [Balfanz *et al.* 2002].

Áudio. Os dados podem ser transmitidos como modulações de sinal de áudio, como música tocando suavemente em toda uma sala, mas com pouco ou nenhum alcance fora dela [Madhavapeddy *et al.* 2003].

Laser. Um usuário aponta um feixe estreito de raio laser transportando dados para um receptor no outro dispositivo [Kindberg e Zhang 2003a]. Este método permite mais precisão do que as outras técnicas de longo alcance.

Código de barras e câmera. Um dispositivo exibe a chave secreta como um código de barras (ou outra imagem que possa ser decodificada) em sua tela, o qual o outro dispositivo – equipado com uma câmera, como um telefone com câmera – lê e decodifica. A precisão deste método é inversamente relacionada à distância entre os dispositivos.

Em geral, os canais fisicamente restritos fornecem apenas um grau de segurança limitado. Um invasor com um receptor suficientemente sensível pode intrometer-se no sinal infravermelho ou de áudio; um invasor com uma câmera poderosa pode ler um código de barras, mesmo em uma tela pequena. O raio laser está sujeito à dispersão na atmosfera, embora as técnicas de modulação de quantum possam tornar o sinal disperso inútil para um invasor [Gibson *et al.* 2004]. Entretanto, quando as tecnologias são implantadas em circunstâncias apropriadas, os ataques exigem um esforço considerável e a segurança obtida pode ser boa o bastante para os propósitos normais.

Uma segunda estratégia para troca segura de uma chave de sessão é usar um canal restrito para autenticar fisicamente a chave pública de um dispositivo, o qual a envia para o outro dispositivo. Então, os dispositivos empregam um protocolo padrão para trocar uma chave de sessão usando a chave pública autenticada. É claro que esse método presume que os dispositivos sejam poderosos o suficiente para realizar criptografia de chave pública.

A maneira mais simples de autenticar a chave pública é fazer com que o dispositivo a transmita por um canal de *envio restrito*, o qual permite um usuário autenticar a chave como proveniente desse dispositivo físico. Existem várias maneiras de implementar canais de envio restrito convenientes. Por exemplo, o contato físico fornece um canal de envio restrito, pois apenas um dispositivo conectado diretamente pode transmitir através dele. O Exercício 19.14 convida o leitor a considerar quais das outras técnicas de canais

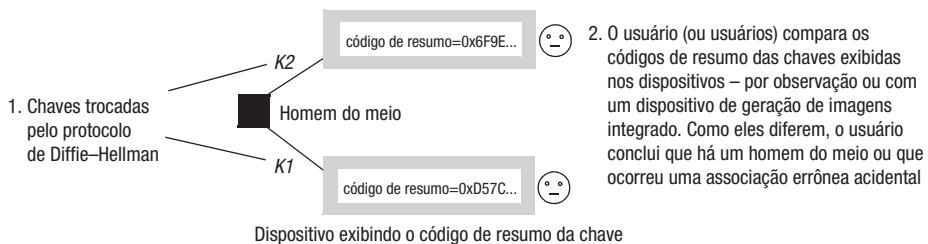


Figura 19.11 Detectando um homem do meio.

de recepção restrita descritas anteriormente também fornecem canais de envio restrito. Além disso, é possível implementar um canal de envio restrito usando um canal de recepção restrita ou vice-versa. Veja o Exercício 19.15.

Uma terceira estratégia utilizando canais fisicamente restritos é a que faz os dispositivos trocarem uma chave de sessão de forma otimista, mas sem segurança, e então usar um canal fisicamente restrito para *validar* a chave – isto é, usar um canal fisicamente restrito para verificar se a chave pertence unicamente à fonte física exigida.

Primeiramente, consideraremos como trocamos uma chave de sessão espontaneamente, mas possivelmente com o principal errado, e passaremos a examinar algumas tecnologias para validar a troca. Se a validação falhar, o processo poderá ser repetido.

Na Seção 19.2.2, descrevemos as técnicas físicas e intermediadas por seres humanos para associar dois dispositivos, como o protocolo de dois botões, no qual os dispositivos trocam seus endereços de rede quando seres humanos pressionam seus botões mais ou menos simultaneamente. É simples adaptar esse protocolo de modo que os dispositivos também troquem chaves de sessão, usando o protocolo de Diffie–Hellman [Diffie e Hellman 1976]. No entanto, como se vê, esse método não é seguro: ainda é possível que grupos separados de usuários associem dispositivos erroneamente, por acidente, executando o protocolo concorrentemente, e que pessoas maldosas lancem ataques do homem no meio.

Uma propriedade do protocolo de Diffie–Hellman é que um homem do meio não pode (exceto com probabilidade desprezível) trocar a mesma chave com cada dispositivo; portanto, podemos validar a associação comparando os códigos de resumo (*hashing*) seguros das chaves obtidas pelos dois dispositivos, após executar o protocolo de Diffie–Hellman (Figura 19.11). As técnicas a seguir nos permitem validar uma chave antes de usá-la. Elas envolvem canais de envio restrito, embora canais de recepção restrita também possam ser usados (veja o Exercício 19.15).

Códigos de resumo exibidos: Stajano e Anderson mostraram que cada dispositivo poderia exibir o código de resumo de sua chave pública como caracteres hexadecimais ou de alguma outra forma que os seres humanos poderiam comparar. Entretanto, argumentaram que esse tipo de envolvimento humano é muito propenso a erros. O método anterior do código de barras seria mais confiável. Esse método é outro exemplo do uso de um canal de envio restrito: o caminho ótico entre a tela de um dispositivo e a câmera do outro, próxima a ele, propaga com segurança o código de resumo seguro do dispositivo exigido.

Ultrassom: um sinal de ultrassom, em combinação com um sinal de rádio, pode ser usado para se inferir a distância e a direção do dispositivo que enviou um código de resumo, usando técnicas semelhantes àquelas utilizadas para o Active Bat, descrito na Seção 19.4.3 [Kindberg e Zhang 2003b].

Passando à consideração de todos os métodos anteriores, eles variam no grau de segurança que fornecem, devido às propriedades dos canais restritos, mas todos são convenientes para a associação espontânea. Nenhum deles exige acesso *online* para nenhum outro componente. Nenhum exige que os usuários se autentiquem ou encontrem nomes ou identificadores eletrônicos para os dispositivos – em vez disso, os usuários recebem uma evidência física sobre quais dispositivos foram associados com segurança. Por suposição, os usuários estabeleceram confiança nesses dispositivos (e em seus usuários). É claro que a segurança obtida é apenas tão boa quanto a fidedignidade dos dispositivos envolvidos: é possível “associar com segurança” um dispositivo com outro que, na verdade, lança um ataque.

Stajano e Anderson [1999, Stajano 2002] usaram canais fisicamente restritos no contexto do protocolo da “ressurreição do patinho” (*resurrecting duckling protocol*). Esse protocolo é relevante para o exemplo do monitor de batimentos cardíacos sem fio, no qual vários dispositivos idênticos devem ser associados várias vezes, com segurança, entre os pacientes. O nome do protocolo se refere ao fato de que os patinhos (reais) nascem no “estado de gravação mental” e ficam sob controle da entidade que primeiro reconhecem (de preferência, suas mães biológicas!) – um processo conhecido como *gravação mental*. Em nosso caso, o dispositivo “patinho” fica sob controle do primeiro dispositivo associado a ele e, então, recusa as requisições de qualquer outra entidade – isto é, qualquer principal que não conheça a chave secreta que o “patinho” trocou com sua “mãe” no ponto de gravação na mente. Uma nova associação só pode ocorrer “matando-se primeiro a alma do patinho” – por exemplo, quando a “mãe” instrui o “patinho” para que reassuma o estado de gravação na mente, no qual sua memória é apagada com segurança. A partir desse ponto, o “patinho” está preparado para ser controlado pelo próximo dispositivo que se associar a ele.

Autenticação baseada na localização • Os exemplos do visitante usando o serviço de projeção de uma sala de conferências e de um usuário imprimindo documentos em um café podem ser vistos das perspectivas dos visitantes e dos administradores. Do ponto de vista dos visitantes, eles podem associar com segurança seus dispositivos ao projetor ou à impressora, usando um dos canais fisicamente restritos anteriores de modo a proteger a privacidade e a integridade de seus dados (embora a impressão de um documento sigiloso em um café possa ser imprudente).

Contudo, os administradores de cada um desses espaços inteligentes têm um requisito adicional: assim como querem que seus visitantes usufruam da segurança, eles precisam implementar o controle de acesso. Apenas pessoas que estejam fisicamente em seus espaços (oradores na sala de conferência, pessoas tomando café no bar) poderão usar seus serviços. E ainda, conforme explicamos, autenticar as identidades dos usuários pode ser inadequado devido aos requisitos de privacidade dos visitantes e à necessidade dos administradores de integrar uma sucessão de usuários e dispositivos que aparecem e desaparecem espontaneamente.

Uma estratégia para a autorização que atende a esses requisitos é basear o controle de acesso na *localização* dos clientes dos serviços, em vez de baseá-lo em suas identidades. Kindberg *et al.* [2002b] descrevem um protocolo para autenticar as localizações dos clientes usando um canal fisicamente restrito que penetra o espaço inteligente, mas não tem alcance além dele. Por exemplo, esse canal poderia ser construído usando-se música tocando em um café ou um sinal infravermelho em uma sala de reuniões. Também existe um *proxy de autenticação de localização* incorporado no espaço inteligente correspondente – isto é, conectado diretamente no mesmo canal restrito – no qual os serviços específicos da localização confiam. Por exemplo, talvez a cadeia de cafés Acme quisesse recompensar

os clientes de toda a sua rede com *downloads* de mídia gratuitos, mas quisesse garantir que ninguém de fora de um café Acme pudesse acessar a mídia, mesmo sendo o serviço de *download* centralizado e conectado à Internet. O protocolo presume que os usuários acessam esses serviços por meio de um navegador Web e usa redirecionamento para que os equipamentos dos visitantes obtenham, de forma transparente, uma prova de localização através do *proxy*. O *proxy* de autenticação de localização verifica se o cliente está onde diz estar e, em caso afirmativo, encaminha sua requisição para o serviço desejado.

Sastray *et al.* [2003] usam canais restritos temporários implementados usando ultrassom para verificar alegações de localização. A base do protocolo é que, como a velocidade do som é fisicamente restrita, apenas um dispositivo que esteja onde diz estar pode transmitir uma mensagem de forma rápida o suficiente, por ultrassom, para um destino nesse local, ao ecoar um número usado uma vez (*nonce*) em um pacote de requisição.

Assim como na associação de dispositivo seguro, a autenticação de localização só torna um sistema seguro até certo ponto. Mesmo que um serviço tenha verificado que um cliente está em um local *genuíno*, esse cliente poderia, contudo, ser mal-intencionado e atuar como um *proxy* para clientes que estivessem em outros lugares.

Proteção à privacidade • A autenticação baseada na localização demonstra uma contrapartida que torna difícil proteger a privacidade em sistemas voláteis: mesmo que o usuário negue sua identidade, ele revela um local que pode ser involuntariamente associado a outros tipos de informações potencialmente identificadoras. É necessário criar salvaguardas em todos os canais pelos quais as informações sobre o usuário possam fluir. Por exemplo, mesmo que um usuário acesse um serviço eletrônico de forma anônima em um café, sua privacidade pode ser violada, caso uma câmera o filme. E se um usuário precisar pagar por um serviço, então terá que fornecer detalhes do pagamento eletrônico, mesmo que faça isso por meio de um terceiro participante. Ele também pode adquirir bens que precisarão ser entregues fisicamente em seu endereço.

Em nível de sistema, a ameaça básica é que, voluntária ou involuntariamente, os usuários fornecem identificadores de vários tipos para espaços inteligentes, quando os visitam e acessam serviços. Primeiro, eles podem fornecer nomes e endereços em acessos ao serviço. Segundo, as interfaces de rede Bluetooth, ou IEEE 802.11, em seus dispositivos pessoais, mantêm, cada uma, um endereço MAC constante que é visível para outros dispositivos, como os pontos de acesso. Terceiro, se os usuários portarem etiquetas, como as RFID (por exemplo, as incorporadas em suas roupas para que as máquinas de lavar inteligentes possam escolher automaticamente um ciclo de lavagem apropriado), então os espaços inteligentes podem identificar essas etiquetas e, assim, levantar informações sobre os usuários ou objetos. As RFIDs são globalmente exclusivas e podem ser usadas tanto para identificar o que o usuário tem com elas (como o tipo de roupas que vestem), como para propósitos de rastreamento.

Qualquer que seja sua origem, os identificadores podem ser associados a um local e a uma atividade em determinado momento; assim, podem ser potencialmente associados às informações pessoais do usuário. Em um espaço inteligente, os usuários podem espionar e coletar os identificadores. Se os espaços inteligentes (ou os serviços neles incorporados) fraudarem, eles poderão rastrear os identificadores pelos locais e inferir movimentos, tudo potencialmente levando à perda da privacidade.

Há pesquisas sendo feitas sobre como tornar o que correntemente são identificadores embutidos (como endereços MAC e RFIDs) em endereços *soft*, que possam ser substituídos de tempos em tempos, para inibir o rastreamento. A dificuldade com os endereços MAC (assim como com os endereços de rede de nível mais alto, como os endereços IP) é que alterá-los causa interrupções da comunicação, as quais representam o custo exigido para se

ter privacidade [Gruteser e Grunwald 2003]. A dificuldade com as RFIDs é que, embora um usuário portando uma RFID não queira ser rastreado pelos sensores “errados”, em geral ele quer que sua etiqueta RFID seja lida por determinados sensores “corretos” (como os de sua máquina de lavar). Uma técnica para tratar desse problema é fazer a etiqueta usar funções de resumo (unilaterais) para substituir o identificador embutido e para gerar o identificador emitido sempre que for lido [Ohkubo *et al.* 2003]. Apenas um terceiro participante confiável, que conheça o identificador exclusivo original da etiqueta, poderá usar um identificador emitido para verificar qual etiqueta foi lida. Além disso, como as etiquetas passam seus identificadores armazenados por uma função de resumo unilateral, antes de emitir-los, os invasores são incapazes (a não ser que possam falsificar a etiqueta) de conhecer o identificador armazenado e, assim, fazer *spoofing* da etiqueta – por exemplo, com a intenção de declarar falsamente que um usuário portando uma etiqueta estava presente na cena de um crime.

Passando para os identificadores de *software* que os clientes fornecem para os serviços, uma estratégia óbvia para ajudar a salvaguardar a privacidade é substituir por um identificador anônimo – escolhido aleatoriamente para toda requisição de serviço – ou usar um *pseudônimo*: um identificador falso que, contudo, é usado consistentemente pelo mesmo principal do cliente por algum período de tempo. A vantagem do pseudônimo sobre um identificador anônimo é que ele permite que um cliente construa um relacionamento de confiança ou uma boa reputação com determinado serviço, sem necessariamente revelar uma identidade verdadeira.

Seria trabalhoso demais para um usuário gerenciar identificadores anônimos ou pseudônimos, de modo que isso normalmente é feito por um componente do sistema chamado *proxy de privacidade*. O *proxy* de privacidade é um componente em que o usuário confia para encaminhar todas as requisições de serviço de forma anônima. Cada um dos dispositivos do usuário tem um canal privativo e seguro com o *proxy* de privacidade. Esse *proxy* substitui identificadores anônimos ou pseudônimos por todos os identificadores verdadeiros nas requisições de serviço.

Um problema do *proxy* de privacidade é que ele é um ponto central de vulnerabilidade: se o *proxy* for atacado com sucesso, toda a utilização de serviço do cliente será revelada. Outro problema é que os *proxies* não ocultam quais serviços o usuário acessa. Um invasor, ou um conjunto fraudulento de invasores, poderia empregar análise de tráfego: isto é, eles poderiam observar correlações no tráfego entre as mensagens que fluem no dispositivo de um usuário em particular e as que fluem em um serviço em particular – examinando fatores como tempo (instantes) e os tamanhos das mensagens.

A *mistura (mixing)* é uma técnica estatística para combinar as comunicações de muitos usuários de tal maneira que os invasores não possam desembaralhar facilmente as ações de um usuário do outro e, assim, ela ajuda a salvaguardar a privacidade dos usuários. Uma aplicação de mistura é a construção de uma rede de sobreposição (*overlay*) de *proxies* que cifram, agregam, reordenam e encaminham mensagens entre eles mesmos por vários nós, de maneira a tornar difícil correlacionar qualquer mensagem que entre na rede de um cliente ou serviço com qualquer mensagem que saia dela, respectivamente para um serviço ou cliente [Chaum 1981]. Cada *proxy* confia e compartilha chaves apenas com seus vizinhos. Seria difícil comprometer a rede sem a conspiração de todos os *proxies*. Al-Muhtadi *et al.* [2002] descrevem uma arquitetura para roteamento de forma anônima das mensagens de um cliente para serviços em um espaço inteligente.

Outra aplicação da mistura é a ocultação das localizações dos usuários por meio da exploração da presença de muitos deles em cada lugar. Beresford e Stajano [2003] descrevem um sistema para ocultar as localizações dos usuários por meio do uso de *zonas de mistura*, que são regiões onde os usuários *não* acessam serviços com reconhecimento

de localização, como os corredores entre salas inteligentes. A ideia é que os usuários mudem as identidades de seus pseudônimos nas zonas de mistura, onde a localização de nenhum usuário é conhecida. Se as zonas de mistura forem suficientemente pequenas, e se pessoas o suficiente passarem por elas, então as zonas de mistura poderão desempenhar uma função parecida com a de uma rede de mistura de anonimato de *proxies*. O Exercício 19.16 considera as zonas de mistura com mais profundidade.

19.5.3 Resumo e perspectiva

Esta seção forneceu uma introdução aos problemas do fornecimento de segurança e privacidade em sistemas voláteis e um breve exame sobre algumas tentativas de soluções, incluindo associação espontânea segura, autenticação baseada na localização e várias técnicas destinadas à proteção da privacidade.

A percepção estendida em uma grande área, os problemas relacionados ao *hardware*, como a escassez de recursos, e a associação espontânea estão na raiz das dificuldades. A percepção aumenta as preocupações dos usuários com a privacidade, pois não apenas seus acessos a serviço podem ser monitorados, como também informações básicas, como suas localizações; e os problemas relacionados ao *hardware* e à espontaneidade desafiam nossa capacidade de fornecer soluções para a segurança. Essa é uma área de pesquisa importante: a segurança e, especialmente, a privacidade podem se transformar em barreiras para o uso de sistemas voláteis.

19.6 Adaptabilidade

Nos sistemas voláteis estudados neste capítulo, os dispositivos são muito mais heterogêneos do que os PCs, em termos de poder de processamento, recursos de E/S, como o tamanho da tela, largura de banda da rede, memória e capacidade de energia. É improvável que a heterogeneidade diminua significativamente, devido aos múltiplos propósitos que temos para os dispositivos. As demandas de transporte e incorporação de dispositivos significam que os mais pobres e os mais ricos em recursos, como energia e tamanho de tela, provavelmente continuarão a diferir muito. (A única tendência positiva global nos recursos está no armazenamento persistente cada vez mais denso, porém financeiramente acessível [Want e Pering 2003].) E, pensando futuramente, o que certamente não vai mudar é a presença de mudança em tempo de execução em si: condições de tempo de execução, como a largura de banda e energia disponíveis provavelmente mudarão dramaticamente.

Esta seção apresenta os sistemas *adaptativos*: que são baseados em um modelo de recursos variáveis e que adaptam seu comportamento, em tempo de execução, à disponibilidade de recursos corrente. O objetivo dos sistemas adaptativos é acomodar a heterogeneidade, permitindo a reutilização de *software* entre contextos que variam em fatores como as capacidades do dispositivo e as preferências do usuário, e acomodar as condições variáveis do recurso em tempo de execução, adaptando o comportamento do aplicativo sem sacrificar suas propriedades fundamentais. No entanto, atingir esses objetivos pode ser extremamente difícil. Esta seção dá uma ideia dessas duas áreas de adaptação.

19.6.1 Adaptação de conteúdo com reconhecimento de contexto

Na Seção 19.3.1, vimos que alguns dispositivos nos sistemas voláteis fornecem conteúdo multimídia uns para os outros. As aplicações multimídia (veja o Capítulo 20) funcionam trocando ou fazendo fluir dados multimídia, como imagens, áudio e vídeo.

Uma estratégia simples para troca de conteúdo seria que os produtores de conteúdo enviassem o mesmo conteúdo, independentemente do dispositivo destino, e que esse dispositivo o representasse apropriadamente, de acordo com suas necessidades e limitações. Na verdade, essa estratégia funciona às vezes, desde que o conteúdo possa ser especificado de forma suficientemente abstrata para que o dispositivo receptor sempre possa encontrar uma representação concreta de acordo com suas necessidades.

Entretanto, verifica-se que, em geral, fatores como limitações da largura de banda e heterogeneidade de dispositivo tornam essa estratégia impraticável. Ao contrário dos PCs, os recursos dos dispositivos nos sistemas voláteis para receber, processar, armazenar e exibir conteúdo multimídia variam muito. Seus tamanhos de tela variam – alguns nem mesmo têm telas –, portanto, enviar imagens de tamanho fixo ou texto com um tamanho fixo de fonte, tudo em um layout fixo, frequentemente levará a resultados insatisfatórios. Os dispositivos podem ter ou não todas as outras formas de E/S que são dadas como certas nos PCs: teclados, microfones, saída de áudio, etc. Mesmo que um dispositivo tenha *hardware* de E/S para representar uma forma de conteúdo como vídeo, ele pode não ter o *software* necessário para determinada codificação (por exemplo, MPEG ou QuickTime) ou pode ter recursos de memória ou de processamento insuficientes para representar mídia com total fidelidade, como resolução de vídeo ou taxa de quadros plena. Finalmente, um dispositivo pode ter todos os recursos para representar determinado conteúdo, mas se a largura de banda para ele for muito pequena, o conteúdo não poderá ser enviado, a não ser que seja convenientemente compactado.

Mais geralmente, o conteúdo que um serviço precisa distribuir para determinado dispositivo é uma função do *contexto*: o produtor de mídia deve levar em conta não apenas os recursos do dispositivo consumidor, mas também fatores como as preferências do usuário do dispositivo e a natureza de sua tarefa. Por exemplo, um usuário em particular pode preferir texto às imagens em uma tela pequena; outro pode preferir saída de áudio à saída visual. Além disso, os itens de um conteúdo que o serviço distribui talvez sejam dependentes de uma interação mais direta com o usuário. Por exemplo, os recursos exigidos em um mapa de determinada região dependerão de o usuário ser um turista procurando atrações ou um trabalhador procurando pontos de acesso de infraestrutura [Chalmers *et al.* 2004]. Em um dispositivo com limitações de tela é mais provável que o mapa seja legível se mostrar apenas um tipo de recurso (atrações ou pontos de acesso, no exemplo).

Seria trabalho demais para os autores de conteúdo multimídia produzir soluções individuais para muitos contextos diferentes. A alternativa é adaptar os dados originais para uma forma conveniente, por meio de programa, selecionando-os, gerando conteúdo a partir deles ou transformando-os – ou qualquer combinação desses três processos. Às vezes, os dados originais são expressos independentemente de como devem ser apresentados – por exemplo, os dados poderiam ser mantidos em XML e *scripts* em eXtensible Style Language Transformations (XSLT) seriam usados para criar formas representáveis para determinado contexto. Em outros casos, os dados originais já são um tipo de dados multimídia, como imagens; nesse caso, o processo de adaptação é conhecido como *transcodificação*. A adaptação pode ocorrer dentro dos tipos de mídia (por exemplo, selecionar dados do mapa ou reduzir a resolução de uma imagem) e entre os tipos de mídia (por exemplo, converter texto em fala ou vice-versa, de acordo com as preferências do usuário, ou de acordo com o fato de o dispositivo consumidor ter uma tela ou saída de áudio).

O problema da adaptação do conteúdo tem recebido muita atenção dos sistemas cliente-servidor na Internet, especialmente na Web. O modelo da Web diz que a adaptação deve ocorrer na infraestrutura rica em recursos – ou no próprio serviço ou em um *proxy* – e não no cliente sem recursos. O protocolo HTTP permite um tipo de negociação de conteúdo

(Seção 5.2): um cliente especifica, em seus cabeçalhos de requisição, as preferências para os tipos MIME do conteúdo que pode aceitar e o servidor pode tentar satisfazer essas preferências no conteúdo que retorna. Contudo, esse mecanismo é limitado demais para a adaptação com reconhecimento de contexto – por exemplo, o cliente pode especificar codificações de imagem aceitáveis, mas não o tamanho da tela do dispositivo. O W3C (World Wide Web Consortium), através de seu grupo de trabalho Device Independence [[www.w3.org XIX](http://www.w3.org/XIX)], e a OMA (Open Mobile Alliance) [www.openmobilealliance.org], estão desenvolvendo padrões por meio dos quais os recursos e as configurações do dispositivo podem ser expressos com alguns detalhes. O W3C produziu o perfil CC/PP (Composite Capabilities/Preferences Profile) para permitir que dispositivos de diferentes classes especifiquem seus recursos e configurações, como tamanho da tela e largura de banda. A especificação de *perfil de agente de usuário* da OMA fornece um vocabulário CC/PP para telefones móveis. Ele pode ser tão detalhado que para um determinado dispositivo ele pode atingir mais de 10 KB de dados. Tal perfil seria dispendioso demais, em termos de largura de banda e energia, para ser enviado junto às requisições; portanto, um telefone móvel envia apenas o URI de seu perfil em um cabeçalho de requisição. O servidor recupera a especificação para fornecer o conteúdo correspondente e armazena a especificação em cache para uso futuro.

Um tipo importante de adaptação para dispositivos com largura de banda limitada é a compactação específica do tipo. Fox *et al.* [1998] descrevem uma arquitetura na qual *proxies* realizam a compactação dinamicamente entre serviços (que podem ou não fazer parte da Web) e clientes. Eles discutem três características principais em sua arquitetura:

- Para acomodar largura de banda limitada, a compactação deve ter perdas, mas ser específica para o tipo de mídia, de modo que a informação semântica possa ser usada para decidir quais recursos de mídia são importantes para manter. Por exemplo, uma imagem pode ser compactada jogando-se fora a informação de cor.
- A transcodificação deve ser realizada dinamicamente, pois as formas de conteúdo previamente preparado estaticamente não fornecerão flexibilidade suficiente para suportar dados dinâmicos e um conjunto cada vez maior de permutações de clientes e serviços.
- A transcodificação deve ser realizada em servidores *proxies* para que os clientes e serviços sejam separados de forma transparente das preocupações com a transcodificação. Nenhum código precisa ser reescrito, e a atividade de transcodificação com uso intenso do computador pode ser realizada em *hardware* com capacidade de mudança de escala conveniente, como *clusters* de computadores montados em um *rack*, para manter as latências dentro de limites aceitáveis.

Quando se trata de sistemas voláteis, como os espaços inteligentes, precisamos rever algumas das suposições feitas para a Web e para outra adaptação na escala da Internet. Os sistemas voláteis são mais exigentes, no sentido de que podem pedir adaptação entre qualquer par de dispositivos associados dinamicamente e, assim, a adaptação não está restrita aos clientes de um serviço particular da infraestrutura. Agora existem potencialmente muito mais provedores cujo conteúdo precisa ser adaptado. Além disso, esses provedores também podem ser pobres demais em recursos para realizar sozinhos certos tipos de adaptação.

Uma implicação é que os espaços inteligentes devem fornecer *proxies* em sua infraestrutura para adaptar conteúdo entre os componentes voláteis que hospedam [Kiciman e Fox 2000; Ponnekanti *et al.* 2001]. A segunda implicação é a necessidade de examinar melhor quais tipos de adaptação de conteúdo podem e devem ser realizados em dispositivos pequenos – em particular, a compactação é um exemplo importante.

Mesmo que exista um poderoso *proxy* de adaptação na infraestrutura, um dispositivo ainda precisa enviar seus dados para esse *proxy*. Discutimos anteriormente como a co-

municação é dispendiosa, comparada com o processamento. Em princípio, pode ser mais eficiente, com relação à energia, compactar os dados antes da transmissão. Entretanto, o padrão de acessos à memória feitos durante a compactação tem forte efeito sobre o consumo de energia. Barr e Asanovic [2003] mostram que, em termos de energia, compactar os dados usando implementações padrão antes de enviá-los – mas que uma otimização cuidadosa dos algoritmos de compactação e descompactação, especialmente com relação aos padrões de acesso à memória – pode levar à economia global de energia, comparada à transmissão de dados não compactados.

19.6.2 Adaptação a sistema de recursos variáveis

Embora os recursos de *hardware*, como o tamanho da tela, sejam heterogêneos entre os dispositivos, pelo menos eles são estáveis e conhecidos. Entretanto, as aplicações também contam com recursos que estão sujeitos à mudança em tempo de execução e que podem ser difíceis de prever, como a energia e a largura de banda da rede disponíveis. Nesta subseção, discutiremos técnicas para lidar com essas alterações de recursos em tempo de execução. Discutiremos o suporte do sistema operacional para aplicações executadas em sistemas voláteis e o suporte necessário à infraestrutura do espaço inteligente para melhorar os recursos disponíveis para estas.

Suporte do sistema operacional para adaptação a recursos voláteis • Satyanarayanan [2001] descreve três estratégias de adaptação. Uma delas é fazer com que as aplicações solicitem e obtenham reservas de recurso. Embora a reserva de recurso possa ser conveniente para as aplicações (Capítulo 20), às vezes é difícil obter garantias de qualidade de serviço satisfatórias nos sistemas voláteis e, às vezes, elas são até impossíveis, como nos casos de esgotamento da energia. Uma segunda estratégia é notificar o usuário sobre alterações na disponibilidade de recursos para que ele possa agir de acordo com a aplicação. Por exemplo, se a largura de banda se tornar baixa, o usuário de um reproduutor de vídeo pode operar um controle deslizante para trocar a taxa de quadros ou a resolução. A terceira estratégia é o sistema operacional notificar a aplicação sobre mudanças nas condições do recurso e esta se adaptar de acordo com suas necessidades específicas.

A arquitetura Odyssey [Noble e Satyanarayanan 1999] fornece suporte de sistema operacional para aplicações que se adaptam às mudanças nos níveis disponíveis de recursos, como a largura de banda da rede. Por exemplo, se a largura de banda cai, um reproduutor de vídeo pode trocar para um fluxo de vídeo com menos cores ou ajustar a resolução ou, ainda, a taxa de quadros. Na arquitetura Odyssey, as aplicações gerenciam tipos de dados como vídeo e imagens e, quando as condições do recurso mudam, eles ajustam a *fidelidade* – a qualidade específica do tipo – com que esses dados são representados. Um componente de sistema chamado *viceroy* divide os recursos totais do dispositivo entre cada uma das várias aplicações que estão sendo executadas nele. Em dado momento, cada aplicação é executada com uma *janela de tolerância* para as alterações nas condições do recurso. A janela de tolerância é um intervalo de níveis de recurso escolhido para ser grande o suficiente para ser realista em termos de variações de recurso reais, mas estreito o suficiente para que a aplicação se comporte de forma mais ou menos consistente dentro desses limites. Quando o *viceroy* precisa mudar os níveis de recurso para um valor fora da janela de tolerância, ele chama a aplicação, a qual, então, reage de acordo. Por exemplo, um reproduutor de vídeo poderia mudar para preto e branco, caso a largura de banda atingisse um nível muito baixo; acima disso, ele poderia ajustar suavemente a taxa de quadros e/ou resolução.

Tirando proveito de recursos do espaço inteligente • *Cyber foraging* [Satyanarayanan 2001; Goyal e Carter 2004; Balan *et al.* 2003] é onde um dispositivo com processamento

limitado descobre um servidor de computação em um espaço inteligente e descarrega nele parte de sua carga de processamento. Por exemplo, converter a fala do usuário em texto é uma atividade que usa muito processamento e que os dispositivos portáteis mal são capazes de realizar satisfatoriamente. Um objetivo é aumentar a capacidade de resposta da aplicação para o usuário – um computador na infraestrutura normalmente tem muitas vezes o poder de processamento de um dispositivo portátil. Porém, isso também é um exemplo de adaptação com reconhecimento de energia: é possível economizar as baterias do dispositivo portátil alocando trabalho para o servidor de computação mais potente.

Existem requisitos desafiantes associados à *cyber foraging*. A aplicação precisa ser decomposta de tal maneira que parte dela possa ser processada eficientemente em um servidor de computação, mas a aplicação ainda deve funcionar corretamente (embora mais lentamente ou com fidelidade reduzida) caso nenhum servidor de computação esteja disponível. O servidor de computação deve executar uma parte da aplicação que envolva relativamente pouca comunicação com o dispositivo portátil – caso contrário, o tempo gasto pela comunicação em uma conexão de baixa largura de banda poderia superar os ganhos do processamento. Além disso, o consumo de energia global do dispositivo portátil deve ser satisfatório. Como a comunicação gasta muita energia, não significa automaticamente que ela será economizada utilizando-se um servidor de computação; pode ser que os custos de energia da comunicação com o servidor de computação superem a economia conseguida com a descarga do processamento.

Balan *et al.* [2003] discutem o problema do particionamento de uma aplicação para enfrentar os desafios anteriores e descrevem um sistema para monitorar níveis de recurso (como a disponibilidade do servidor de computação, largura de banda e energia) e, consequentemente, adaptar o particionamento da aplicação entre o dispositivo portátil e os servidores de computação usando uma das opções de decomposição de um pequeno conjunto delas. Por exemplo, considere uma situação na qual um usuário fala em um dispositivo móvel para ditar um texto, o qual então é traduzido para um outro idioma (o do país que ele está visitando). Existem várias maneiras de dividir essa aplicação entre o dispositivo móvel e os servidores de computação, com diferentes implicações na utilização de recurso. Se vários servidores de computação estiverem disponíveis, então os diversos estágios de reconhecimento e tradução poderão ser divididos entre eles; se apenas um servidor de computação estiver disponível, os estágios poderão ser executados em conjunto nessa máquina, ou entre o dispositivo móvel e o servidor de computação.

Goyal e Carter [2004] adotam uma estratégia mais estática para dividir a aplicação, a qual se presume que tenha sido decomposta em programas de comunicação separados. Por exemplo, um dispositivo móvel poderia realizar o reconhecimento de fala de duas maneiras. No primeiro modo, a aplicação é executada inteiramente – e muito lentamente – no dispositivo móvel. No segundo modo, o dispositivo móvel executa apenas a interface com o usuário, a qual envia o áudio digital da voz do usuário para um programa sendo executado em um servidor de computação; esse programa devolve o texto reconhecido para o dispositivo móvel exibir. Seria muito dispendioso, em termos de energia, o dispositivo móvel enviar o programa de reconhecimento para um computador servidor; portanto, em vez disso, o dispositivo envia o URL do programa, a partir do qual o servidor de computação faz o *download* de uma fonte externa e executa.

19.6.3 Resumo e perspectiva

Esta seção descreveu duas categorias principais de adaptação nos sistemas voláteis, as quais são motivadas pela sua heterogeneidade e pela volatilidade de suas condições de tempo de execução. Existe a adaptação de dados multimídia no contexto do consumidor de mídia,

como as características do dispositivo e da tarefa do usuário do dispositivo, e existe a adaptação para os níveis dinâmicos dos recursos do sistema, como a energia e a largura de banda.

Argumentamos que, em princípio, seria melhor produzir *software* adaptativo que pudesse acomodar condições variadas, de acordo com um modelo de variação bem compreendido, do que desenvolver *software* e *hardware* de maneira *ad hoc*, à medida que surgir a necessidade. Entretanto, produzir esse *software* adaptativo é difícil e não existe um acordo geral sobre como fazer isso. Primeiro, os próprios modelos de variação – sobre como prover recursos para a mudança de níveis e sobre como reagir quando eles mudam – podem ser difíceis de inferir com qualquer generalidade. Segundo, existem desafios de engenharia de *software*. Encontrar pontos de adaptação convenientes em *software* já existente exige um conhecimento íntimo de seu funcionamento, e essa atividade pode nem sempre ser bem-sucedida. Entretanto, ao se criar um novo *software* adaptativo desde o início, existe uma série de técnicas da comunidade de engenharia de *software*, como a programação orientada a aspectos [Elrad *et al.* 2001], para ajudar os programadores a gerenciar a adaptação.

19.7 Estudo de caso: Cooltown

O objetivo do projeto Cooltown da Hewlett-Packard [Kindberg *et al.* 2002a; Kindberg e Barton 2001] foi fornecer infraestrutura para a *computação nômade* [Kindberg 1997], um termo que o projeto usou para computação móvel e ubíqua orientada para seres humanos. A palavra “nômade” se refere a seres humanos se movendo entre lugares como sua casa, o trabalho e lojas, à medida que vivem seu cotidiano. A palavra “computação” se refere, aqui, aos serviços fornecidos para esses usuários nômades – e não apenas aos serviços, como correio eletrônico, que podem ser fornecidos em qualquer lugar onde haja conectividade, mas mais particularmente aos serviços integrados com entidades no mundo físico cotidiano, no qual os usuários se movem. Para acessar esses serviços, supõe-se que os seres humanos carregam, ou vestem, dispositivos equipados com sensores e conectados sem fio. Essa descrição frequentemente se refere à forma mais predominante desses dispositivos na época do projeto: os PDAs. Contudo, também foram considerados os *smartphones* e equipamentos mais experimentais, como os relógios inteligentes.

Mais especificamente, o objetivo do projeto era aplicar na computação nômade as lições aprendidas a partir do sucesso da Web, por intermédio de dois objetivos. Primeiro, como a Web fornece um conjunto de recursos no mundo virtual rico e extensível, muito pode ser ganho no mundo físico, por meio da ampliação da sua arquitetura e dos recursos nela existentes. Um objetivo do projeto do Cooltown foi expresso na máxima “tudo tem uma página Web”: cada entidade de nosso mundo físico, seja eletrônica ou não, deve ter um recurso Web associado, chamado de *presença Web*, o qual o usuário pode acessar convenientemente. Uma presença Web poderia ser simplesmente uma página Web contendo informações sobre a entidade, mas poderia ser qualquer serviço fornecido em associação com a entidade. Por exemplo, a presença Web de um produto físico poderia ser um serviço para obter peças de reposição.

O segundo objetivo era obter o alto grau de interação da Web para interações com os dispositivos. Talvez os usuários nômades precisem interagir em lugares que nunca visitaram antes, com presenças Web que nunca tinham encontrado. Não seria aceitável o usuário ter que carregar um novo *software* ou reconfigurar um *software* existente em seus dispositivos portáteis para utilizar esses serviços.

Os aspectos da arquitetura Cooltown que vamos abordar aqui (Figura 19.12) são: presenças Web; *hyperlinks físicos*, que são *links* das entidades físicas para presenças Web e, assim, para os recursos Web com *hyperlinks*; e o *eSquirt*, um protocolo para interação com dispositivos presentes na Web.



Figura 19.12 Camadas do projeto Cooltown.

19.7.1 Presenças Web

O projeto Cooltown considera entidades físicas divididas em três categorias: pessoas, lugares e coisas. A presença Web de uma pessoa, lugar ou coisa é potencialmente qualquer recurso Web escolhido de acordo com uma aplicação específica, mas o projeto Cooltown adota certas funções para as presenças Web de pessoas e lugares. As presenças Web de coisas e pessoas são reunidas nas presenças Web de lugares; portanto, a descrição segue essa ordem.

Coisas: uma “coisa” é um dispositivo ou uma entidade física não eletrônica. As coisas se tornam presentes na Web incorporando servidores Web ou hospedando sua presença dentro de um servidor Web. Se a coisa é um dispositivo, então seu URL é o do serviço que ele implementa. Por exemplo, uma “rádio da Internet” é um dispositivo tocando música que hospeda sua própria presença Web. Um usuário que tenha descoberto o URL de uma rádio da Internet recupera uma página Web com controles que permitem “sintonizar” uma fonte de transmissão da Internet, ajustar suas configurações, como o volume, ou fazer *upload* do arquivo de som do próprio usuário. Contudo, até coisas não eletrônicas podem ter uma presença Web – um recurso Web associado a uma coisa, mas hospedado por um servidor Web em outro lugar. Por exemplo, a presença Web de um documento impresso poderia ser seu documento eletrônico correspondente: em vez de fazer a fotocópia do documento impresso (com uma consequente redução na qualidade), um usuário pode descobrir sua presença Web a partir de um artefato físico – conforme explicaremos na Seção 19.7.2 – e solicitar uma nova impressão. A presença Web de um CD de música poderia ser algum conteúdo digital associado, como clipes de música e fotografias extras, hospedados na coleção de mídia pessoal de seu proprietário.

Pessoas: as pessoas se tornam presentes na Web oferecendo *home pages* globais com serviços para facilitar a comunicação com elas e oferecendo informações sobre seu contexto corrente. Por exemplo, usuários sem telefones móveis poderiam tornar disponível o número do telefone local, por intermédio de sua presença Web – um valor que sua presença Web atualiza automaticamente, à medida que eles mudam. Eles também poderiam escolher para sua presença Web um registro explícito de sua localização corrente por meio de um *link* para a presença Web do lugar no qual estão presentes fisicamente.

Lugares: os lugares são espaços inteligentes, para usar a terminologia deste capítulo. Os lugares se tornam presentes na Web registrando a presença de pessoas e coisas que estão dentro deles – e até a presença Web de lugares aninhados ou relacionados de alguma forma – em um serviço de diretório específico do lugar (Seção 13.3). O diretório de um lugar também contém informações relativamente estáticas, como uma descrição das propriedades físicas e da função do lugar. O serviço de diretório permite que os componentes descubram e, assim, interajam com o conjunto dinâmico de presenças Web dentro do lugar. Ele também é usado como uma fonte de informações sobre o lugar e seu conteúdo, para serem apresentadas para os seres humanos, na forma de páginas Web.

As entradas de diretório para as presenças Web dentro de um lugar podem ser estabelecidas de duas maneiras principais. Primeiro, um serviço de descoberta de rede (Seção 19.2.1) pode ser usado para registrar automaticamente todas as presenças Web implementadas pelos dispositivos dentro da sub-rede do lugar – dispositivos conectados sem fio dentro do lugar ou os servidores de infraestrutura do lugar. Entretanto, embora os serviços de descoberta de rede sejam úteis, eles têm o problema de que nem todas as presenças Web são hospedadas pelos dispositivos na sub-rede do lugar. As presenças Web de entidades físicas não eletrônicas, como os usuários humanos, documentos impressos e CDs de música, que se movem no lugar, ou são levadas para ele, podem estar hospedadas em qualquer lugar. Essas presenças Web precisam ser registradas lá manualmente, ou por intermédio de mecanismos de percepção, em um processo chamado *registro físico* [Barton et al. 2002] – por exemplo, pela percepção de suas etiquetas RFID.

Um serviço chamado *gerenciador de presença na Web* [Debaty e Caswell 2001] gerencia os lugares presentes na Web – por exemplo, todas as salas dentro de um prédio – e também pode gerenciar as presenças de pessoas e coisas. Os lugares são um caso particular da abstração de *contexto* do Cooltown: um conjunto de entidades relacionadas presentes na Web, interligadas para propósitos como a navegação. O gerenciador de presença relaciona cada entidade presente na Web com as presenças de entidades que estão em seu contexto. Por exemplo, se a entidade é uma coisa, as entidades relacionadas poderiam ser a pessoa que a transporta e o lugar onde ela está localizada. Se a entidade é uma pessoa, suas entidades relacionadas poderiam ser as coisas carregadas por ela, o lugar onde a pessoa está correntemente localizada e, possivelmente, as pessoas que a rodeiam.

19.7.2 Hyperlinks físicos

As presenças Web são recursos Web como qualquer outro; portanto, as páginas Web podem conter *links* de texto ou imagem para presenças Web como todos os outros *links*. Contudo, nesse modelo de *link* padrão, o usuário se depara com a presença Web de uma pessoa, lugar ou coisa por intermédio de uma fonte de *informação*: a página Web. O projeto Cooltown permite, adicionalmente, que os seres humanos acessem diretamente as presenças Web a partir de sua fonte física: a saber, a pessoa, lugar ou coisa física específica que encontraram em suas movimentações diárias pelo mundo físico.

Um *hyperlink físico* é qualquer meio pelo qual um usuário pode recuperar o URL da presença Web de uma entidade, a partir da própria entidade física ou de seus vizinhos imediatos. Consideraremos agora as maneiras de implementar os *hyperlinks* físicos. Primeiro, considere a marcação em HTML de um *link* típico em uma página Web, digamos:

```
<a href="http://cdk4.net/ChopSuey.html">Tela Chop Suey de Hopper</a>
```

Isso vincula o texto “Tela Chop Suey de Hopper” de uma página Web a uma página no endereço <http://cdk4.net/Hopper.html>, sobre o quadro Chop Suey de Edward Hopper, que está sendo referenciado. Agora, considere a questão de como um visitante de um museu que encontrasse o quadro poderia “clicar” na tela para obter informações sobre ela no navegador de seu telefone móvel, PDA ou outro dispositivo portátil. Isso exigiria uma maneira de descobrir o URL do quadro a partir da configuração física da própria tela. Uma maneira seria escrever o URL na parede, perto do quadro, para que o usuário pudesse digitá-lo no navegador de seu dispositivo, mas isso seria deselegante e trabalhoso.

Em vez disso, o Cooltown utiliza o fato de que os usuários têm sensores integrados em seus dispositivos, e os projetistas investigaram duas estratégias principais para descobrir os URLs das entidades por intermédio desses sensores: *percepção direta* e *percepção indireta*.

Percepção direta: neste modelo, o dispositivo do usuário percebe um URL diretamente a partir de uma etiqueta (uma etiqueta de “identificação automática”) ou baliza anexada à entidade de interesse ou próxima a ela (veja a Seção 19.4.3). Uma entidade relativamente grande, como uma sala, poderia ter várias etiquetas ou balizas, localizadas em lugares bem visíveis. Uma etiqueta é um dispositivo, ou artefato passivo, que apresenta o URL quando o usuário coloca o sensor do dispositivo perto dela. Por exemplo, um telefone com câmera poderia, em princípio, realizar o reconhecimento ótico de caracteres no URL escrito em uma tabuleta ou poderia ler o URL codificado em um código de barras bidimensional. Uma baliza, por outro lado, emite regularmente o URL da entidade, normalmente por meio de raios infravermelhos (direcionais), em vez de sinais de rádio, que normalmente são onidirecionais e, assim, podem levar a uma ambiguidade quanto a qual URL pertence a qual entidade.

Em particular, o projeto Cooltown desenvolveu balizas na forma de pequenos dispositivos (a alguns centímetros um do outro) que emitem um *string* a cada poucos segundos por meio de raios infravermelhos, usando um protocolo sem conexão de tentativa única (Figura 19.13a). O *string* emitido é um documento do tipo XML, consistindo no URL da presença Web da entidade e um título curto. Muitos dispositivos portáteis, disponíveis na época do projeto, como os telefones móveis e PDAs, tinham trancetores de sinal infravermelho integrados e, portanto, eram capazes de receber esses *strings*. Quando um programa cliente recebe o *string*, ele pode, por exemplo, fazer o navegador do dispositivo ir diretamente para o URL recebido ou criar um *hyperlink* para o URL adquirido a partir do título recebido e adicionar esse *hyperlink* em uma lista de *hyperlinks* recebidos, na qual o usuário pode clicar quando quiser.

Percepção indireta: a percepção indireta se dá onde o dispositivo do usuário obtém um identificador a partir de uma etiqueta ou baliza, as quais são pesquisadas para se obter um URL. O dispositivo de percepção conhece o URL de um *resolvedor* – um servidor de nomes que mantém um conjunto de vínculos de identificadores para URLs (um contexto de atribuição de nomes, na terminologia do Capítulo 13) e que retorna o URL vinculado ao identificador dado [Kindberg 2002]. De preferência, o espaço de nomes usado para identificadores de entidade deve ser suficientemente grande para permitir que cada entidade física tenha um identificador exclusivo e, assim, elimine a possibilidade de ambiguidade. Entretanto, em princípio, poderiam ser usados identificadores locais, desde que fossem pesquisados apenas usando um resolvedor local – caso contrário, seria possível obter resultados espúrios, pois alguém poderia ter usado o mesmo identificador para uma entidade diferente.

Às vezes, a percepção indireta é usada porque as restrições na tecnologia da etiqueta significam que a percepção direta do URL é impossível. Por exemplo, os códigos de barras lineares não têm capacidade suficiente para armazenar um URL arbitrário; as etiquetas RFID baratas armazenam apenas um identificador binário de comprimento fixo. Em cada caso, o identificador armazenado precisa ser pesquisado para se obter o URL da presença Web.

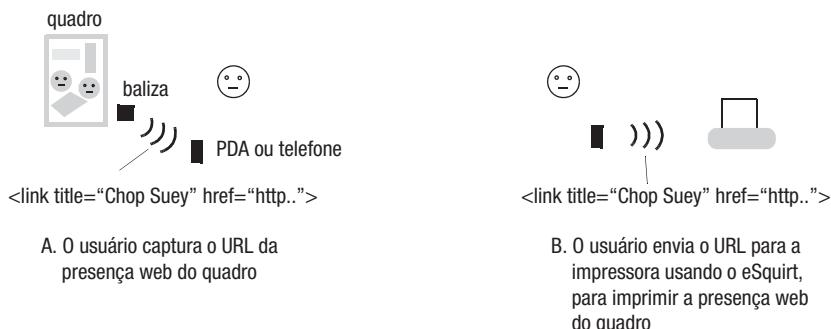


Figura 19.13 Capturando e imprimindo a presença Web de um quadro.

Contudo, também existe um motivo positivo para se usar percepção indireta: ela permite que determinada entidade física tenha um conjunto de presenças Web, em vez de apenas uma. Assim como a mesma frase “Tela Chop Suey de Hopper” pode aparecer em várias páginas Web diferentes, determinado quadro físico poderia levar a diferentes presenças Web, de acordo com a escolha do resolvedor. Por exemplo, uma presença Web do quadro poderia ser um *link* para um serviço que imprimisse uma cópia em uma impressora próxima no museu; outra presença Web do mesmo quadro poderia ser uma página fornecendo informações sobre a tela, de um terceiro participante independente, sem nenhuma conexão com o museu.

A implementação da resolução segue a arquitetura da Web no sentido de que cada resolvedor é um *site* da Web independente. O *software* cliente é um navegador melhorado com um *plugin* simples. Os resolvedores fornecem formulários Web contendo um campo em que o cliente preenche como um efeito colateral da percepção, em vez de apresentar o campo para entrada manual do usuário. Quando o usuário, digamos, varre um código de barras, o identificador resultante é automaticamente preenchido no formulário e o cliente envia o formulário para o resolvedor. O resolvedor retorna o URL correspondente, se ele existir.

Como os próprios resolvedores são recursos Web, o usuário navega por eles como faria em qualquer outra página Web [Kindberg 2002] e atualiza o cliente com o que o resolvedor deve usar. Em particular, o usuário pode escolher o URL de um resolvedor local, usando um *hyperlink* físico local. Por exemplo, os administradores do museu poderiam configurar balizas do Cooltown para emitir o URL do resolvedor local, para que os visitantes pudessem usar esse resolvedor para obter presenças Web relevantes dos quadros dentro do museu. Igualmente, se os identificadores dos quadros fossem conhecidos e globalmente estabelecidos, então os visitantes poderiam utilizar outros resolvedores de qualquer parte da Web – por exemplo, um visitante espanhol poderia utilizar o resolvedor de um *site* espanhol de comentários sobre arte, enquanto visitasse um museu na América do Norte.

Finalmente, embora tenhamos mencionado algumas vantagens da percepção indireta em relação à percepção direta, sua principal desvantagem é o tempo de ida e volta extra do cliente até um resolvedor e as consequências da latência e do consumo de energia.

19.7.3 Interação e o protocolo eSquirt

Um método de interação entre um dispositivo de destino presente na Web e o dispositivo portátil do usuário é usar os protocolos padrão da Web. O dispositivo portátil do usuário executa uma operação HTTP GET, ou POST; o dispositivo de destino responde com uma interface com o usuário, na forma de uma página Web, a qual o dispositivo portátil mostra para

o usuário. Retornando a um exemplo anterior, uma rádio presente na Web pode apresentar o URL de seu serviço na Web por meio de uma baliza voltada para seus usuários. O usuário vai para a frente do rádio e aponta o receptor de sinal infravermelho de seu dispositivo portátil – digamos, seu PDA – para ele; o cliente presente em seu PDA recebe o URL da rádio e passa o URL para seu navegador. O resultado é a *home page* da rádio no PDA, com controles para ajustar seu volume, para *upload* e reproduzir arquivos de música do PDA, etc.

Uma impressora presente na Web em um museu poderia se comportar de modo semelhante. O usuário obtém a *home page* da impressora por meio de sua baliza e, assim, pode fazer *upload* do conteúdo na impressora e especificar as configurações da impressora por intermédio da página Web. É claro que dispositivos como as impressoras podem ter interfaces com o usuário físicas, mas aparelhos mais simples, como as telas de fotografia digital, podem não ter, e então uma interface com o usuário virtual é fundamental.

A forma de interação anterior é orientada a dados e, portanto, é independente do dispositivo, como a Web em geral. Como o dispositivo de destino fornece sua própria interface com o usuário, um usuário pode controlá-lo por meio de seu navegador, sem exigir um *software* específico do destino. Por exemplo, um usuário com um arquivo de imagem em seu PDA pode exibi-lo em um dispositivo de representação de imagem arbitrário, seja ele uma impressora ou uma tela de fotografia digital; e um usuário com um arquivo de som em seu PDA pode ouvi-lo em um dispositivo de reprodução de áudio arbitrário, seja ele, por exemplo, uma rádio da Internet ou um sistema de alta-fidelidade “inteligente”.

Um problema desses cenários é que o dispositivo portátil do usuário, relativamente pobre em recursos, pode ter uma conexão sem fio de baixa largura de banda. Suponha que o usuário tenha obtido uma imagem de um quadro em um museu equipado com Cooltown, ou tenha obtido um clipe de áudio de alguém falando sobre o quadro. Em cada caso, as técnicas de adaptação da Seção 19.6 podem ter sido aplicadas devido à limitação dos recursos, como o tamanho da tela e a largura de banda, resultando em versões de fidelidade bastante baixa da imagem ou do clipe de som no dispositivo portátil. Quando o usuário passar a imagem para uma impressora no museu, ou o *clipe* de som para a rádio da Internet em seu quarto de hotel, ele verá versões de baixa qualidade, mesmo que esses dispositivos sejam capazes de representação de alta qualidade e ele possa ter uma conexão de rede com fio de alta largura de banda.

O protocolo *eSquirt* do Cooltown para interação entre dispositivos elimina o problema da baixa fidelidade – e evita o consumo de largura de banda e energia preciosas –, passando o *URL* do conteúdo de um dispositivo para outro, em vez do conteúdo em si. Na verdade, o protocolo é idêntico àquele usado para enviar um URL (e o título) de uma baliza do Cooltown para um dispositivo por meio de sinal infravermelho (Figura 19.13b). O dispositivo passa esse pequeno volume de dados por intermédio do meio infravermelho de baixa energia, e essa é a única operação de rede na qual cada dispositivo está envolvido durante o protocolo *eSquirt* em si. Entretanto, o dispositivo receptor pode atuar, então, como um cliente Web para recuperar o conteúdo usando o URL e executar uma operação como a representação dos dados resultantes.

Por exemplo, um usuário que tenha obtido o URL de uma imagem de um quadro de Hopper a partir de uma baliza ao lado da tela, envia esse URL para uma impressora usando seu PDA compatível com o protocolo *eSquirt*. O protocolo usado pelo *eSquirt* não é confiável, mas, assim como acontece com um controle remoto de TV, se a transmissão falhar, o usuário pressiona o botão “*squirt*” novamente, até que a realimentação na impressora confirme o sucesso da operação. A impressora (ou melhor, o serviço de impressão, que pode ser implementado na infraestrutura) atua, então, como um cliente Web para recuperar o conteúdo do URL – em sua forma de alta fidelidade – e o imprimir.

Assim, o dispositivo portátil do usuário pode atuar como uma área de transferência independente de dispositivo para URLs, semelhante a uma área de transferência independente de uma aplicação para as operações de copiar e colar em uma interface com o usuário na área de trabalho. O usuário emprega o dispositivo para “copiar e colar” URLs entre fontes e nós coletores (*sink*) para transferir conteúdo entre elas.

A independência de dispositivo é a vantagem mais importante do paradigma eSquirt. O protocolo eSquirt sempre funciona da mesma maneira; o que difere é o processamento do URL por parte do receptor. Entretanto, o usuário precisa ter uma ideia razoável de quais combinações de URLs enviados e dispositivos receptores fazem sentido. O projetista do dispositivo receptor deve esperar alguns erros: um usuário pode enviar um URL de um arquivo de áudio para uma impressora por engano. Entretanto, ainda é desaconselhável planejar esses erros antecipadamente. As medidas de prevenção, como a verificação de tipo, podem levar a fenômenos de oportunidades perdidas e interação frágil que identificamos na Seção 19.2.2.

Embora a simplicidade seja uma vantagem do protocolo eSquirt, uma desvantagem é que ele conta com o uso de configurações padrão no dispositivo receptor ou com o uso de controles físicos para a entrada de suas configurações. Isto é, o protocolo eSquirt não permite o paradigma da interação com o qual começamos esta subseção, em que um dispositivo cliente obtém uma interface com o usuário virtual para controlar as configurações de um dispositivo de destino. Por exemplo, após enviar o URL de um arquivo de som, ou uma estação de rádio de fluxo para uma rádio da Internet, como o usuário controlaria o volume de seu dispositivo portátil? O Exercício 19.19 explora esse problema.

19.7.4 Resumo e perspectiva

Descrevemos, em linhas gerais, as principais características da arquitetura Cooltown. O objetivo desse projeto era beneficiar os usuários nômades estendendo a Web, um conjunto virtual de conteúdo ligado por *hyperlinks*, para entidades de seu mundo físico, independentemente dessas entidades terem funcionalidade eletrônica própria. A arquitetura considerou entidades físicas, incluindo pessoas, lugares e coisas, associadas a presenças na Web. Em seguida estão os *hyperlinks* físicos – os mecanismos para identificar o URL da presença Web de uma entidade física. O projeto implementou *hyperlinks* físicos usando balizas de sinal infravermelho, etiquetas como os códigos de barras e RFIDs e transformadores para transformar identificadores em URLs. O suporte para infravermelho tem diminuído com a mudança dos PDAs para os *smartphones*, mas o suporte para leitura de códigos de barras bidimensionais e unidimensionais (e, em alguns países, NFC e sua sub-rede RFID) tem aumentado. Finalmente, o eSquirt é um protocolo de interação independente de dispositivo que diminui a necessidade de dispositivos portáteis de baixa potência estarem no caminho do conteúdo entre fontes e coletores de conteúdo.

O projeto Cooltown atingiu amplamente seus objetivos, mas apenas sob a suposição de que o ser humano está encerrado “em um circuito fechado”. Os seres humanos descobrem fisicamente os serviços associados às entidades que encontram por meio de *hyperlinks* físicos. Os seres humanos também podem ter de registrar fisicamente as presenças Web de entidades não eletrônicas etiquetadas, como os CDs de música, quando são colocadas no contexto de um lugar presente na Web, como uma casa, para que possam ser descobertas eletronicamente. Finalmente, os seres humanos não apenas associam seus dispositivos portáteis a entidades presentes na Web “clicando” em *hyperlinks* físicos, mas também por intermédio do protocolo eSquirt, causando interação independente de dispositivo. O envolvimento do usuário humano contribui para uma grande flexibilidade e elimina

o problema das oportunidades de interação perdidas. Entretanto, o modelo de interação simples do eSquirt não dá aos usuários controle sobre o que um dispositivo receptor faz com um URL que foi enviado a ele.

Um desenvolvimento alternativo seria uma associação automatizada e a interação de entidades presentes na Web. Cada entidade física poderia ter uma instância de tipo uniforme de presença Web, a qual gravaria detalhes da semântica dessa entidade (talvez usando tecnologias da Web semântica), incluindo os relacionamentos entre essas entidades e outras – em particular, o relacionamento entre uma pessoa, ou coisa, presentes na Web e a presença Web do lugar que as contém. Assim, todas as presenças Web dentro de determinado lugar poderiam descobrir-se umas às outras e interagir. Por exemplo, a presença Web da secretaria em uma reunião poderia descobrir documentos dentro de uma sala de reuniões que precisassem ser impressos, quais membros estavam presentes, uma impressora próxima e obter o número exigido de cópias. O gerenciador de presença Web do Cooltown [Debaty e Caswell 2001] iniciou a percepção dessa visão, por meio do gerenciamento uniforme não apenas de lugares, mas de coisas e pessoas presentes na Web que possuem *links* para entidades relacionadas, como o lugar presente na Web onde estão. Por exemplo, quando uma entidade entra em um novo lugar, e lá é registrada, ela é automaticamente atualizada com um *link* para a presença Web de seu novo lugar hospedeiro. De preferência, todos os relacionamentos da entidade seriam estabelecidos por meio de programa, em vez do suporte limitado disponível atualmente. Contudo, a semântica complicada de nosso mundo cotidiano torna provável que realizar uma aplicação como o apoio à reunião automatizado, de uma maneira útil em vez de propensa a erros, ainda está longe de ser concretizada. Nesse meio-tempo, envolver seres humanos no circuito permite que se faça progresso.

19.8 Resumo

Este capítulo apresentou os principais desafios levantados pelos sistemas móveis e ubíquos e poucas soluções, pois não existem muitas disponíveis. A maioria dos desafios se origina do fato de que esses sistemas são voláteis, o que, por sua vez, é devido ao fato de que eles são integrados com nosso mundo físico cotidiano. Os sistemas são voláteis no sentido de que o conjunto de usuários, *hardware* e componentes de *software* em determinado espaço inteligente está sujeito a mudanças imprevisíveis. Os componentes tendem a estabelecer e desfazer associações rotineiramente, tanto quando mudam de um espaço inteligente para outro, como devido a falhas. A largura de banda da conexão pode variar muito com o passar do tempo. Os componentes podem falhar quando as baterias acabam ou por outros motivos. As Seções 19.1 a 19.3 discutiram completamente esses aspectos da volatilidade e algumas técnicas para associar componentes e permitir que eles interajam, apesar das dificuldades da “constante mudança”.

A integração de dispositivos com nosso mundo físico envolvem percepção e reconhecimento de contexto (Seção 19.4) e descrevemos algumas arquiteturas para processamento de dados percebidos. Contudo, ainda resta um desafio que poderíamos descrever como *fidelidade física*: com que precisão um sistema com percepção e computação pode se comportar, de acordo com a semântica sutil que nós, seres humanos, associamos ao mundo físico em que vivemos? Um “telefone com reconhecimento de contexto” se comporta realmente como desejarmos, no sentido de inibir os toques apropriadamente, à medida que nos movemos entre os lugares? A presença Web de um lugar como um quarto de hotel no Cooltown registra realmente todas as entidades presentes na Web que um ser humano diria que estavam nesse lugar – e não, por exemplo, em alguma sala adjacente?

A segurança e a privacidade (Seção 19.5) se destacam bastante na pesquisa sobre sistemas móveis e ubíquos. A volatilidade complica a segurança, pois levanta a questão de base que há para a confiança entre os componentes que desejam estabelecer um canal seguro. Felizmente, a existência de canais fisicamente restritos permite, até certo ponto, que existam canais seguros nos que há um ser humano presente. A integração física tem implicações na privacidade: se o usuário está sendo rastreado para receber serviços com reconhecimento de contexto, pode haver uma séria perda de privacidade. Descrevemos algumas estratégias de gerenciamento de identificador e mencionamos técnicas estatísticas destinadas a reduzir esse problema.

A integração física também significa novos graus de restrição, em termos de fatores como a capacidade de energia do dispositivo, largura de banda sem fio e interfaces com o usuário – um nó em uma rede de sensores tem pouco dos dois primeiros e nada do último; um telefone móvel tem mais de todos os três, mas ainda muito menos do que uma máquina de mesa. A Seção 19.6 discutiu algumas das arquiteturas pelas quais os componentes podem se adaptar às restrições de recurso.

A Seção 19.7 descreveu a arquitetura do projeto Cooltown como um estudo de caso. A arquitetura se distingue por aplicar na computação ubíqua as lições aprendidas com a Web. A vantagem é um alto grau de interação. Porém, como resultado, o projeto do Cooltown se aplica principalmente às situações em que os seres humanos supervisionam as interações.

Finalmente, este capítulo se concentrou nas *diferenças* entre os sistemas móveis e ubíquos e os sistemas distribuídos mais convencionais que aparecem em outras partes deste livro – nos aspectos da volatilidade e da integração física. O Exercício 19.20 convida o leitor a listar algumas das semelhanças e a considerar até que ponto as soluções dos sistemas distribuídos convencionais se aplicam.

Exercícios

- 19.1 O que é um sistema volátil? Liste os principais tipos de mudanças que ocorrem em um sistema ubíquo. *página 821*
- 19.2 Discuta se é possível melhorar o modelo *pull* de descoberta de serviço, enviando consultas por *multicast* (ou *broadcast*) e armazenando na cache as respostas recebidas. *página 831*
- 19.3 Explique o uso de arrendamentos (*leases*) em um serviço de descoberta para enfrentar o problema da volatilidade do serviço *página 831*
- 19.4 O serviço de pesquisa Jini faz as ofertas de serviço corresponderem às requisições do cliente com base nos atributos ou na tipagem de Java. Explique com exemplos a diferença entre esses dois métodos de correspondência. Qual é a vantagem de permitir os dois tipos de correspondência? *página 832*
- 19.5 Descreva o uso de *multicast IP* e nomes de grupo no serviço de descoberta Jini, que permite a clientes e servidores localizarem servidores de *lookup*. *página 832*
- 19.6 O que é programação orientada a dados e em que ela difere da programação orientada a objetos? *página 837*
- 19.7 Discuta a questão de como a abrangência de um sistema de eventos pode e deve ser relacionada com a extensão física de um espaço inteligente no qual ela é usada. *página 838*
- 19.8 Compare e contraste os requisitos de persistência associados aos sistemas de evento e aos espaços de tuplas na infraestrutura de espaços inteligentes. *página 840*

- 19.9 Descreva três maneiras de perceber a presença de um usuário ao lado de uma tela e, assim, motivar algumas características exigidas em uma arquitetura para sistemas de reconhecimento de contexto. *página 844*
- 19.10 Explique e dê a motivação para processamento na própria rede em redes de sensores sem fio. *página 849*
- 19.11 No sistema de localização *Active Bat*, apenas três receptores de ultrassom são usados, por padrão, para obter uma posição tridimensional, enquanto quatro satélites são exigidos para obter uma posição tridimensional na navegação via satélite. Por que existe diferença? *página 854*
- 19.12 Em alguns sistemas de localização, os objetos rastreados abandonam seus identificadores na infraestrutura. Explique como isso pode dar origem a preocupações sobre a privacidade, mesmo que os identificadores sejam anônimos. *página 856*
- 19.13 Muitos nós sensores devem ser espalhados por toda uma região. Os nós devem se comunicar com segurança. Explique o problema da distribuição de chaves e descreva, em linhas gerais, uma estratégia probabilística para distribuir chaves. *página 858*
- 19.14 Descrevemos várias tecnologias que fornecem canais de recepção restrita para uso na associação espontânea e segura de dispositivos. Quais dessas tecnologias também fornecem canais de envio restrito? *página 861*
- 19.15 Mostre como se faz para construir um canal de envio restrito a partir de um canal de recepção restrita e vice-versa. Dica: use um nó confiável conectado com o canal dado. *página 861*
- 19.16 Um grupo de espaços inteligentes está conectado apenas por um espaço entre eles, como um corredor ou quarteirão. Discuta os fatores que determinam se esse espaço interveniente pode atuar como uma zona de mistura. *página 865*
- 19.17 Explique os fatores contextuais a serem levados em conta ao se adaptar conteúdo multimídia. *página 866*
- 19.18 Suponha que um dispositivo possa executar 3 milhões de instruções com a mesma quantidade de energia (3J) usada para transmitir ou receber 1 Kbit de dados a uma distância de 100 m, via rádio. O dispositivo tem a opção de enviar um programa binário de 100 Kbyte para um servidor de computação a 100 m de distância, o qual executará 60 bilhões de instruções e trocará 10.000 de mensagens de 1Kbit com o dispositivo. Se a energia for a única consideração, o dispositivo deve enviar a computação ou executá-la? *página 869*
- 19.19 Um usuário do Cooltown envia o URL de um arquivo de som ou de uma estação de rádio por fluxo para uma rádio da Internet. Sugira uma modificação no protocolo eSquirt que permita ao usuário controlar o volume de seu dispositivo portátil. Dica: considere quais dados extras o dispositivo de envio deve fornecer. *página 877*
- 19.20 Discuta a aplicabilidade nos sistemas móveis e ubíquos das técnicas extraídas das áreas de:
i) sistemas *peer-to-peer* (Capítulo 10);
ii) protocolos de coordenação e acordo (Capítulo 15);
iii) replicação (Capítulo 18). *página 879*

20

Sistemas Multimídia Distribuídos

- 20.1 Introdução
- 20.2 Características de dados multimídia
- 20.3 Gerenciamento de qualidade de serviço
- 20.4 Gerenciamento de recursos
- 20.5 Adaptação de fluxo
- 20.6 Estudos de caso: Tiger, BitTorrent e End System Multicast
- 20.7 Resumo

Os aplicativos multimídia geram e consomem fluxos contínuos de dados em tempo real. Eles são compostos por grandes quantidades de áudio, vídeo e outros elementos de dados vinculados no tempo, e o processamento e a distribuição dos elementos de dados individuais (amostras de áudio, quadros de vídeo) respeitando critérios temporais são fundamentais. Os elementos de dados distribuídos com atrasos não têm valor e normalmente são eliminados.

Uma especificação para um fluxo multimídia é expressa em termos de valores aceitáveis para a taxa com que os dados passam de uma origem para um destino (a largura de banda), para o atraso da distribuição de cada elemento (latência) e com a taxa que os elementos são perdidos ou eliminados. A latência é particularmente importante nos aplicativos interativos. Um pequeno grau de perda de dados de fluxos multimídia frequentemente é aceitável, desde que o aplicativo possa voltar a sincronizar os elementos que vêm após aqueles que foram perdidos.

A alocação e o escalonamento dos recursos para atender às necessidades dos aplicativos multimídia, e outros, é referida como gerenciamento da qualidade do serviço. A alocação da capacidade de processamento, a largura de banda da rede e a memória (para o armazenamento em *buffer* dos elementos de dados distribuídos antecipadamente), tudo isso é importante. Eles são alocados em resposta às exigências da qualidade do serviço dos aplicativos. Uma requisição de qualidade do serviço bem-sucedida gera uma garantia de qualidade do serviço para o aplicativo e resulta na reserva, e no subsequente escalonamento, dos recursos solicitados.

Este capítulo recorre substancialmente a um tutorial de Ralf Herrtwich [1995] e agradecemos a ele pela permissão para usar seu material.

20.1 Introdução

Os computadores modernos podem manipular fluxos de dados contínuos (*streams*), baseados no tempo, como áudio e vídeo digital. Essa capacidade levou ao desenvolvimento de aplicações multimídia distribuídas, como as bibliotecas de vídeo interligadas em rede, a telefonia pela Internet e a videoconferência. Tais aplicações são viáveis com as redes e os sistemas de propósito geral atuais, embora a qualidade do áudio e vídeo resultantes seja frequentemente insatisfatória. Aplicações mais exigentes, como videoconferência de larga escala, produção de TV digital, TV interativa e sistemas de supervisão com vídeo estão além da capacidade das tecnologias de interligação em rede e dos sistemas distribuídos atuais.

Os aplicativos multimídia exigem a distribuição de fluxos dos dados multimídia com restrições temporais para os usuários finais. Os fluxos de áudio e vídeo são gerados e consumidos em tempo real e a distribuição oportuna dos elementos individuais (amostras de áudio, quadros de vídeo) é fundamental para a integridade da aplicação. Em resumo, os sistemas multimídia são sistemas em tempo real: eles precisam executar tarefas e apresentar resultados de acordo com um escalonamento determinado externamente. O grau com que isso é conseguido pelo sistema subjacente é conhecido como *qualidade de serviço* (QoS – *Quality of Service*) usufruída por um aplicativo.

Embora os problemas do projeto de sistemas em tempo real tenham sido estudados antes do advento dos sistemas multimídia, e muitos sistemas em tempo real bem-sucedidos tenham sido desenvolvidos (veja, por exemplo, Kopetz e Verissimo [1993]), geralmente, eles não têm sido integrados em sistemas operacionais e redes de propósito geral. A natureza das tarefas executadas pelos sistemas em tempo real existentes, como na aviação, controle de tráfego aéreo, controle de processos de fabricação e centrais telefônicas, diferem das executadas nos aplicativos multimídia. As primeiras geralmente lidam com volumes de dados relativamente pequenos, e poucas vezes têm *prazos finais rígidos*, mas o não cumprimento de um prazo pode ter consequências sérias ou mesmo desastrosas. Em tais casos, a solução adotada têm sido superestimar os recursos de computação e alocá-los com um escalonamento fixo que sempre garanta o atendimento desses requisitos de pior caso. Esse tipo de solução não existe para a maior parte das aplicações de fluxos multimídia da Internet em computadores de mesa, resultando em uma qualidade de serviço do tipo “melhor esforço”, usando os recursos disponíveis.

A alocação e o escalonamento planejado dos recursos para atender às necessidades dos aplicativos multimídia, e outros, é referida como *gerenciamento de qualidade de serviço*. A maioria dos sistemas operacionais e redes atuais não inclui os recursos de gerenciamento da qualidade de serviço necessários para garantir a qualidade de aplicativos multimídia.

As consequências do não cumprimento dos prazos finais em aplicativos multimídia podem ser sérias, especialmente nos ambientes comerciais, como nos serviços de vídeo sob demanda, aplicações de videoconferência comercial e medicina à distância, mas os requisitos diferem significativamente daqueles de outros aplicativos em tempo real:

- Frequentemente, os aplicativos multimídia são altamente distribuídos e operam dentro de ambientes de computação distribuída de propósito geral. Portanto, eles competem com outros aplicativos distribuídos pela largura de banda da rede e pelos recursos de computação nas estações de trabalho dos usuários e nos servidores.
- Os requisitos de recursos dos aplicativos multimídia são dinâmicos. Uma videoconferência exigirá mais ou menos largura de banda de rede à medida que o número de participantes aumentar ou diminuir. O uso de recursos de computação na estação de trabalho de cada usuário também variará, pois, por exemplo, o nú-

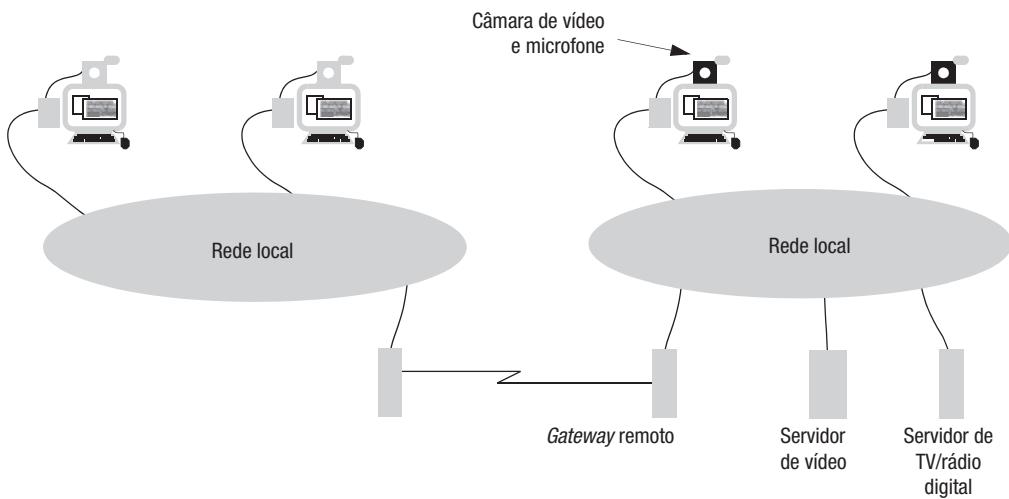


Figura 20.1 Um sistema multimídia distribuído.

mero de fluxos de vídeo que precisa ser exibido varia. Os aplicativos multimídia podem envolver outras cargas variáveis ou intermitentes. Por exemplo, uma aula expositiva multimídia em rede poderia incluir uma atividade de simulação com uso intenso do processador.

- Frequentemente, os usuários desejam contrabalançar os custos de recursos de um aplicativo multimídia com outras atividades. Assim, talvez eles queiram reduzir suas demandas por largura de banda de vídeo em um aplicativo de videoconferência para permitir que uma conversa de voz separada prossiga, ou que o desenvolvimento de um programa, ou de uma atividade de processamento de texto, continue enquanto estão participando de uma conferência.

Os sistemas de gerenciamento de qualidade de serviço se destinam a satisfazer todas essas necessidades, administrando os recursos disponíveis dinamicamente e variando as alocações em resposta às demandas variáveis e às prioridades do usuário. Um sistema de gerenciamento de qualidade de serviço deve supervisionar todos os recursos de computação e comunicação necessários para adquirir, processar e transmitir fluxos de dados multimídia, especialmente onde os recursos são compartilhados entre os aplicativos.

A Figura 20.1 ilustra um sistema multimídia distribuído típico, capaz de suportar uma variedade de aplicações, como videoconferência em *desktop*, fornecimento de acesso a sequências de vídeo armazenadas, transmissão de TV e rádio digital. Os recursos para os quais o gerenciamento de qualidade de serviço é exigido incluem largura de banda de rede, ciclos do processador e capacidade da memória. A largura de banda de disco no servidor de vídeo também pode ser incluída. Adotaremos o termo genérico *largura de banda do recurso* para nos referirmos à capacidade de qualquer recurso de *hardware* (rede, processador central, subsistema de disco) transmitir ou processar dados multimídia.

Em um sistema distribuído aberto, os aplicativos multimídia podem ser iniciados e usados sem organização prévia. Vários aplicativos podem coexistir na mesma rede e até na mesma estação de trabalho. Portanto, surge a necessidade do gerenciamento de quali-

dade de serviço, independentemente da *quantidade total* de largura de banda do recurso ou da capacidade de memória presente no sistema. O gerenciamento de qualidade de serviço é necessário para garantir que os aplicativos possam obter a quantidade de recursos necessária nos momentos exigidos, mesmo quando outros aplicativos estão competindo pelos recursos.

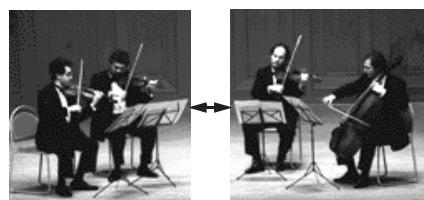
Alguns aplicativos multimídia têm sido implantados, mesmo nos atuais ambientes de computação e de redes baseados em melhor esforço e com pouco suporte à qualidade de serviço. Isso inclui:

Multimídia baseada na Web: são aplicativos que fazem os melhores esforços para a garantir a qualidade de serviço para acessar os fluxos de dados de áudio e vídeo publicados por meio da Web. Eles têm sido bem-sucedidos quando há pouca ou nenhuma necessidade de sincronização dos fluxos de dados em diferentes locais. Seu desempenho varia com a largura de banda e com as latências encontradas nas redes e é prejudicado pela incapacidade dos sistemas operacionais atuais de suportar escalonamento de recursos em tempo real. Contudo, aplicações como *YouTube*, *Hulu* e *BBC iPlayer* oferecem uma demonstração eficaz e popular da exequibilidade da reprodução de fluxos multimídia em computadores pessoais pouco carregados. Elas exploraram o uso extensivo de *buffers* no destino para atenuar as variações na largura de banda e na latência e conseguem uma reprodução contínua e suave de áudio de alta qualidade e de sequências de vídeo de média resolução, embora com um atraso da origem para o destino que pode chegar a vários segundos.

Serviços de vídeo sob demanda: esses serviços fornecem informações de vídeo em forma digital, recuperando os dados de grandes sistemas de armazenamento *online* e enviando-os para a tela do usuário final. Eles são bem-sucedidos onde está disponível uma largura suficiente de banda de rede dedicada e onde o servidor de vídeo e as estações receptoras são dedicados. Eles também fazem uso considerável de *buffers* no destino.

Os aplicativos altamente interativos apresentam problemas muito maiores. Muitos aplicativos multimídia são cooperativos (envolvendo vários usuários) e síncronos (exigindo que as atividades dos usuários sejam fortemente coordenadas). Eles abrangem um amplo espectro de contextos e cenários de aplicação. Por exemplo:

- Telefonia na Internet. Veja o quadro a seguir.
- Uma videoconferência simples, envolvendo dois ou mais usuários, cada um usando uma estação de trabalho equipada com uma câmera de vídeo digital, microfone, saída de som e capacidade de exibição de vídeo. Software aplicativo para suportar teleconferência simples surgiu há mais de uma década (*CU-SeeMe* [Dorcey 1995]) e agora é amplamente distribuído (por exemplo, *Skype*, *NetMeeting* [www.microsoft.com III], *iChat AV* [www.apple.com II]).
- Um recurso de ensaio e execução musical que permita músicos, em diferentes locais, tocar em conjunto [Konstantas *et al.* 1997]. Essa é uma aplicação multimídia particularmente exigente, porque as restrições de sincronização são muito severas.



Aplicações como essas exigem:

Comunicação com baixa latência: atrasos de ida e volta (RTT) de 100–300 ms, para que a interação entre os usuários pareça ser síncrona.

Estado distribuído síncrono: se um usuário interromper uma exibição de vídeo em determinado quadro, os outros usuários devem vê-la parada no mesmo quadro.

Sincronismo de mídia: todos os participantes de uma execução musical devem ouvi-la aproximadamente ao mesmo tempo (Konstantas *et al.* [1997] identificaram um requisito de sincronização dentro de 50 ms). Fluxos separados de trilha sonora e de vídeo devem manter “sincronização labial”; por exemplo, para o comentário de um usuário ao vivo em uma reprodução de vídeo ou para uma sessão de karaokê distribuída.

Sincronização externa: em conferências e em outras aplicações cooperativas podem existir dados ativos em outros formatos, como animações geradas por computador, dados de CAD, quadros-negros eletrônicos e documentos compartilhados. Suas atualizações devem ser distribuídas e agir de maneira que pareçam pelo menos aproximadamente sincronizadas com os fluxos multimídia baseados no tempo.

Na Seção 20.2, examinaremos as características dos dados multimídia. A Seção 20.3 descreverá estratégias para a alocação de recursos escassos para obter qualidade de serviço e a Seção 20.4 discutirá os métodos para seu escalonamento. A Seção 20.5 discutirá mé-

Telefonia na Internet – VoIP

A Internet não foi projetada para aplicações interativas em tempo real, como a telefonia, mas tornou-se possível usá-la para isso, como resultado do aumento na capacidade e no desempenho dos seus componentes básicos – seu *backbone* de enlaces de rede funciona a 10–40 Gbps e os roteadores que os interligam têm desempenho comparável. Normalmente, esses componentes funcionam com baixos fatores de carga (< 10% de utilização de largura de banda) e, portanto, o tráfego de IP raramente é atrasado ou eliminado como resultado da disputa por recursos.

Isso resultou na possibilidade de construção de aplicações de telefonia na Internet pública, por meio da transmissão de fluxos de amostras de voz digitalizadas da origem para o destino, através de datagramas UDP, sem nenhum preparativo especial para obter qualidade do serviço. As aplicações de VoIP (*Voice-over-IP*), como Skype e Vonage, contam com essa técnica, assim como os recursos de voz de aplicações de troca de mensagens instantânea, como AOL Instant Messaging, Apple iChat AV e Microsoft NetMeeting.

É claro que se trata de aplicações interativas de tempo real e permanece o problema da latência. Conforme discutido no Capítulo 3, o roteamento de pacotes IP acarreta um atraso inevitável em cada roteador pelos quais eles passam. Para rotas longas, esses atrasos podem ultrapassar facilmente 150 ms e os usuários observarão isso na forma de atrasos na interação das conversas. Por isso, as ligações telefônicas interurbanas (especialmente intercontinentais) na Internet sofrem muito mais com os atrasos do que as que usam a rede telefônica convencional.

Contudo, grande parte do tráfego de voz é transportada na Internet e a integração com a rede telefônica convencional está em andamento. O SIP (Session Initiation Protocol, definido no RFC 2543 [Handley *et al.* 1999]) é um protocolo em nível de aplicação para o estabelecimento de ligações de voz (assim como de outros serviços, como a troca de mensagens instantânea) pela Internet. Existem *gateways* para a rede telefônica convencional em muitos locais pelo mundo, permitindo que as ligações sejam iniciadas a partir de dispositivos conectados na Internet e terminem nos telefones convencionais ou em computadores pessoais.

	Taxa de dados (aproximada)	Amostra ou quadro tamanho	frequência
Conversação telefônica	64 kbps	8 bits	8.000/s
Som com qualidade de CD	1,4 Mbps	16 bits	44.000/s
Vídeo de TV padrão (não compactado)	120 Mbps	até 640 × 480 pixels × 16 bits	24/s
Vídeo de TV padrão (MPEG-1 compactado)	1,5 Mbps	variável	24/s
Vídeo HDTV (não compactado)	1.000-3.000 Mbps	até 1920 × 1080 pixels × 24 bits	24-60/s
Vídeo HDTV (MPEG-2/MPEG-4 compactado)	6-20 Mbps	variável	24-60/s

Figura 20.2 Características dos fluxos multimídia típicos.

todos para otimizar o fluxo de dados em sistemas multimídia. A Seção 20.6 descreverá três estudos de caso de sistemas multimídia: o servidor de arquivos de vídeo Tiger, um sistema de baixo custo, escalável, para a distribuição de fluxos de vídeo armazenados, concorrentemente, para grandes números de clientes; o BitTorrent, como um exemplo de aplicação de compartilhamento de arquivos *peer-to-peer* que suporta o *download* de arquivos multimídia grandes; e o End System *Multicast*, da CMU, como um exemplo de sistema que suporta transmissão de conteúdo de vídeo pela Internet.

20.2 Características dos dados multimídia

Referimo-nos aos dados de vídeo e áudio como contínuos e baseados no tempo. Como podemos definir essas características mais precisamente? O termo “contínuo” se refere à visão que o usuário tem dos dados. Internamente, a mídia contínua é representada como sequências de valores discretos que substituem uns aos outros com o passar do tempo. Por exemplo, uma imagem é amostrada 25 vezes por segundo para dar a impressão de uma cena se movendo com a qualidade de uma TV; um sinal sonoro é amostrado 8.000 vezes por segundo para transmitir fala com a qualidade de um telefone.

Diz-se que os fluxos multimídia são denominados *baseados no tempo* (ou *isocrônicos*), pois elementos de dados temporais nos fluxos de áudio e vídeo definem a semântica ou o “conteúdo” do fluxo. Os tempos nos quais os valores são reproduzidos, ou gravados, afetam a validade dos dados. Portanto, os sistemas que suportam aplicações multimídia precisam preservar a temporização ao manipular dados contínuos.

Frequentemente, os fluxos multimídia são volumosos. Assim, os sistemas que suportam aplicações multimídia precisam mover dados com maior desempenho de saída do que os sistemas convencionais. A Figura 20.2 mostra algumas taxas de dados e frequências de quadro/amostragem típicas. Observamos que os requisitos de largura de banda de recurso para algumas delas são muito grandes. Isso é especialmente verdade no caso de vídeo de qualidade razoável. Por exemplo, um fluxo de vídeo de TV

padrão não compactado exige mais de 120 Mbps, o que ultrapassa a capacidade de uma rede Ethernet de 100 Mbps. Um programa que copia ou que aplica uma transformação simples nos dados em cada quadro de um fluxo de vídeo de TV padrão exige menos de 10% da capacidade da CPU de um PC. Os valores para fluxos de televisão de alta definição são ainda mais altos e devemos notar que, em muitas aplicações, como as videoconferências, há necessidade de manipular vários fluxos de vídeo e áudio concorrentemente. Portanto, o uso de representações compactadas para superar esses problemas é fundamental, embora transformações, como a mistura e edição de vídeo, sejam difíceis de realizar com fluxos compactados.

A compactação pode reduzir os requisitos de largura de banda por fatores entre 10 e 100, mas os requisitos de temporização de dados contínuos não são afetados. Há bastante pesquisa e atividades de padronização sendo realizadas com o objetivo de produzir representações eficientes de propósito geral e métodos de compactação para fluxos de dados multimídia. Esse trabalho tem resultado em vários formatos de dados compactados, como GIF, TIFF e JPEG para imagens estáticas, e MPEG-1, MPEG-2 e MPEG-4 para sequências de vídeo.

Embora o uso de dados de vídeo e áudio compactados reduza os requisitos de largura de banda em redes de comunicação, ele impõe cargas adicionais substanciais sobre os recursos de processamento na origem e no destino. Frequentemente, isso tem sido fornecido usando-se *hardware* de propósito especial para processar e distribuir informação de vídeo e áudio – os codificadores/decodificadores (*codecs*) de vídeo e áudio encontrados nas placas de vídeo fabricadas para computadores pessoais. No entanto, o poder cada vez maior dos computadores pessoais e das arquiteturas de multiprocessadores agora permite que eles realizem grande parte desse trabalho em *software* usando filtros de codificação e decodificação de *software*. Essa estratégia oferece maior flexibilidade, com melhor suporte para formatos de dados específicos da aplicação, lógica de aplicação de propósito especial e o tratamento simultâneo de vários fluxos de mídia.

O método de compactação usado para os formatos de vídeo MPEG é assimétrico, com um algoritmo de compactação complexo e descompactação mais simples. Isso tende a ajudar no seu uso em videoconferências em *desktop*, no qual a compactação é frequentemente realizada por um *codec* de *hardware*, mas a descompactação dos vários fluxos que chegam ao computador de cada usuário é realizada por *software*, permitindo que o número de participantes na videoconferência varie sem considerar o número de *codecs* no computador de cada usuário.

20.3 Gerenciamento de qualidade de serviço

Quando as aplicações multimídia são executadas em redes de computadores pessoais, elas competem por recursos nas estações de trabalho que executam outras aplicações (ciclos de processador, ciclos de barramento, capacidade de *buffer*) e nas redes (enlaces físicos de transmissão, comutadores, *gateways*). Talvez as estações de trabalho e as redes tenham de suportar várias aplicações multimídia e convencionais. Há competição entre as aplicações multimídia e convencionais, entre diferentes aplicações multimídia e até entre os fluxos de mídia dentro de aplicações individuais.

O uso concorrente de recursos físicos para uma variedade de tarefas tem sido possível há muito tempo em sistemas operacionais multitarefa e em redes compartilhadas. Nos sistemas operacionais multitarefa, o processador central é alocado para tarefas (ou processos) individuais no esquema de rodízio, ou em outro esquema de escalonamento

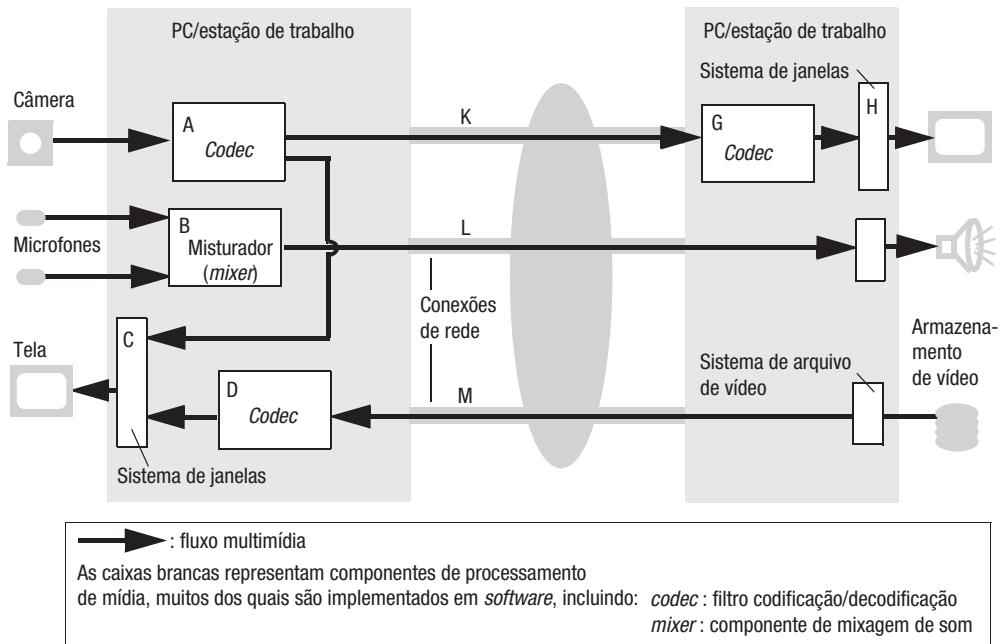


Figura 20.3 Componentes típicos de infraestrutura para aplicações multimídia.

que compartilhe os recursos de processamento na base do *melhor esforço*, entre todas as tarefas que estão competindo pelo processador central.

As redes são projetadas de forma a permitir que mensagens de diferentes origens sejam intercaladas, possibilitando a existência de muitos canais de comunicação virtuais nos mesmos canais físicos. A tecnologia de rede local predominante, a Ethernet, gerencia um meio de transmissão compartilhado na base do *melhor esforço*. Qualquer nó pode usar o meio quando ele estiver desocupado, mas podem ocorrer colisões no envio de quadros e, quando elas ocorrem, os nós de envio esperam por períodos aleatórios para evitar colisões repetidas. Provavelmente, as colisões ocorrem quando a rede está muito carregada e esse esquema não pode fornecer quaisquer garantias a respeito da largura de banda ou da latência em tais situações.

A principal característica desses esquemas de alocação de recursos é que eles tratam dos aumentos na demanda distribuindo os recursos disponíveis entre as tarefas concorrentes. O rodízio (*round robin*) e outros métodos de melhor esforço para compartilhamento de ciclos de processador e largura de banda de rede não podem satisfazer as necessidades das aplicações multimídia. Conforme vimos, o processamento e a transmissão de fluxos multimídia em momentos oportunos são fundamentais para eles. Uma distribuição atrasada não tem valor. Para conseguir a distribuição com restrições temporais, as aplicações precisam garantir que os recursos necessários sejam alocados e escalonados nos momentos exigidos.

O gerenciamento e a alocação de recursos para fornecer tais garantias são referidos como *gerenciamento de qualidade de serviço*. A Figura 20.3 mostra os componentes de infraestrutura para uma aplicação simples de conferência multimídia sendo executada em dois computadores pessoais, usando compactação de dados de *software* e conversão de

<i>Componente</i>		<i>Largura de banda</i>	<i>Latência</i>	<i>Taxa de perda</i>	<i>Recursos exigidos</i>
Câmera	Saída:	10 quadros/s, vídeo bruto 640x480x16 bits	—	Zero	—
A Codec	Entrada:	10 quadros/s, vídeo bruto	Interativa	Baixa	10 ms CPU a cada 100 ms; 10 Mbytes RAM
	Saída:	fluxo MPEG-1			
B Misturador	Entrada:	2 × 44 kbps áudio	Interativa	Muito baixa	1 ms CPU a cada 100 ms; 1 Mbytes RAM
	Saída:	1 × 44 kbps áudio			
H Sistema de janelas	Entrada:	Várias	Interativa	Baixa	5 ms CPU a cada 100 ms; 5 Mbytes RAM
	Saída:	Framebuffer de 50 quadros/s			
K Conexão de rede	Entrada/ Saída:	Fluxo MPEG-1, aprox. 1,5 Mbps	Interativa	Baixa	1,5 Mbps, protocolo de fluxo de baixa perda
L Conexão de rede	Entrada/ Saída:	Áudio 44 kbps	Interativa	Muito baixa	44 kbps, protocolo de fluxo de perda muito baixa

Figura 20.4 Especificações da qualidade de serviço para componentes de aplicações multimídia mostrada na Figura 20.3.

formato. As caixas brancas representam os componentes de *software* cujos requisitos de recursos podem afetar a qualidade de serviço da aplicação.

A figura mostra a arquitetura abstrata mais comumente usada para *software* multimídia, na qual *fluxos* de mídia de elementos de dados gerados continuamente (quadros de vídeo, amostras de áudio) são processados por um conjunto de processos e transferidos por meio de conexões entre eles. Os processos produzem, transformam e consomem fluxos contínuos de dados multimídia. As conexões ligam os processos em uma sequência de uma *origem* para um *destino*, no qual os elementos de mídia são representados ou consumidos. As conexões entre os processos podem ser implementadas por conexões de rede, ou por meio de transferências na memória quando os processos residem na mesma máquina. Para os elementos de dados multimídia chegarem a seu destino a tempo, cada processo deve receber tempo de CPU, capacidade de memória e largura de banda de rede adequadas para a execução da tarefa designada, e devem ser programados para usar os recursos de forma suficientemente frequente para enviar a tempo os elementos de dados de seu fluxo para o próximo processo.

Na Figura 20.4, especificamos os requisitos de recurso dos principais componentes de *software* e conexões de rede da Figura 20.3 (observe as letras correspondentes aos componentes nessas duas figuras). Claramente, os recursos exigidos só poderão ser garantidos se houver um componente de sistema responsável pela alocação e pelo escalonamento desses recursos. Vamos nos referir a esse componente como *gerenciador de qualidade de serviço*.

A Figura 20.5 mostra as responsabilidades do gerenciador de qualidade de serviço em forma de fluxograma. Nas duas próximas subseções, descreveremos as duas principais subtarefas do gerenciador de qualidade de serviço:

Negociação de qualidade de serviço: a aplicação indica seus requisitos de recurso para o gerenciador de qualidade de serviço. O gerenciador de qualidade de serviço avalia a possibilidade de atender aos requisitos com um banco de dados dos recur-

sos disponíveis e em relação aos comprometimentos de recurso correntes, e dá uma resposta positiva ou negativa. Se a resposta for negativa, a aplicação poderá ser reconfigurada para usar recursos reduzidos e o procedimento é repetido.

Controle de admissão: se o resultado da avaliação de recurso for positivo, os recursos solicitados serão reservados e a aplicação receberá um *contrato de recurso* indicando os recursos que foram reservados. O contrato inclui um limite de tempo. Então, a aplicação fica livre para ser executada. Se ela mudar seus requisitos de recurso, deverá notificar o gerenciador de qualidade de serviço. Se os requisitos diminuírem, os recursos liberados serão retornados ao banco de dados como recursos disponíveis. Se eles aumentarem, uma nova rodada de negociação e controle de admissão será iniciada.

No restante desta seção, descreveremos com mais detalhes as técnicas para executar essas subtarefas. É claro que, enquanto uma aplicação está em execução, há necessidade de escalonamento refinado de recursos como tempo de processador e largura de banda de rede para garantir que os processos em tempo real recebam a tempo seus recursos alocados. As técnicas para isso serão discutidas na Seção 20.4.

20.3.1 Negociação de qualidade de serviço

Para negociar a qualidade de serviço entre uma aplicação e seu sistema subjacente, a aplicação deve especificar seus requisitos de qualidade de serviço para o gerenciador de qualidade de serviço. Isso é feito por meio da transmissão de um conjunto de parâmetros. Três parâmetros têm maior interesse quando se trata do processamento e transporte de fluxos multimídia: *largura de banda*, *latência* e *taxa de perda*.

Largura de banda: a largura de banda de um fluxo, ou componente multimídia, é a taxa na qual os dados fluem por ela.

Latência: latência é o tempo exigido para um elemento individual de dados se mover em um fluxo, da origem até o destino. Isso pode variar, dependendo do volume de outros dados presentes no sistema e de outras características da carga do sistema. Essa variação é chamada *jitter* – formalmente, *jitter* é a primeira derivada da latência.

Taxa de perda: como a distribuição atrasada de dados multimídia não tem valor, os elementos de dados serão eliminados quando for impossível entregá-los antes de seu tempo de distribuição programado. Em um ambiente de qualidade de serviço perfeitamente gerenciado isso nunca deve acontecer, mas até agora existem poucos ambientes assim, pelos motivos mencionados anteriormente. Além disso, o custo em recursos para garantir a distribuição temporal de cada elemento de mídia é frequentemente inaceitável – para tratar com picos ocasionais, provavelmente haverá o envolvimento de muito mais do que o requisito médio de reserva de recursos. A alternativa adotada é aceitar certa taxa de perda de dados, isto é, quadros de vídeo ou amostras de áudio eliminados. Normalmente, as relações aceitáveis são mantidas baixas – raramente mais do que 1% e muito menos para aplicações cuja qualidade é crítica.

Os três parâmetros podem ser usados:

1. Para descrever as características de um fluxo multimídia em um ambiente específico. Por exemplo, um fluxo de vídeo pode exigir uma largura de banda média de 1,5 Mbps e, como ele é usado em uma aplicação de conferência, precisa ser transferido com um atraso máximo de 150 ms para evitar hiatos na palestra. O algoritmo de

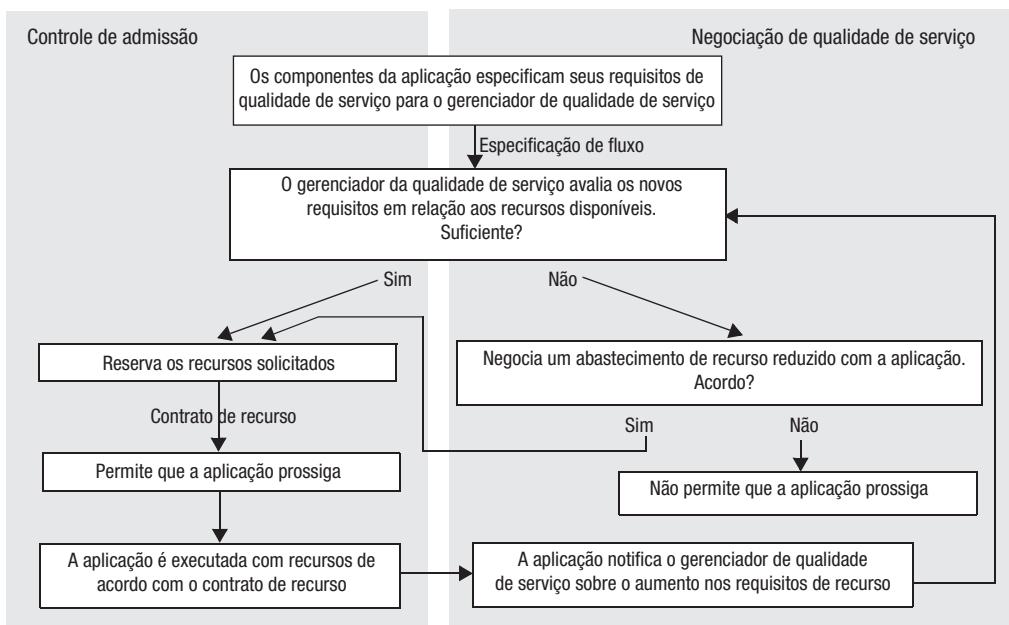


Figura 20.5 A tarefa do gerenciador de qualidade de serviço.

descompactação usado no destino pode produzir aceitavelmente imagens com uma taxa de perda de um quadro em 100.

2. Para descrever a capacidade dos recursos de transportar um fluxo. Por exemplo, uma rede pode fornecer conexões de largura de banda de 64 kbps, seus algoritmos de enfileiramento garantir atrasos de menos de 10 ms e o sistema de transmissão garantir uma taxa de perda menor do que 1 em 10^6 .

Os parâmetros são interdependentes. Por exemplo:

- A taxa de perda nos sistemas modernos raramente depende de erros em bits devido a ruído ou defeitos; ela resulta no estouro do *buffer* e de dados dependentes do tempo chegando atrasados. Portanto, quanto maior puderem ser a largura de banda e o atraso, mais provável é que a taxa de perda seja baixa.
- Quanto menor a largura de banda global de um recurso, comparada com sua carga, mais mensagens serão armazenadas e maiores precisam ser os *buffers* para esse acúmulo, para evitar perda. Quanto maiores são os *buffers*, torna-se mais provável que as mensagens esperem que outras mensagens que estão na sua frente sejam atendidas – isto é, maior se tornará o atraso.

Especificando os parâmetros de qualidade de serviço para fluxos • Os valores dos parâmetros de qualidade de serviço podem ser declarados explicitamente (por exemplo, para o fluxo de saída da câmera da Figura 20.3, poderíamos exigir *largura de banda*: 50 Mbps, *atraso*: 150 ms, *perda*: < 1 quadro em 10^3) ou implicitamente (por exemplo, a largura de banda do fluxo de entrada para a conexão de rede *K* é o resultado da aplicação de compactação MPEG-1 na saída da câmera).

Contudo, o caso mais comum é precisarmos especificar um valor e um intervalo de variação permissível. Aqui, consideramos esse requisito para cada um dos parâmetros:

Largura de banda: a maior parte das técnicas de compactação de vídeo produz um fluxo de quadros de diferentes tamanhos, dependendo do conteúdo original do vídeo bruto. Para MPEG, a razão de compactação média está situada entre 1:50 e 1:100, mas isso variará dinamicamente, dependendo do conteúdo; por exemplo, a largura de banda exigida será maior quando o conteúdo estiver mudando mais rapidamente. Por isso, frequentemente é útil citar os parâmetros da qualidade do serviço como valores máximo, mínimo ou médio, dependendo do tipo de regime de gerenciamento de qualidade de serviço que será usado.

Outro problema que surge na especificação da largura de banda é a caracterização da *tакса de rajada (burst)*. Considere três fluxos de 1 Mbps. Um fluxo transfere um único quadro de 1 Mbit a cada segundo, o segundo é um fluxo assíncrono de elementos de animação gerados por computador, com uma largura de banda média de 1 Mbps, e o terceiro envia uma amostra de som de 100 bits a cada microsegundo. Embora todos os três fluxos exijam a mesma largura de banda, seus padrões de tráfego são muito diferentes.

Uma maneira de lidar com as irregularidades é definir um parâmetro de rajada, além da taxa de transmissão e do tamanho do quadro. O parâmetro de rajada especifica o número máximo de elementos de mídia que podem chegar cedo – isto é, antes do que devem chegar, de acordo com a taxa de chegada normal. O modelo LBAP (linear-bounded arrival processes) usado em Anderson [1993] define o número máximo de mensagens em um fluxo durante qualquer intervalo de tempo t como $Rt + B$, onde R é a taxa de transmissão e B é o tamanho máximo de rajada. A vantagem de usar esse modelo é que ele reflete bem as características das origens multimídia: os dados multimídia lidos de discos normalmente são distribuídos em blocos grandes, e os dados recebidos das redes frequentemente chegam na forma de sequências de pacotes menores. Neste caso, o parâmetro de rajada define a quantidade de espaço no *buffer* exigida para evitar perda.

Latência: alguns requisitos de temporização em aplicações multimídia resultam do próprio fluxo: se um quadro de um fluxo não for processado com a mesma taxa com que os quadros chegam, haverá acúmulo de trabalho e a capacidade do *buffer* poderá ser excedida. Se isso precisa ser evitado, um quadro não deve permanecer em um *buffer*, em média, por mais do que $1/R$, onde R é a taxa de quadros de um fluxo, senão ocorrerá um acúmulo de trabalho. Caso ocorram acúmulos de trabalho, o número e o tamanho dos trabalhos acumulados afetarão o atraso de ponta a ponta máximo de um fluxo, além dos tempos de processamento e propagação.

Outros requisitos de latência surgem do ambiente da própria aplicação. Em aplicações de conferência, a necessidade de interação aparentemente instantânea entre os participantes torna necessário obter atrasos de ponta a ponta absolutos de não mais do que 150 ms para evitar problemas na percepção humana, enquanto que na reprodução de vídeo, para garantir uma resposta apropriada do sistema aos comandos, como *Pausa* e *Parar*, a latência máxima deve ser da ordem de 500 ms.

Uma terceira consideração para o tempo de distribuição de mensagens multimídia é o *jitter* – a variação no período entre a distribuição de dois quadros adjacentes. Embora a maioria dos dispositivos multimídia certifique-se de apresentar dados em sua velocidade normal, sem variação, as apresentações de *software* (por exemplo, em um decodificador de *software* para quadros de vídeo) precisam tomar um cuidado extra para evitar o *jitter*. O *jitter* é resolvido basicamente com o uso de *buffers*, mas sua capacidade de remoção é limitada, pois o atraso de ponta a ponta total é restrito pela consideração mencionada

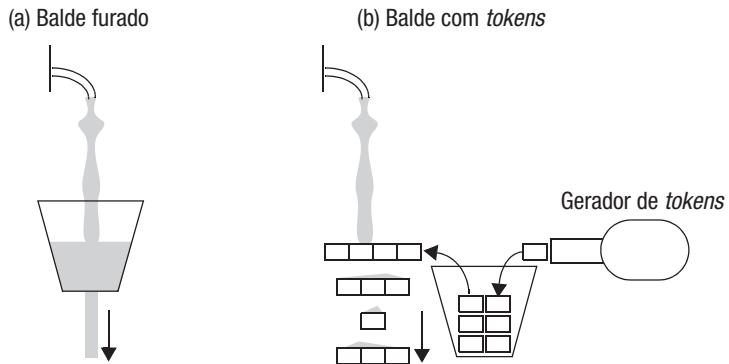


Figura 20.6 Algoritmos de conformação de tráfego.

anteriormente; portanto, a reprodução de sequências de mídia também exige que os elementos da mídia cheguem antes de prazos finais fixos.

Taxa de perda: a taxa de perda é o parâmetro de qualidade de serviço mais difícil de especificar. Os valores de taxa de perda normais resultam de cálculos de probabilidade sobre o estouro de *buffers* e mensagens atrasadas. Esses cálculos são baseados em suposições de pior caso ou em distribuições padrão. Nenhum deles é necessariamente uma boa solução para situações práticas. Entretanto, as especificações de taxa de perda são necessárias para quantificar os parâmetros de largura de banda e latência: duas aplicações podem ter as mesmas características de largura de banda e latência; elas parecerão substancialmente diferentes quando uma aplicação perder cada quinto elemento da mídia e a outra perder apenas um em um milhão.

Assim como nas especificações de largura de banda, em que é importante não apenas o volume dos dados enviados em um intervalo de tempo, mas também sua distribuição nesse intervalo de tempo, uma especificação de taxa de perda precisa determinar o intervalo de tempo durante o qual deve esperar certa perda. Em particular, taxas de perda dadas para períodos de tempo infinitos não são úteis, pois qualquer perda em um tempo curto pode ultrapassar a taxa de longo prazo significativamente.

Conformação de tráfego • Conformação de tráfego é o termo usado para descrever o uso de *buffers* de saída para suavizar o fluxo de elementos de dados. O parâmetro da largura de banda de um fluxo multimídia normalmente fornece uma aproximação ideal do padrão de tráfego real que ocorrerá quando o fluxo for transmitido. Quanto mais próximo for o padrão de tráfego real da sua descrição, melhor o sistema poderá manipular o tráfego; em particular, quando ele utilizar métodos de escalonamento projetados para requisições periódicas.

O modelo LBAP de variações de largura de banda exige regular a taxa de rajada dos fluxos multimídia. Todo fluxo pode ser regulado pela inserção de um *buffer* na origem e pela definição de um método com que os elementos de dados sejam consumidos do *buffer*. Uma boa ilustração desse método é a imagem de um balde furado (Figura 20.6): o balde pode ser preenchido arbitrariamente com água, até ficar cheio e, por meio de um furo embaixo do balde, a água fluirá continuamente. O algoritmo do balde furado garante que um fluxo nunca será maior do que uma taxa R . O tamanho do *buffer* B define a rajada máxima que um fluxo pode estar sujeito sem perder elementos. B também limita o tempo durante o qual um elemento permanecerá no balde.

	Versão de protocolo
Largura de banda:	Unidade de transmissão máxima
	Taxa do balde com <i>tokens</i>
	Tamanho do balde com <i>tokens</i>
Atraso:	Taxa de transmissão máxima
	Atraso mínimo notado
Perda:	Variação máxima do atraso
	Sensibilidade de perda
	Sensibilidade de perda de rajada
	Intervalo de perda
	Garantia da qualidade do serviço

Figura 20.7 A especificação de fluxo RFC 1363.

O algoritmo do balde furado elimina completamente as rajadas. Tal eliminação nem sempre é necessária, já que a largura de banda é limitada em qualquer intervalo de tempo. O algoritmo do balde com fichas (*tokens*) consegue isso, permitindo que rajadas maiores ocorram, quando um fluxo estiver ocioso por algum tempo (Figura 20.6b). Trata-se de uma variação do algoritmo do balde furado, na qual são gerados *tokens* para enviar dados a uma taxa fixa R . Eles são coletados em um balde de tamanho B . Dados de tamanho S só podem ser enviados se pelo menos S *tokens* estiverem no balde. O processo de envio remove, então, esses S *tokens*. O algoritmo do balde com *tokens* garante que, em qualquer intervalo t , o volume de dados enviados não seja maior do que $Rt + B$. Trata-se, portanto, de uma implementação do modelo LBAP.

Picos altos de tamanho B só podem ocorrer em um sistema de balde com *tokens* quando um fluxo estiver ocioso por algum tempo. Para evitar essas rajadas, um balde furado simples pode ser combinado com um balde com *tokens*. A taxa de fluxo F desse balde precisa ser significativamente maior do que R para que esse esquema faça sentido. Seu único objetivo é fragmentar rajadas realmente grandes.

Especificações de fluxo • Um conjunto de parâmetros de qualidade de serviço é normalmente conhecido como *especificação de fluxo*. Existem vários exemplos de especificações de fluxo e todos são semelhantes. Na RFC 1363 [Partridge 1992], uma especificação de fluxo é definida como 11 valores numéricos de 16 bits (Figura 20.7), os quais refletem os parâmetros de qualidade de serviço discutidos anteriormente, da seguinte maneira:

- A unidade de transmissão máxima e a taxa de transmissão máxima determinam a largura de banda máxima exigida pelo fluxo.
- O tamanho do balde com *tokens* e a taxa determinam a taxa de rajada do fluxo.
- As características de atraso são especificadas pelo atraso mínimo que uma aplicação pode observar (pois queremos evitar a super-otimização para atrasos curtos) e o *jitter* máximo que ela pode aceitar.
- As características de perda são definidas pelo número total aceitável de perdas em certo intervalo e o número máximo de perdas consecutivas.

Existem muitas alternativas para expressar cada grupo de parâmetros. Em SRP [Anderson *et al.* 1990], a taxa de rajada de um fluxo é dada por um parâmetro de trabalho antecipado máximo, que define o número de mensagens em que um fluxo pode estar à frente de sua taxa de chegada normal, em qualquer ponto no tempo. Em Ferrari e Verma [1990], é dado um limite de atraso para pior caso: se o sistema não puder garantir o transporte dos dados dentro desse período de tempo, seu transporte será inútil para a aplicação. Na RFC 1190, a especificação do protocolo ST-II [Topolcic 1990], a perda é representada como a probabilidade de cada pacote ser eliminado.

Todos os exemplos anteriores fornecem um espectro contínuo de valores de qualidade de serviço. Se o conjunto de aplicações e fluxos a serem suportados for limitado, poderá ser suficiente definir um conjunto distinto de classes de qualidade de serviço: por exemplo, áudio com qualidade de telefone e de alta fidelidade, vídeo ao vivo e reprodução, etc. Os requisitos de todas as classes devem ser conhecidos implicitamente por todos os componentes do sistema; o sistema pode até ser configurado para certa mistura de tráfego.

Procedimentos de negociação • Para aplicações multimídia distribuídas, os componentes de um fluxo provavelmente estarão localizados em vários nós. Haverá um gerenciador de qualidade de serviço em cada nó. Uma estratégia simples de negociação de qualidade de serviço é seguir o fluxo de dados ao longo de cada fluxo, da origem até o destino. Um componente de origem inicia a negociação enviando uma especificação de fluxo para seu gerenciador de qualidade de serviço local. O gerenciador pode verificar, em seu banco de dados de recursos disponíveis, se a qualidade de serviço solicitada pode ser fornecida. Se outros sistemas estiverem envolvidos na aplicação, a especificação de fluxo será encaminhada para o próximo nó onde os recursos são exigidos. A especificação de fluxo percorre todos os nós, até que o destino final seja encontrado. Então, a informação sobre se a qualidade de serviço desejada pode ser fornecida pelo sistema é passada de volta para a origem. Essa estratégia simples de negociação é satisfatória para muitos propósitos, mas ela não considera as possibilidades de conflito entre negociações de qualidade de serviço concorrentes partindo de nós diferentes. Um procedimento de negociação de qualidade de serviço transacional distribuído exigiria uma solução completa para esse problema.

Raramente as aplicações têm requisitos de qualidade do serviço fixos. Em vez de retornar um valor booleano sobre se certa qualidade de serviço pode ser fornecida ou não, é mais apropriado o sistema determinar qual tipo de qualidade de serviço pode fornecer e deixar a aplicação decidir se ela é aceitável. Para evitar qualidade de serviço superotimizada, ou para cancelar a negociação quando se tornar claro que a qualidade desejada não pode ser obtida, é comum especificar um valor desejado e o pior valor para cada parâmetro de qualidade de serviço. Uma aplicação pode especificar que deseja uma largura de banda de 1,5 Mbps, mas que também poderia manipular 1 Mbps, ou que o atraso deve ser de 200 ms, mas que 300 ms seria o pior caso ainda aceitável. Como apenas um parâmetro pode ser otimizado por vez, sistemas como o HeiRAT [Vogt *et al.* 1993] esperam que o usuário defina valores de apenas dois parâmetros e deixam que o sistema otimize o terceiro.

Se um fluxo tem vários destinos, o caminho de negociação bifurca de acordo com o fluxo de dados. Como uma ampliação simples do esquema anterior, nós intermediários podem agregar mensagens de retorno de qualidade de serviço dos destinos, para produzir valores de pior caso para os parâmetros de qualidade de serviço. Então, a largura de banda disponível se torna a menor largura de banda disponível de todos os destinos, o atraso se torna o mais longo de todos os destinos e a taxa de perda se torna a maior de todos os destinos. Esse é o procedimento praticado pelos protocolos de negociação iniciada pelo remetente, como SRP, ST-II e RCAP [Banerjea e Mah 1991].

Nas situações com destinos heterogêneos, normalmente é inadequado atribuir uma qualidade de serviço de pior caso comum para todos os destinos. Em vez disso, cada destino deve receber a melhor qualidade de serviço possível. Isso exige um processo de negociação iniciada pelo receptor, em vez de orientada pelo remetente. O RSVP [Zhang *et al.* 1993] é um protocolo de negociação de qualidade de serviço alternativo no qual os destinos se conectam a fluxos. As origens comunicam a existência de fluxos e suas características inerentes para todos os destinos. Então, os destinos podem se conectar no nó mais próximo através do qual o fluxo passa e extraírem dados de lá. Para que obtenham dados com a qualidade de serviço apropriada, talvez eles precisem usar técnicas como a filtragem (discutida na Seção 20.5).

20.3.2 Controle de admissão

O controle de admissão regula o acesso aos recursos para evitar sobrecarga de recurso e para proteger os recursos de requisições que eles não possam atender. Isso envolve rejeitar pedidos de serviço, caso os requisitos de recurso de um novo fluxo multimídia violem as garantias de qualidade de serviço existentes.

Um esquema de controle de admissão é baseado em algum conhecimento da capacidade global do sistema e da carga gerada por aplicação. A especificação do requisito de largura de banda de uma aplicação pode refletir a quantidade máxima de largura de banda que a aplicação exigirá, a largura de banda mínima que precisará para funcionar ou algum valor médio. Correspondentemente, um esquema de controle de admissão pode basear sua alocação de recursos em qualquer um desses valores.

Para recursos que possuem um único alocador de recursos, o controle de admissão é simples. Os recursos que possuem pontos de acesso distribuídos, como muitas redes locais, exigem uma entidade de controle de admissão centralizada, ou algum algoritmo de controle de admissão distribuído, que evite admissões concorrentes e conflitantes. O controle de acesso em uma rede de barramento, feito pelas estações de trabalho, pertence a essa categoria; entretanto, mesmo os sistemas multimídia que realizam alocação de largura de banda extensivamente não controlam o acesso ao meio (barramento), pois não se considera que a largura de banda do barramento esteja na janela de escassez.

Reserva de largura de banda • Uma maneira comum de garantir determinado nível de qualidade de serviço para um fluxo multimídia é reservar parte da largura de banda de recurso para seu uso exclusivo. Para atender durante todo tempo aos requisitos de um fluxo é necessário que seja feita uma reserva para sua largura de banda máxima. Essa é a única maneira possível de fornecer qualidade de serviço garantida para uma aplicação – pelo menos enquanto não ocorra nenhuma falha de sistema catastrófica. Ela é usada para aplicações que não conseguem se adaptar a diferentes níveis de qualidade de serviço ou que se tornam inúteis quando ocorrem quedas de qualidade. Exemplos incluem algumas aplicações na área médica (um sintoma pode aparecer em um vídeo de raio X exatamente no momento em que quadros de vídeo são eliminados) e o armazenamento de vídeo (em que quadros eliminados resultarão em uma falha visível sempre que o vídeo for reproduzido).

A reserva baseada nos requisitos máximos pode ser simples: ao se controlar o acesso a uma rede de determinada largura de banda B , podem ser admitidos s fluxos multimídia de largura de banda b_s desde que $\sum b_s \leq B$. Assim uma rede *token ring* de 16 Mb/s pode suportar até 10 fluxos de vídeo digital de 1,5 Mb/s cada.

Infelizmente, os cálculos de capacidade nem sempre são tão simples como no caso de uma rede *token ring*. Alocar largura de banda de CPU da mesma maneira exige o conhecimento do perfil de execução de cada aplicação. Entretanto, os tempos de execução

dependem do processador usado e frequentemente não podem ser determinados com precisão. Embora existam várias propostas para o cálculo automático do tempo de execução [Mok 1985, Kopetz *et al.* 1989], nenhuma delas tem sido amplamente usada. Os tempos de execução normalmente são determinados por meio de medidas que frequentemente têm margens de erros grandes e portabilidade limitada.

Para codificações de mídia normais, como MPEG, a largura de banda real consumida por uma aplicação pode ser significativamente menor do que sua largura de banda máxima. As reservas baseadas nos requisitos máximos podem levar, então, ao desperdício de largura de banda de recurso: as solicitações de novas admissões são rejeitadas, embora pudesse ser atendidas com a largura de banda reservada, mas não usada realmente pelas aplicações existentes.

Multiplexação estatística • Devido à potencial subutilização que pode ocorrer, é comum reservar recursos além dos disponíveis. As garantias resultantes, frequentemente chamadas de estatísticas ou condicionais, para distingui-las das garantias determinísticas ou incondicionais, apresentadas anteriormente, só são válidas com alguma probabilidade (normalmente muito alta). As garantias estatísticas tendem a fornecer melhor utilização de recurso quando não consideram o pior caso. Contudo, assim como quando a alocação de recurso é baseada nos requisitos mínimos, ou médios, cargas de pico simultâneas podem causar quedas na qualidade de serviço; as aplicações precisam ser capazes de tratar dessas quedas.

A multiplexação estatística é baseada na hipótese de que para um grande número de fluxos, a largura de banda agregada exigida permanece quase constante, independentemente da largura de banda dos fluxos individuais. Isso presume que, quando um fluxo enviar um grande volume de dados, haverá também outro fluxo que enviará um volume pequeno e, no total, os requisitos serão equilibrados. Entretanto, isso só acontece com fluxos não correlacionados.

Conforme mostram as experiências [Leland *et al.* 1993], o tráfego multimídia em ambientes típicos contradiz essa hipótese. Dado um número maior de fluxos de transferência simultânea, o tráfego agregado ainda permanecerá repleto de transferências simultâneas. O termo *autossemelhante* tem sido aplicado a esse fenômeno, significando que o tráfego agregado mostra semelhança com os fluxos individuais dos quais é composto.

20.4 Gerenciamento de recursos

Para fornecer certo nível de qualidade de serviço para uma aplicação, um sistema não apenas precisa ter recursos suficientes (desempenho), como também precisa tornar esses recursos disponíveis para a aplicação quando eles forem necessários (escalonamento).

20.4.1 Escalonamento de recursos

Os processos precisam ter recursos atribuídos de acordo com suas prioridades. Um escalonador de recursos determina a prioridade dos processos com base em certos critérios. Os escalonadores tradicionalmente encontrados em sistemas operacionais frequentemente baseiam suas decisões de alocar a CPU a processos em função do tempo de resposta e na imparcialidade: as tarefas com uso intenso de E/S recebem alta prioridade para garantir resposta rápida às requisições de usuário, as tarefas ligadas à CPU recebem prioridades mais baixas e, de modo geral, os processos na mesma classe são tratados igualmente.

Os dois critérios permanecem válidos para sistemas multimídia, mas a existência de prazos finais para a distribuição de elementos de dados multimídia individuais muda a natureza do problema do escalonamento. Algoritmos de escalonamento em tempo real

podem ser aplicados a esse problema, conforme discutido a seguir. Como os sistemas multimídia têm de manipular mídia discreta e contínua, torna-se um desafio fornecer um serviço suficientemente ágil para tratar fluxos dependentes do tempo, sem causar inanição de acesso à mídia discreta e de outras aplicações interativas.

Métodos de escalonamento precisam ser aplicados (e coordenados) em todos os recursos que afetam o desempenho de uma aplicação multimídia. Em um cenário típico, um fluxo multimídia seria recuperado do disco e depois enviado, por meio de uma rede, para uma estação de destino, onde seria sincronizado com um fluxo proveniente de outra origem e, finalmente, exibido. Os recursos exigidos nesse exemplo incluem o disco, a rede e as CPUs, assim como memória e largura de banda, em todos os sistemas envolvidos.

Escalonamento imparcial (fairness) • Se vários fluxos concorrem pelo mesmo recurso, torna-se necessário ser imparcial e impedir que fluxos mal comportados ocupem largura de banda em demasia. Uma estratégia simples para garantir a imparcialidade é aplicar escalonamento em rodízio (*round-robin*) em todos os fluxos da mesma classe. Embora em Nagle [1987] tal método tenha sido introduzido pacote por pacote, em Demers *et al.* [1989] ele é usado bit por bit, o que proporciona uma maior imparcialidade com relação aos tamanhos de pacote variados e aos tempos de chegada de pacote. Esses métodos são conhecidos como *enfileiramento imparcial (fair queuing)*.

Na prática, os pacotes não podem ser enviados bit a bit, mas dada uma taxa de quadros é possível calcular, para cada pacote, quando ele deve ser enviado completamente. Se as transmissões de pacote forem ordenadas com base nesse cálculo, se obterá quase o mesmo comportamento do rodízio de bit a bit real, exceto que, quando um pacote grande for enviado, ele poderá bloquear a transmissão de um pacote menor, o qual teria sido preferido no esquema bit a bit. Entretanto, nenhum pacote é atrasado por mais do que o tempo de transmissão de pacote máximo.

Todos os esquemas de rodízio básicos atribuem a mesma largura de banda para cada fluxo. Para levar em conta a largura de banda individual dos fluxos, o esquema bit a bit pode ser ampliado de modo que, para certos fluxos, um número maior de bits possa ser transmitido por ciclo. Esse método é chamado de *enfileiramento imparcial ponderado (weighted fair queuing)*.

Escalonamento em tempo real • Vários algoritmos de escalonamento em tempo real foram desenvolvidos para atender às necessidades de escalonamento da CPU de aplicações como o controle de processo industrial. Supondo que os recursos de CPU não tenham sido alocados em demasia (o que é a tarefa do gerenciador de qualidade de serviço), esses algoritmos atribuem repartições de tempo de CPU para um conjunto de processos de uma maneira que garanta a execução de suas tarefas a tempo.

Os métodos tradicionais de escalonamento em tempo real se adaptam muito bem ao modelo de fluxos multimídia contínuos regulares. O escalonamento EDF (*Earliest-Deadline-First*) quase tem se tornado um sinônimo para esses métodos. Um escalonador EDF utiliza um prazo final, que é associado a cada um de seus itens de trabalho, para determinar o próximo item a ser processado: o item com o prazo final mais adiantado é processado primeiro. Nas aplicações multimídia, identificamos cada elemento de mídia que chega a um processo como item de trabalho. O escalonamento EDF se mostrou excelente para a alocação de um único recurso, baseado em critérios de temporização: se houver um escalonamento que atenda a todos os requisitos de temporização, o escalonamento EDF o encontrará [Dertouzos 1974].

O escalonamento EDF exige uma decisão de escalonamento por mensagem (isto é, por elemento multimídia). Seria mais eficiente basear o escalonamento nos elementos que exis-

tem por um tempo maior. O escalonamento RM (*Rate-Monotonic*) é uma técnica proeminente para escalonamento em tempo real de processos periódicos, que obtêm exatamente isso. Os fluxos recebem prioridades de acordo com sua velocidade: quanto maior a velocidade dos itens de trabalho em um fluxo, maior é a prioridade de um fluxo. O escalonamento RM tem se mostrado excelente para situações que utilizam apenas determinada largura de banda por menos de 69% [Liu e Layland 1973]. Usando-se tal esquema de alocação, a largura de banda restante poderia ser dada a aplicações que não fossem em tempo real.

Para suportar tráfego em tempo real com transferência simultânea, os métodos de escalonamento em tempo real básicos devem ser ajustados para distinguirem entre itens de trabalho de mídia contínua críticos e não críticos quanto ao tempo. Em Govindan e Anderson [1991], é introduzido o escalonamento de prazo final/trabalho antecipado. Ele permite que mensagens em um fluxo contínuo cheguem antes em rajadas, mas só aplica escalonamento EDF em uma mensagem no momento de sua chegada normal.

20.5 Adaptação de fluxo

Quando certa qualidade de serviço não pode ser garantida, ou só pode ser garantida com certa probabilidade, uma aplicação precisa se adaptar a níveis de qualidade de serviço mutantes, ajustando seu desempenho de acordo. Para fluxos de mídia contínuos, o ajuste se traduz em diferentes níveis de qualidade de apresentação de mídia.

A forma mais simples de ajuste é eliminar informações. Isso é feito facilmente em fluxos de áudio, em que as amostras são independentes umas das outras, mas pode ser notado imediatamente pelo ouvinte. Desaparecimentos em um fluxo de vídeo codificado em Motion JPEG, em que cada quadro tem posição livre, são mais toleráveis. Os mecanismos de codificação, como o MPEG, em que a interpretação de um quadro depende dos valores de vários quadros adjacentes, são menos robustos em relação às omissões: eles demoram mais para se recuperar de erros e o mecanismo de codificação pode, na verdade, amplificar os erros.

Se houver largura de banda insuficiente, e os dados não forem eliminados, o atraso de um fluxo aumentará com o passar do tempo. Para aplicações não interativas isso pode ser aceitável, embora possa eventualmente levar a estouros de *buffer*, à medida que os dados são acumulados entre a fonte e os destinos. Para aplicações de videoconferência e outras aplicações interativas, atrasos cada vez maiores não são aceitáveis ou só devem existir por um curto período de tempo. Se um fluxo estiver atrasado em seu tempo de execução designado, sua taxa de execução deverá ser aumentada até que ele esteja novamente no tempo certo: enquanto um fluxo está atrasado, os quadros devem aparecer na saída assim que estiverem disponíveis.

20.5.1 Mudanças de escala

Se a adaptação for realizada apenas no destino de um fluxo, a carga em qualquer gargalo no sistema não diminuirá e a situação de sobrecarga persistirá. É interessante adaptar um fluxo para a largura de banda disponível no sistema antes que ele necessite de um recurso em que há gargalos. Isso é conhecido como *mudança de escala*.

A mudança de escala é melhor aplicada quando existem amostras de fluxos ao vivo. Para fluxos armazenados, a facilidade de gerar um fluxo com diferente qualidade depende do método de codificação. A mudança de escala pode ser problemática, caso o fluxo inteiro tenha de ser descompactado e novamente codificado apenas para esse propósito. Os algoritmos de mudança de escala são dependentes da mídia, embora a estratégia de mudança de escala global seja a mesma: fazer uma nova amostragem de determinado

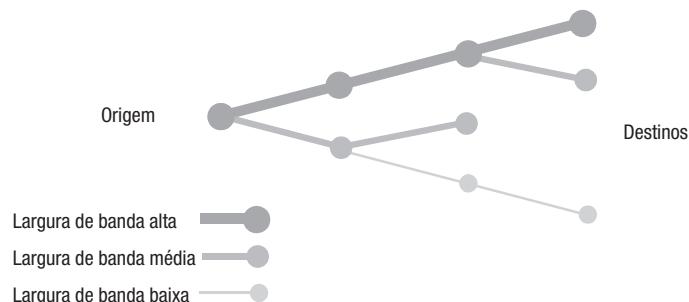


Figura 20.8 Filtragem.

sinal. Para informações de áudio, essa nova amostragem pode ser obtida reduzindo-se a taxa de amostragem de áudio. Ela também pode ser obtida eliminando-se um canal em uma transmissão em estéreo. Conforme esse exemplo mostra, diferentes métodos de mudança de escala podem trabalhar com diferentes granularidades.

Para vídeo, os seguintes métodos de mudança de escala são apropriados:

Mudança de escala temporal: reduz a resolução do fluxo de vídeo no domínio tempo, diminuindo o número de quadros de vídeo transmitidos dentro de um intervalo. A mudança de escala temporal é mais conveniente para fluxos de vídeo em que os quadros individuais são autocontidos e podem ser acessados independentemente. As técnicas de compactação delta são mais difíceis de manipular, pois nem todos os quadros podem ser eliminados facilmente. Portanto, a mudança de escala temporal é mais conveniente para Motion JPEG do que para fluxos MPEG.

Mudança de escala espacial: reduz o número de *pixels* de cada imagem em um fluxo de vídeo. Para a mudança de escala espacial, a organização hierárquica é ideal, pois o vídeo compactado está disponível imediatamente, em várias resoluções. Portanto, o vídeo pode ser transferido pela rede usando diferentes resoluções, sem gravação de cada imagem antes de finalmente transmiti-la. JPEG e MPEG-2 suportam diferentes resoluções espaciais de imagens e são muito convenientes para esse tipo de mudança de escala.

Mudança de escala de frequência: modifica o algoritmo de compactação aplicado a uma imagem. Isso resulta em alguma perda de qualidade, mas em um cenário típico, a compactação pode ser aumentada significativamente, antes que uma redução da qualidade da imagem se torne visível.

Mudança de escala de amplitude: reduz as intensidades das cores de cada *pixel* da imagem. Este método de mudança de escala é usado nas codificações H.261 para chegar a uma vazão (*throughput*) constante quando o conteúdo da imagem varia.

Mudança de escala do espaço de cores: reduz o número de entradas no espaço de cores. Uma maneira de perceber a mudança de escala do espaço de cores é trocar de apresentação em cores para escala de tons.

Se necessário, podem ser usadas combinações desses métodos de mudança de escala.

Um sistema para realizar mudança de escala consiste em um processo monitor no destino e um processo de mudança de escala na origem. O monitor controla os tempos de chegada de mensagens em um fluxo. Mensagens atrasadas são uma indicação de um gargalo no sistema. Então, o monitor envia uma mensagem de *diminuir escala* para a origem, e esta

reduz a largura de banda do fluxo. Após algum período de tempo, a origem aumenta a escala do fluxo novamente. Se o gargalo ainda existir, o monitor detectará novamente um atraso e reduzirá a escala do fluxo [Delgrossi *et al.* 1993]. Um problema para o sistema de mudança de escala é evitar as operações de *aumento de escala* desnecessárias e a oscilação do sistema.

20.5.2 Filtragem

Como a mudança de escala modifica um fluxo na origem, ela nem sempre é conveniente para aplicações que envolvam vários receptores: quando ocorre um gargalo na rota para um destino, esse destino envia uma mensagem de *diminuir escala* para a origem e todos os destinos recebem a qualidade degradada, embora alguns não tivessem problemas para manipular o fluxo original.

A filtragem é um método que fornece a melhor qualidade de serviço possível para cada destino, aplicando mudança de escala em cada nó relevante no caminho da origem para o destino (Figura 20.8). O RSVP (*Resource reservation protocol*) [Zhang *et al.* 1993] é um exemplo de protocolo de negociação de qualidade de serviço que suporta filtragem. A filtragem exige que um fluxo seja dividido em um conjunto de subfluxos hierárquicos, cada um adicionando um nível mais alto de qualidade. A capacidade dos nós em um caminho determina o número de subfluxos recebidos por um destino. Todos os outros subfluxos são filtrados o mais próximo da origem possível (talvez até na origem), para evitar a transferência de dados que posteriormente serão jogados fora. Um subfluxo não é filtrado em um nó intermediário, caso exista um caminho em algum lugar mais adiante que possa transportar o subfluxo inteiro.

20.6 Estudos de caso: Tiger, BitTorrent e End System Multicast

Conforme discutido no Capítulo 1, a multimídia é uma tendência básica nos sistemas distribuídos modernos. Em vez de ser uma área propriamente dita, a multimídia é mais considerada como algo que permeia todos os sistemas distribuídos e, assim, apresenta desafios que precisam ser levados em consideração no projeto de todos os aspectos desses sistemas. Nesta seção, apresentaremos estudos de caso que ilustram como a multimídia influencia três áreas importantes do desenvolvimento de sistemas distribuídos:

- o projeto de um sistema de arquivos distribuído para suportar arquivos de vídeo (*o servidor de arquivos de vídeo Tiger*);
- o projeto de um sistema de *download peer-to-peer* destinado a suportar arquivos multimídia muito grandes (*BitTorrent*);
- o projeto de um serviço de *fluxos multicast* em tempo real (*End System Multicast*).

Note que já vimos outro exemplo de serviço multimídia, quando examinamos a estrutura de rede de sobreposição adotada pelo Skype (veja a Seção 4.5.2).

20.6.1 O servidor de arquivos de vídeo Tiger

Um sistema de armazenamento de vídeos que fornece múltiplos fluxos de vídeo em tempo real simultaneamente é considerado um importante componente de sistema para suportar aplicações multimídia orientadas para o consumidor. Vários protótipos de sistemas desse tipo foram desenvolvidos e alguns até se transformaram em produtos (consulte [Cheng 1998]). Um dos mais avançados é o servidor de arquivos de vídeo Tiger, desenvolvido no Microsoft Research Labs [Bolosky *et al.* 1996].

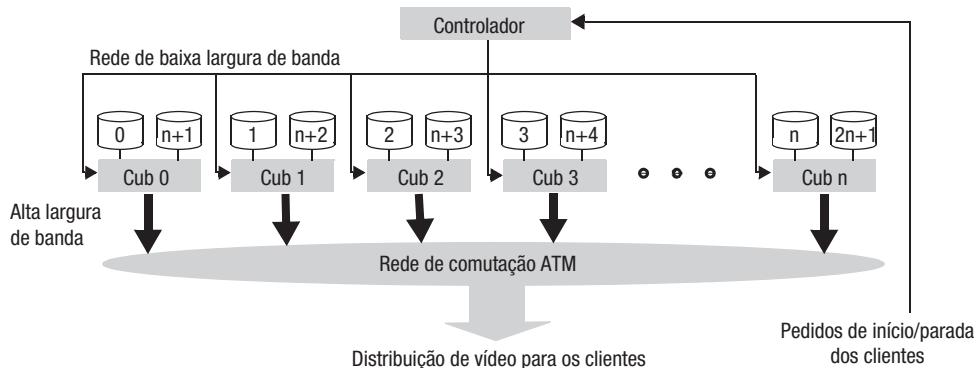


Figura 20.9 Configuração de *hardware* do servidor de arquivos de vídeo Tiger.

Objetivos de projeto • Os principais objetivos de projeto do sistema foram os seguintes:

Video sob demanda para um grande número de usuários: a aplicação típica é um serviço que fornece filmes para clientes pagantes. Os filmes são selecionados em uma grande biblioteca de filmes digitais armazenados. Os clientes devem receber os primeiros quadros de seus filmes escolhidos dentro de poucos segundos após emitirem um pedido e devem ser capazes de executar operações de pausa, retrocesso e avanço rápido à vontade. Embora a biblioteca de filmes disponíveis seja grande, alguns filmes podem ser muito populares e serão motivo de vários pedidos não sincronizados, resultando em várias reproduções concorrentes, porém deslocadas no tempo.

Qualidade do serviço: os fluxos de vídeo devem ser fornecidos a uma velocidade constante, com uma flutuação (*jitter*) máxima determinada pela quantidade (presumida como sendo pequena) de uso de *buffers* disponíveis nos clientes e com uma taxa de perda muito baixa.

Sistema com escalabilidade e distribuído: o objetivo era projetar um sistema com uma arquitetura extensível (pela adição de computadores) para suportar até 10.000 clientes simultaneamente.

Hardware de baixo custo: o sistema era para ser construído usando *hardware* de baixo custo (PCs comerciais com unidades de disco padrão).

Tolerante a falhas: o sistema deveria continuar a funcionar sem degradação notável após a falha de qualquer computador servidor ou unidade de disco.

Tomados em conjunto, esses requisitos exigem uma estratégia radical para o armazenamento e a recuperação de dados de vídeo e um algoritmo de escalonamento eficiente, que harmonize a carga de trabalho entre um grande número de servidores semelhantes. A principal tarefa é a transferência de fluxos de dados de vídeo de largura de banda alta do armazenamento em disco para uma rede, e é essa carga que precisa ser compartilhada entre os servidores.

Arquitetura • A arquitetura de *hardware* do Tiger aparece na Figura 20.9. Todos os componentes são produtos de prateleira. Os computadores *filhotes* (*cub*) mostrados na figura são PCs idênticos, com o mesmo número de unidades de disco rígido padrão (normal-

mente, entre 2 e 4) ligadas em cada um. Eles também estão equipados com placas de rede Ethernet e ATM (veja o Capítulo 3). O *controlador* é outro PC. Ele não está envolvido na manipulação de dados multimídia e é responsável apenas por manipular os pedidos de clientes e pelo gerenciamento das programações de execução dos *cubs*.

Organização do armazenamento • O principal problema de projeto é a distribuição dos dados de vídeo entre os discos ligados aos *cubs* para permitir que eles compartilhem a carga. Como a carga pode envolver o fornecimento de vários fluxos do mesmo filme, assim como o fornecimento de fluxos de muitos filmes diferentes, é improvável que qualquer solução baseada no uso de um único disco para armazenar cada filme atinja o objetivo. Em vez disso, os filmes são armazenados em uma representação segmentada (*strips*) entre todos os discos. Isso leva a um modelo de falha no qual a perda de um disco, ou de um *cub*, resulta em uma lacuna na sequência de cada filme. Isso é tratado por meio de um esquema de espelhamento do armazenamento que replica os dados e de um mecanismo de tolerância a falhas, conforme descrito a seguir.

Segmentação (stripping): um filme é dividido em *blocos* (trechos de vídeo com tempo de reprodução igual, normalmente em torno de 1 segundo, ocupando cerca de 0,5 Mbytes) e o conjunto de blocos que compõem um filme (normalmente, cerca de 7.000 deles para um filme de duas horas) é armazenado nos discos ligados aos diferentes *cubs*, em uma sequência indicada pelos números de disco mostrados na Figura 20.9. Um filme pode começar em qualquer disco. Quando o disco de número mais alto é atingido, o filme “circula”, de modo que o próximo bloco é armazenado no disco 0 e o processo continua.

Espelhamento: o esquema de espelhamento divide cada bloco em várias partes, chamadas de *secundários*. Isso garante que, quando um *cub* falhar, a carga de trabalho extra do fornecimento de dados para blocos no *cub* defeituoso recaia sobre vários dos *cubs* restantes e não apenas sobre um deles. O número de secundários por bloco é determinado por um *fator de desagrupamento*, *d*, com valores típicos no intervalo de 4 a 8. Os secundários de um bloco armazenado no disco *i* são armazenados nos discos *i* + 1 a *i* + *d*. Note que, desde que existam mais do que *d cubs*, nenhum desses discos é ligado no mesmo *cub* que o disco *i*. Com um fator de desagrupamento igual a 8, aproximadamente 7/8 da capacidade de processamento e da largura de banda de disco dos *cubs* pode ser alocada para tarefas isentas de falha. Os 1/8 restantes de seus recursos devem ser suficientes para servir secundários, quando necessário.

Escalonamento distribuído • O centro do projeto Tiger é o escalonamento da carga de trabalho dos *cubs*. O escalonamento é organizado como uma lista de *repartições (slots)*, em que cada repartição representa o trabalho que deve ser feito para reproduzir um bloco de um filme – isto é, lê-lo do disco relevante e transferi-lo para a rede ATM. Existe exatamente uma repartição para cada cliente em potencial receber um filme (chamado de *visualizador*), e cada repartição ocupada representa um visualizador recebendo um fluxo de dados de vídeo em tempo real. O estado do visualizador é representado no escalonamento pelo(a):

- endereço do computador cliente;
- identidade do arquivo que está sendo reproduzido;
- posição do visualizador no arquivo (o próximo bloco a ser distribuído no fluxo);
- número de sequência de exibição do visualizador (a partir do qual pode ser calculado um tempo de distribuição para o próximo bloco);
- por algumas informações de contabilidade.

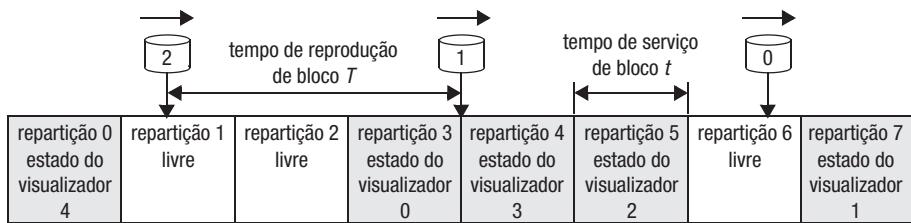


Figura 20.10 Escalonamento Tiger.

O escalonamento está ilustrado na Figura 20.10. O *tempo de reprodução do bloco T* é o tempo que será exigido para um visualizador exibir um bloco no computador cliente; normalmente, cerca de 1 segundo, e supõe-se ser o mesmo para todos os filmes armazenados. Portanto, o Tiger deve manter um intervalo de tempo T entre os tempos de distribuição dos blocos em cada fluxo, com uma pequena flutuação (*jitter*) permitida, que é determinada pelo uso de *buffers* disponíveis nos computadores clientes.

Cada *cub* mantém um ponteiro para o escalonamento de cada disco que controla. Durante cada tempo de reprodução de bloco, ele deve processar todas as repartições com números de bloco que caiam nos discos que controla, e tempos de distribuição que caiam dentro do tempo de reprodução de bloco corrente. O *cub* percorre o escalonamento em repartições de processamento em tempo real, como segue:

1. Lê o próximo bloco no armazenamento em *buffer* no *cub*.
2. Empacota o bloco e o envia para o controlador de rede ATM do *cub*, com o endereço do computador cliente.
3. Atualiza o estado do visualizador no escalonamento para mostrar o novo próximo bloco e exibir o número de sequência, e passa a repartição atualizada para o próximo *cub*.

Supõe-se que essas ações ocupam um tempo máximo t , que é conhecido como tempo de serviço de bloco. Conforme pode ser visto na Figura 20.10, t é substancialmente menor do que o tempo de reprodução de um bloco. O valor de t é determinado pela largura de banda de disco ou pela largura de banda de rede, o que for menor. (Os recursos de processamento em um *cub* são adequados para executar o trabalho programado de todos os discos anexos.) Quando um *cub* tiver concluído as tarefas programadas para o tempo de reprodução corrente do bloco, ele estará disponível para tarefas não programadas até o início do próximo tempo de reprodução. Na prática, os discos não fornecem blocos com um atraso fixo e, para acomodar sua distribuição desigual, a leitura do disco é iniciada pelo menos um tempo de serviço de bloco antes que o bloco seja necessário para empacotamento e distribuição.

Um disco pode fazer o trabalho de atender T/t fluxos e os valores de T e t normalmente resultam em um valor > 4 para essa relação. Isso e o número de discos no sistema inteiro determinam o número de visualizadores a que um sistema Tiger pode atender. Por exemplo, um sistema Tiger com cinco *cubs* e três discos ligados a cada um pode distribuir aproximadamente 70 fluxos de vídeo simultaneamente.

Tolerância a falhas • Devido à segmentação de todos os arquivos de filme entre todos os discos em um sistema Tiger, a falha de qualquer componente (uma unidade de disco ou um *cub*) resultaria em uma interrupção do serviço para todos os clientes. O projeto do Tiger resolve isso por meio da recuperação de dados das cópias secundárias espelhadas, quando um bloco primário está indisponível devido à falha de um *cub* ou de uma unidade de disco. Lembre-se de que os blocos secundários são menores do que os primários na

relação do fator de desagrupamento d e que os secundários são distribuídos de modo a caírem em vários discos ligados em diferentes *cubs*.

Quando um *cub* ou um disco falha, o escalonamento é modificado por um *cub* adjacente para mostrar vários *estados do visualizador espelho*, representando a carga de trabalho dos d discos que contêm os secundários desses filmes. Um estado do visualizador espelho é semelhante ao estado de um visualizador normal, mas com diferentes números de bloco e requisitos de temporização. Como essa carga de trabalho extra é compartilhada entre d discos e d *cubs*, ela pode ser acomodada sem interromper as tarefas nas outras repartições, desde que exista uma pequena quantidade de capacidade sobressalente no escalonamento. A falha de um *cub* é equivalente à falha de todos os discos ligados a ele e é tratada de maneira semelhante.

Suporte de rede • Os blocos de cada filme são simplesmente passados para a rede ATM pelos *cubs* que os contêm, junto ao endereço do cliente relevante. Depende-se das garantias de qualidade de serviço dos protocolos de rede ATM para distribuir os blocos para os computadores clientes em sequência e a tempo. O cliente precisa de armazenamento suficiente em *buffer* para conter dois blocos primários, o que está sendo reproduzido no momento na tela do cliente e o que está chegando da rede. Quando os blocos primários estiverem sendo servidos, o cliente só precisará verificar o número de sequência de cada bloco recebido e passá-lo para a rotina de exibição. Quando os secundários estiverem sendo servidos, os d *cubs* responsáveis por um bloco desagrupado distribuem seus secundários para a rede na sequência, e é responsabilidade do cliente reuni-los e montá-los em seu armazenamento em *buffer*.

Outras funções • Descrevemos as atividades críticas quanto ao tempo de um servidor Tiger. Os requisitos de projeto exigidos para fornecimento de funções de avanço rápido e retrocesso. Essas funções exigem a distribuição de uma parte dos blocos do filme para o cliente, para dar o retorno visual normalmente fornecido pelos gravadores de vídeo. Isso é feito com base no melhor esforço por parte dos *cubs*, em um tempo não programado para execução.

As tarefas restantes incluem o gerenciamento e a distribuição do escalonamento e o gerenciamento do banco de dados de filmes, a exclusão de filmes antigos e a gravação de novos nos discos e a manutenção de um índice de filmes.

A implementação inicial, o gerenciamento do escalonamento e a distribuição do Tiger eram manipulados pelo computador controlador. Como isso constituía um único ponto de falha e um gargalo de desempenho em potencial, o gerenciamento do escalonamento foi subsequentemente refeito como um algoritmo distribuído [Bolosky *et al.* 1997]. O gerenciamento do banco de dados de filmes é realizado pelos *cubs* em um tempo não programado, em resposta aos comandos do controlador.

Desempenho e escalabilidade • O protótipo inicial foi desenvolvido em 1994 e usava cinco PCs Pentium de 133MHz, cada um equipado com 48 Mbytes de memória RAM, três unidades de disco SCSI de 2 Gbytes e um controlador de rede ATM, executando Windows NT. Essa configuração foi medida sob uma carga de cliente simulada. Ao servir filmes para 68 clientes, sem falhas no sistema Tiger, a distribuição dos dados era perfeita – nenhum bloco era perdido, nem distribuído com atraso para os clientes. Com um *cub* danificado (e, portanto, três discos), o serviço era mantido com uma taxa de perda de dados de apenas 0,02%, dentro do objetivo de projeto.

Outra medida feita foi a latência de inicialização para distribuir o primeiro bloco de um filme, após a recepção do pedido de um cliente. Isso será altamente dependente do número e da posição das repartições livres no escalonamento. O algoritmo usado para

isso colocaria inicialmente o pedido de um cliente na repartição livre mais próxima, no disco contendo o bloco 0 do filme solicitado. Isso resultou em valores medidos para a latência de inicialização no intervalo de 2 a 12 segundos. Um trabalho recente resultou em um algoritmo de alocação de repartição que reduz o agrupamento de repartições ocupadas no escalonamento, deixando as repartições livres distribuídas mais igualmente na programação e melhorando a latência de inicialização média [Douceur e Bolosky 1999].

Embora as experiências iniciais tenham sido feitas com uma configuração pequena, foram feitas medidas posteriores com uma configuração de 14 *cubs*, 56 discos e o esquema de escalonamento distribuído descrito por Bolosky *et al.* [1997]. A carga que poderia ser servida por esse sistema mudou de escala com sucesso, para distribuir 602 fluxos de dados de 2 Mbps simultâneos, com uma taxa de perda de menos de um bloco em 180.000, quando todos os *cubs* estavam funcionando. Com um *cub* defeituoso, menos de 1 em 40.000 blocos eram perdidos. Esses resultados são impressionantes e parecem corroborar a afirmação de que um sistema Tiger poderia ser configurado com até 1.000 *cubs*, servindo até 30.000–40.000 visualizadores simultâneos.

20.6.2 BitTorrent

O BitTorrent [www.bittorrent.com] é uma aplicação de compartilhamento de arquivos *peer-to-peer* popular, projetada especificamente para o *download* de arquivos grandes (incluindo arquivos de vídeo). Ele não se destina aos fluxos de conteúdo em tempo real, mas sim ao *download* inicial de arquivos a serem reproduzidos posteriormente. O BitTorrent foi mencionado brevemente no Capítulo 10, como um exemplo de protocolo de compartilhamento de arquivos *peer-to-peer*. Neste capítulo, vamos examinar o projeto do BitTorrent mais detalhadamente, dando ênfase ao suporte fornecido para o *download* de arquivos de vídeo.

A principal característica de projeto no BitTorrent é a divisão dos arquivos em trechos de tamanho fixo e sua subsequente disponibilidade em vários *sites* pela rede *peer-to-peer*. Os clientes podem, então, fazer o *download* de vários trechos em paralelo, a partir de diferentes *sites*, reduzindo a carga de qualquer *site* em particular para atender ao *download* (lembrando que o BitTorrent conta com os recursos de máquinas de usuário normais e também que pode haver muitas solicitações simultâneas de arquivos populares). Isso é melhor do que as estratégias mais centralizadas, nas quais um cliente faria o *download* de um arquivo de um servidor usando, por exemplo, HTTP.

O protocolo BitTorrent funciona como segue. Quando um arquivo se torna disponível no BitTorrent, é criado um arquivo *.torrent* que contém metadados associados a esse arquivo, incluindo:

- o nome e o comprimento do arquivo;
- a localização de um *rastreador* (especificado como um URL) – um servidor centralizado que gerencia os *downloads* desse arquivo em particular;
- uma *soma de verificação* (*checksum*) associada a cada trecho, gerada pelo algoritmo *hashing* SHA-1, a qual permite que o conteúdo seja verificado após o *download*.

O uso de rastreadores é uma transigêncie em relação aos princípios *peer-to-peer* puros, mas isso permite que o sistema mantenha as informações acima facilmente e de maneira centralizada.

Os rastreadores são responsáveis por monitorar o status do *download* associado a um arquivo em particular. Para se entender as informações mantidas pelo rastreador, é necessário voltar e considerar o ciclo de vida de determinado arquivo.

<i>Termo</i>	<i>Significado</i>
arquivo <i>.torrent</i>	Arquivo que mantém metadados sobre um arquivo disponível
rastreador	Servidor contendo informações sobre os <i>downloads</i> em andamento
trecho	Parte de tamanho fixo de determinado arquivo
<i>seeder</i> (semeador)	Par que contém uma cópia completa de um arquivo (consistindo em todos os seus trechos)
<i>leecher</i> (parasita)	Par envolvido no <i>download</i> de um arquivo e que, no momento, contém apenas uma parte de seus trechos
torrente (ou enxame)	Conjunto de <i>sites</i> envolvidos no <i>download</i> de um arquivo, incluindo o rastreador, os <i>seeders</i> e os <i>leechers</i>
<i>tit-for-tat</i> (olho por olho)	Mecanismo de incentivo que governa o escalonamento de <i>downloads</i> no BitTorrent
desafogamento otimista	Mecanismo para permitir que novos pares estabeleçam suas credenciais
mais raro primeiro	Esquema de escalonamento por meio do qual o BitTorrent prioriza os trechos raros dentro de seu conjunto de pares conectados

Figura 20.11 Terminologia do BitTorrent.

Qualquer par (*peer*) com uma versão completa de um arquivo (em termos de todos os seus trechos) é conhecido como *seeder* (*semeador*) na terminologia do BitTorrent. Por exemplo, o par que cria o arquivo fornece a semente inicial para sua distribuição. Os pares que desejam fazer *download* de um arquivo são conhecidos como *leechers* (*parasitas*), sendo que, a qualquer momento, determinado *leecher* vai conter vários trechos associados a esse arquivo. Quando um *leecher* faz o *download* de todos os trechos associados a um arquivo, ele se torna um *seeder* para *downloads* subsequentes. Desse modo, os arquivos se espalham como vírus pela rede, com a difusão estimulada pela demanda. Com base nisso, o rastreador mantém informações sobre o estado atual dos *downloads* de determinado arquivo, em termos dos *seeders* e *leechers* associados. O rastreador, junto aos *seeders* e *leechers* associados, são referidos no BitTorrent como *torrente* (ou *enxame*) desse arquivo. (Um resumo da terminologia do BitTorrent pode ser encontrada na Figura 20.11.)

Quando um par quer fazer o *download* de um arquivo, ele primeiro entra em contato com o rastreador e obtém uma visão parcial da torrente, em termos do conjunto de pares que podem suportar o *download*. Depois disso, o trabalho do rastreador está terminado – ele não se envolve no escalonamento de *downloads* subsequente. Esse é um assunto para os vários pares envolvidos e, assim, essa parte do protocolo é descentralizada. Então, os trechos são pedidos e transmitidos em qualquer ordem para o par solicitante (compare isso com o CoolStreaming, apresentado no quadro a seguir).

O BitTorrent, junto a muitos protocolos *peer-to-peer*, conta com os pares se comportando como bons cidadãos, contribuindo e usufruindo o sistema. Fundamentalmente, o sistema tem um mecanismo de incentivos incorporado para recompensar essa cooperação, conhecido como *tit-for-tat* (olho por olho) [Cohen 2003]. Informalmente, essa estratégia dá preferência ao *download* a pares que já fizeram ou estão fazendo *upload* nesse *site*. Além de atuar como mecanismo de incentivo, o *tit-for-tat* também estimula padrões de comunicação em que o *download* e o *upload* acontecem concomitantemente, fazendo o melhor uso da largura de banda.

Em mais detalhes, determinado par suporta o *download* de n pares simultâneos por *desafogar* esses pares. As decisões sobre quais pares vão ser desafogados são baseadas

em cálculos de suas taxas de *download*, com a decisão revista a cada 10 segundos. O algoritmo também aplica *desafogamento otimista* em um par aleatório a cada 30 segundos, para permitir que novos pares participem e estabeleçam suas credenciais. Note que o esquema de incentivos tem sido o tema de pesquisa significativa, com esquemas alternativos também propostos – para exemplos, consulte Sirivianos *et al.* [2007]. O BitTorrent acopla isso à política do *mais raro primeiro* para escalonamento de *downloads*, por meio da qual um par prioriza o trecho mais raro em seu conjunto de pares conectados, garantindo que os trechos ainda não prontamente disponíveis sejam espalhados rapidamente.

20.6.3 End System Multicast (ESM)

Um dos maiores desafios técnicos nos sistemas multimídia distribuídos é suportar transmissão de vídeo em tempo real pela Internet. Esses sistemas são exigentes por diversas razões [Liu *et al.* 2008]:

- Os sistemas devem *possuir escalabilidade* para números de usuários potencialmente muito grandes;
- Eles são particularmente exigentes em termos de *utilização de recursos*, impondo restrições significativas de largura de banda, armazenamento e processamento no sistema;
- Rigorosos *requisitos de tempo real* devem ser satisfeitos para que a experiência do usuário seja satisfatória;
- Os sistemas devem ser *flexíveis* e capazes de se *adaptar* a condições variáveis na rede.

Apesar desses desafios, foram feitos avanços significativos e atualmente vários serviços comerciais estão disponíveis, incluindo o BBC iPlayer, o BoxeeTV [[boxee.tv](#)] e o Hulu [[hulu.com](#)]. Nesta seção, apresentaremos um exemplo de sistema influente nessa área: o End System Multicast [Liu *et al.* 2008], desenvolvido na CMU e agora comercializado pela Conviva [[www.conviva.com](#)]. Antes de examinarmos a estratégia técnica defendida pelo ESM, é interessante contextualizar este trabalho.

Contexto do ESM • As primeiras experiências em fluxos de vídeo pela Internet foram feitas diretamente sobre *multicast IP*, conforme descrito na Seção 4.4.1. Essa estratégia tem a vantagem de o suporte para *multicast* ser oferecido diretamente em uma camada baixa no sistema, contribuindo, assim, para o desempenho global. No entanto, a estratégia tem vários inconvenientes, incluindo a falta de suporte para *multicast IP* em muitos roteadores e a necessidade de manter estado tolerante nos roteadores para suportar *multicast*. Mais fundamentalmente, isso também transgride o princípio fim-a-fim, discutido na Seção 2.3.3, o qual defende que o suporte para funções de comunicação (neste caso, fluxos multimídia) só pode ser implementado completamente e de forma confiável com o conhecimento e a ajuda da aplicação que está nos pontos extremos do sistema de comunicação.

Como resultado, agora a maioria dos sistemas defende *estratégias de sistema final (end-system)* para fluxos de vídeo, em que o controle e a inteligência residem nas extremidades da rede e não nela própria. Essa estratégia também é chamada de *multicast em nível de aplicação* e implica na formação de uma rede de sobreposição para suportar o tráfego multimídia associado (veja na Seção 4.5 uma discussão sobre redes de *overlay*).

Levando isso um passo adiante, há um interesse considerável nas estratégias *peer-to-peer* para suportar transmissão multimídia na Internet, e o ESM é um exemplo importante de tal sistema. Mais especificamente, o ESM emprega técnicas *peer-to-peer* estruturadas pelas quais os próprio pares formam uma estrutura em árvore para a subsequente distribuição da mídia em tempo real. Como uma alternativa à estratégia estruturada de-

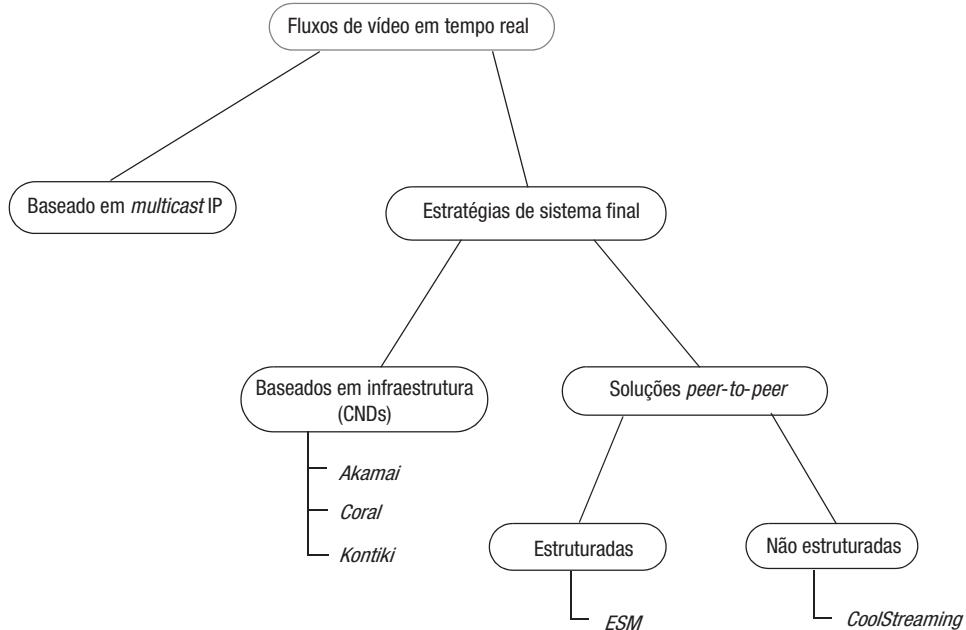


Figura 20.12 Estratégias de fluxo de vídeo em tempo real.

fendida pelo ESM, o CoolStreaming oferece uma estratégia não estruturada, ampliando a ideia do BitTorrent, discutido na Seção 20.6.2 (veja os detalhes no quadro a seguir).

Vários sistemas adotam a metodologia do sistema final, mas, em vez de uma estratégia *peer-to-peer*, oferecem uma infraestrutura fixa para manter várias cópias do conteúdo multimídia (ou outro) localizado em nós por toda a Internet, suportando, assim, uma distribuição mais rápida. Esses sistemas são referidos como *redes de distribuição de conteúdo* (ou CDNs, Content Distribution Networks). Os principais exemplos são o Akamai [www.akamai.com], o Coral [www.coralcdn.org] e o Kontiki [www.kontiki.com]. Tais sistemas suportam uma variedade de estilos de distribuição de conteúdo, incluindo aceleração da Web (maior desempenho no acesso ao conteúdo da Web) e fluxos de vídeo – por exemplo, o Kontiki foi usado na versão original do BBC iPlayer.

As diversas técnicas para suportar fluxos multimídia em tempo real estão resumidas na Figura 20.12.

Arquitetura do ESM • O ESM (End System Multicast) é uma solução *peer-to-peer* estruturada para *multicast* de vídeo em tempo real pela Internet. Inicialmente, a estratégia foi desenvolvida na Universidade Carnegie Mellon, como parte de uma pesquisa que investigava as propriedades da *estratégia de sistema final* para multicast, e o protótipo inicial foi usado para fluxos de vídeos em tempo real em diversas conferências importantes, incluindo SIGCOMM, INFOCOM e NOSSDAV [esm.cs.cmu.edu]. Conforme mencionado anteriormente, agora a estratégia é comercializada pela Conviva [www.conviva.com], que recentemente fez um acordo com a NBC Universal para utilizar a plataforma Conviva (denominada C3) na distribuição de seu conteúdo pela Internet.

Além de investigar a estratégia de sistema final, outro objetivo importante do ESM é ser flexível às mudanças, por meio de *auto-organização*. Em particular, os protocolos

de base são projetados para lidar elegantemente com o ingresso e a saída dinâmicas dos nós, com a falha de nós e com mudanças na configuração e no desempenho da rede subjacente. Em particular, eles promovem *adaptação com reconhecimento de desempenho*, por meio da qual as estruturas de sobreposição associadas ao sistema *peer-to-peer* são frequentemente reavaliadas para maximizar o desempenho global. Vamos discutir como isso é atingido, a seguir.

O ESM funciona construindo uma árvore para cada fluxo de vídeo, cuja raiz está na origem desse fluxo em particular. Os principais elementos algorítmicos para suportar isso são:

- como manter informações sobre participação como membro;
- como lidar com novos pares ingressando na árvore;
- como lidar com pares saindo da árvore (seja propositalmente ou por causa de falha);
- como adaptar as estruturas em árvore para favorecer o desempenho (adaptação com reconhecimento de desempenho, conforme mencionado anteriormente).

CoolStreaming: uma estratégia P2P não estruturada para fluxos de vídeo

Muitas estratégias de fluxos de vídeo são baseadas em uma estratégia *peer-to-peer* estruturada, construindo uma árvore (como no ESM) ou uma estrutura de sobreposição alternativa, como uma floresta – uma união disjunta de árvores – ou uma malha. O CoolStreaming [Zhang *et al.* 2005b] adota uma estratégia não estruturada radicalmente diferente para fluxos de vídeo, referida por seus criadores como estratégia *centrada em dados*. No CoolStreaming, os nós mantêm uma visão parcial da participação como membro, a qual é atualizada periodicamente por meio de um protocolo de fofoca (como no ESM). Quando um novo par ingressa, ele primeiro entra em contato com o nó de origem do fluxo de vídeo desejado (o qual presumidamente é anunciado) e esse nó escolhe outro aleatoriamente em seu conjunto de membros conhecidos, para atuar como representante (balanceando, assim, a carga entre todos os membros). Então, o novo nó obtém do representante um conjunto inicial de parceiros, carregando-os no sistema. Deve-se enfatizar que, ao contrário de uma estratégia baseada em árvore, esse conjunto de parceiros não significa quaisquer relações pai-filho para o *download*; em vez disso, o escalonamento do *download* é determinado dinamicamente e governado pela disponibilidade de dados, conforme explicado a seguir.

No CoolStreaming, um arquivo de vídeo é decomposto em vários *segmentos* de tamanho fixo, como no BitTorrent. Cada par cria um *mapa de buffer* para indicar a disponibilidade local de um ou mais segmentos de um arquivo e, então, troca essa informação com seus parceiros conhecidos. Essa informação é usada para obter todos os segmentos necessários de determinada origem de vídeo. Até aqui, isso é muito parecido com o BitTorrent, mas com duas diferenças importantes, devidas à necessidade de fluxos em tempo real. Primeiro, enquanto o BitTorrent pode fazer o *download* de trechos em qualquer ordem, o CoolStreaming deve satisfazer as restrições de tempo real desejadas para a reprodução do vídeo. Segundo, em dado momento, o CoolStreaming está interessado apenas em uma *janela de tempo variável*, desde o presente até um período no futuro próximo (na prática, uma janela de tempo variável de 120 segmentos de um segundo), em vez do arquivo inteiro. O cálculo do escalonamento associado é fundamental para o funcionamento do CoolStreaming e, embora encontrar uma solução perfeita seja o conhecido problema polinomial não determinista (*NP-hard*), o CoolStreaming adotou um conjunto bem-sucedido de heurísticas, baseadas em fatores que incluem o número de fornecedores em potencial de um segmento, a largura de banda dos fornecedores e o tempo disponível para processar a requisição.

O resultado final é uma arquitetura de sistemas distribuídos que pode satisfazer os requisitos de tempo real dos fluxos de vídeo e que é mais naturalmente flexível à falha de nós e às mudanças no desempenho ou na disponibilidade da rede.

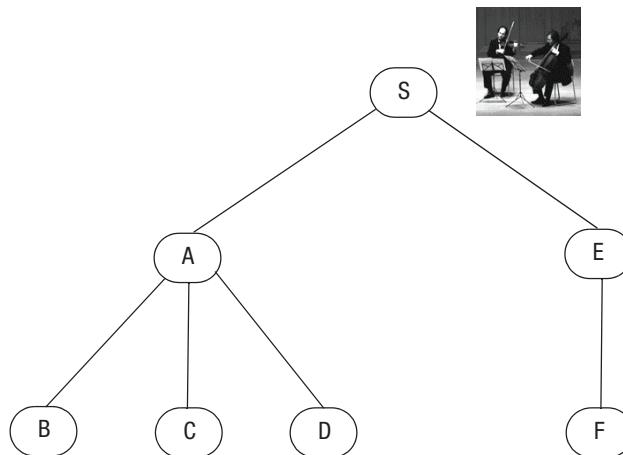


Figura 20.13 Um exemplo de árvore no ESM.

Vamos tratar de cada um desses elementos a seguir. Em cada uma das descrições, vamos nos referir ao exemplo de estrutura em árvore mostrada na Figura 20.13, a qual está transmitindo, em fluxos ao vivo, a apresentação de nossos músicos mencionados anteriormente (Seção 20.1).

Gerenciamento da participação como membro • Cada nó mantém uma visão parcial da participação como membro da árvore, com essa visão atualizada periodicamente com um *protocolo de fofoca* (um método popular para manter a participação como membro de grupo em estruturas *peer-to-peer*, descrito na Seção 18.4). Isso funciona com cada membro escolhendo periodicamente outro do grupo e enviando a ele um subconjunto de sua visão da participação como membro, anotada com informações sobre quando foi a última vez que ele soube de cada membro do grupo (na forma de um carimbo de tempo). Portanto, não há nenhuma tentativa de ter uma visão global consistente das informações sobre participação como membro do grupo, mas essa visão parcial é suficiente para o funcionamento do protocolo, conforme ficará claro a seguir.

Ingressando em uma árvore • É presumido que o nó de origem (a raiz da árvore) é anunciado e, portanto, conhecido pelo sistema. Um novo nó entra em contato com a origem e recebe um conjunto de nós selecionados aleatoriamente, extraídos da visão do grupo mantida pela origem. Esses são efetivamente candidatos a pais do novo nó, o qual deve selecionar um pai adequado nesse conjunto de possibilidades.

O protocolo de *seleção do pai* é fundamental para o funcionamento do ESM, com o objetivo global de otimizar a árvore para favorecer o desempenho (em particular, conforme veremos, para o desempenho de saída (*throughput*), sendo a latência uma consideração secundária).

A primeira fase da seleção do pai é sondar o conjunto de membros fornecido pela origem e coletar as seguintes informações em cada candidato:

- O desempenho que o nó está apresentando atualmente, em termos do *desempenho de saída* e da *latência* da origem;
- a saturação desse nó, em termos do número de filhos que ele já suporta.

A partir de um parâmetro obtido por sondagem, também é possível determinar a latência de ida e volta entre o novo nó e os vários nós candidatos.

O novo nó elimina os candidatos que considera *saturados* (definidos por uma constante interna) e, então, calcula o serviço que pode esperar de cada um dos outros candidatos, em termos de desempenho de saída e atraso. O desempenho de saída é estimado como o mínimo desempenho de saída relatado obtido por esse nó e por dados históricos do novo nó até esse candidato (esses dados podem estar disponíveis se, por exemplo, o novo nó conectou esse candidato em particular anteriormente). O atraso pode ser estimado com base no somatório do atraso relatado da origem e a latência experimentada pela sondagem. A seleção de nós é baseada na melhor largura de banda disponível até o novo nó; se as informações sobre a largura de banda não estão disponíveis, a seleção é feita com base nos valores de latência.

Voltando à árvore de exemplo da Figura 20.13, suponha que um novo nó *G* quer ingressar nesse fluxo de vídeo. *G* entra em contato com o nó de origem *S* e recebe (aleatoriamente) o seguinte conjunto de nós: {*A*, *C*, *E* e *F*}. *A* é eliminado imediatamente, pois é considerado saturado (supondo que a definição de saturação seja ter 3 filhos) e *C* relata características de desempenho de saída ruins, talvez porque *A* esteja saturado. Isso leva a uma escolha entre os nós *E* e *F*. *E* é escolhido por relatar os melhores valores de desempenho de saída disponíveis (talvez o enlace entre *E* e *F* seja por meio de uma conexão de largura de banda relativamente baixa) e, além disso, *G* se conectou a *E* anteriormente, tendo experimentado boas características de desempenho de saída.

Lidando com nós que saem • Os membros podem sair de uma árvore por causa de uma requisição de saída explícita ou devido a uma falha. No primeiro caso, para evitar rompimento, o membro que está saindo notifica os filhos sobre esse fato e deve continuar encaminhando dados por certo período, para evitar a interrupção do serviço mais abaixo na árvore. No último caso, os membros enviam periodicamente mensagens de *ativos* para seus filhos e a falha é detectada quando essas mensagens não são recebidas.

Nos dois casos, todos os filhos devem invocar o procedimento de seleção de pai, conforme definido anteriormente, com verificações extras realizadas para garantir que os candidatos já não sejam descendentes dos nós dados.

Suponha que logo após *G* ingressar na árvore, o nó *E* falhe. Nesse caso, tanto *F* como *G* devem executar o algoritmo de seleção de pai para restabelecer a conectividade.

Adaptação com reconhecimento de desempenho • Cada nó monitora continuamente o serviço que está recebendo de seu nó pai (e, conforme mencionado anteriormente, mantém essas informações de histórico para futura referência). A adaptação é ativada se a taxa detectada cai significativamente abaixo da esperada da origem. Para evitar o descarte (*thrashing*), um nó deve esperar por um período específico, conhecido como *tempo de detecção*, antes de optar por fazer a adaptação.

Uma vez tomada a decisão de adaptar, o nó invocará o algoritmo de seleção de pai para determinar um novo pai mais eficiente. Desse modo, a construção da árvore é constantemente reavaliada e se auto-organizará para otimizar o desempenho global.

Por exemplo, após um período de tempo, *C* pode decidir que o desempenho de saída recebido por meio de *A* é insatisfatório e executar o algoritmo de seleção de pai, resultando em *C* se tornando filho de *E*.

20.7 Resumo

As aplicações multimídia exigem novos mecanismos de sistema para permitir que eles manipulem grandes volumes de dados dependentes do tempo. Os mais importantes desses mecanismos se preocupam com o gerenciamento de qualidade de serviço. Eles devem alocar largura de banda e outros recursos, de uma maneira que garanta que os requisitos de recursos da aplicação sejam atendidos e devem programar o uso dos recursos para que os prazos finais exigentes das aplicações multimídia sejam atendidos.

O gerenciamento de qualidade de serviço trata as requisições de qualidade de serviço das aplicações, especificando a largura de banda, a latência e taxas de perda aceitáveis para os fluxos multimídia, e realiza o controle de admissão, determinando se há recursos disponíveis suficientes para atender a cada nova requisição e negociar com a aplicação, se necessário. Uma vez que uma requisição de qualidade de serviço seja aceita, os recursos são reservados e uma garantia é emitida para a aplicação.

A capacidade do processador e a largura de banda de rede alocada para uma aplicação devem, então, ser programadas para atender a suas necessidades. Um algoritmo de escalonamento de processador em tempo real, como o *earliest-deadline-first* (EDF) ou o *rate-monotonic* (RM), é exigido para garantir que cada elemento do fluxo seja processado a tempo.

Conformação de tráfego é o nome dado aos algoritmos que colocam dados no *buffer* em tempo real para suavizar as irregularidades da temporização que surgem inevitavelmente. Os fluxos podem ser adaptados para utilizar menos recursos, reduzindo a largura de banda da origem (mudança de escala) ou em pontos ao longo do caminho (filtragem).

O servidor de arquivos de vídeo Tiger é um exemplo excelente de sistema com capacidade de escalabilidade que fornece distribuição de fluxo em uma escala potencialmente muito grande, com fortes garantias de qualidade do serviço. Seu escalonamento de recurso é altamente especializado e ele oferece um exemplo excelente da estratégia de projeto alterada que frequentemente é exigida para tais sistemas. Os outros dois estudos de caso, BitTorrent e ESM, também fornecem sólidos exemplos de como suportar, respectivamente, o *download* e o fluxo de dados de vídeo em tempo real, novamente destacando o impacto que a multimídia tem no projeto de sistemas.

Exercícios

- 20.1 Descreva, em linhas gerais, um sistema para suportar um recurso de ensaio musical distribuído. Sugira os requisitos de qualidade de serviço convenientes e uma configuração de *hardware* e *software* que possa ser usada. *páginas 884, 889*
- 20.2 Atualmente, a Internet não oferece nenhuma facilidade de reserva de recurso ou de gerenciamento de qualidade de serviço. Como as aplicações de fluxo de áudio e vídeo baseados na Internet existentes obtêm qualidade aceitável? Quais limitações as soluções que eles adotam impõem às aplicações multimídia? *páginas 884, 893, 899*
- 20.3 Explique as diferenças entre as três formas de sincronização (estado distribuído síncrono, sincronização de mídia e sincronização externa) que podem ser exigidas nas aplicações multimídia distribuídas. Sugira mecanismos por meio dos quais cada uma delas poderia ser obtida, por exemplo, em uma aplicação de videoconferência. *página 885*

- 20.4 Descreva, em linhas gerais, o projeto de um gerenciador de qualidade de serviço para permitir que computadores *desktop* conectados por uma rede ATM suportem várias aplicações multimídia concorrentes. Defina uma API para seu gerenciador de qualidade de serviço, fornecendo as principais operações, com seus parâmetros e resultados. *páginas 889–891*
- 20.5 Para especificar os componentes de *software* dos requisitos de recurso que processam dados multimídia, precisamos de estimativas de suas cargas de processamento. Como essa informação pode ser obtida sem trabalho desnecessário? *páginas 889–891*
- 20.6 Como o sistema Tiger suporta um grande número de clientes, todos solicitando o mesmo filme em momentos aleatórios? *páginas 901–906*
- 20.7 A escalonamento do Tiger é potencialmente uma grande estrutura de dados que muda frequentemente, mas cada *cub* precisa de uma representação atualizada das partes que está manipulando no momento. Sugira um mecanismo para a distribuição do escalonamento dos *cubs*. *páginas 901–906*
- 20.8 Quando o Tiger está operando com um disco, ou com um *cub* defeituoso, blocos de dados secundários são usados no lugar dos primários ausentes. Os blocos secundários são n vezes menores do que os primários (onde n é o fator de desagrupamento). Como o sistema acomoda essa variabilidade no tamanho do bloco? *página 905*
- 20.9 Discuta os méritos relativos da política de *download* do *mais raro primeiro* no BitTorrent, em comparação com a estratégia de *download* sequencial mais tradicional. *páginas 906–908*
- 20.10 Além do Tiger (veja o Exercício 20.6), o ESM e o CoolStreaming também suportam acesso por fluxo ao mesmo filme, por, potencialmente, grandes números de usuários. Discuta as estratégias adotadas pelo ESM e pelo CoolStreaming para gerenciar esse acesso simultâneo e compare-as com as defendidas pelo Tiger. *páginas 901–906, 908–912*

21

Projeto de Sistemas Distribuídos – Estudo de Caso: Google

- 21.1 Introdução
- 21.2 Introdução ao estudo de caso: Google
- 21.3 Arquitetura global e filosofia de projeto
- 21.4 Paradigmas de comunicação
- 21.5 Serviços de armazenamento de dados e coordenação
- 21.6 Serviços de computação distribuída
- 21.7 Resumo

A capacidade de criar um projeto eficiente é uma habilidade importante em sistemas distribuídos, exigindo o conhecimento das diferentes escolhas tecnológicas apresentadas por todo o livro e um entendimento completo dos requisitos do domínio de aplicação relevante. O objetivo final é propor uma arquitetura de sistema distribuído coerente, que incorpore um conjunto consistente e completo de escolhas de projeto capazes de atender aos requisitos globais. Essa é uma tarefa desafiadora, que exige experiência considerável no desenvolvimento de sistemas distribuídos.

Ilustraremos o projeto distribuído por meio de um estudo de caso de peso, examinando em detalhes o projeto da infraestrutura do Google, uma plataforma e *middleware* associado que suportam a pesquisa no Google e um conjunto de serviços e aplicativos Web associados, incluindo o Google Apps. Isso engloba o estudo de importantes componentes subjacentes, como a infraestrutura física que serve de alicerce para o Google, os paradigmas de comunicação oferecidos pela infraestrutura do Google e os serviços básicos associados para armazenamento e computação.

É dada ênfase nos principais princípios de projeto por trás da infraestrutura do Google e em como eles permeiam a arquitetura de sistema global, resultando em um projeto consistente e eficiente.

21.1 Introdução

Este livro abordou os principais conceitos que servem de base para o desenvolvimento de sistemas distribuídos, dando ênfase ao tratamento dos principais desafios dos sistemas distribuídos, incluindo heterogeneidade, abertura, segurança, escalabilidade, tratamento de falhas, concorrência, transparência e qualidade de serviço. O tratamento subsequente inevitavelmente se concentra nas partes que compõem um sistema distribuído, incluindo paradigmas de comunicação, como a invocação remota e suas alternativas indiretas; as abstrações de programação oferecidas pelos objetos, componentes ou serviços Web; serviços específicos dos sistemas distribuídos para suporte à segurança, atribuição de nomes e sistema de arquivos; e soluções algorítmicas, como coordenação e acordo. Contudo, é igualmente importante considerar o projeto global dos sistemas distribuídos e como as partes componentes são reunidas, tratando dos inevitáveis compromissos entre os diferentes desafios para produzir uma arquitetura de sistema global que satisfaça os requisitos de um domínio de aplicação e um ambiente operacional de larga escala. Um tratamento mais completo dos métodos de projeto de sistemas distribuídos necessariamente nos levaria para o campo das metodologias de engenharia de *software* para sistemas distribuídos. Isso está fora dos objetivos deste livro; os interessados nesse assunto devem consultar o quadro a seguir, que traz alguns tópicos e fontes relevantes. Em vez disso, optamos por dar uma ideia desta área apresentando o estudo de caso de um sistema distribuído complexo, destacando as decisões de projeto e compromissos envolvidos nesse projeto.

Para motivar os estudos, o Capítulo 1 apresentou, em linhas gerais, três exemplos de importantes domínios de aplicação que representam muitos dos maiores desafios nos sistemas distribuídos: pesquisa na Web, games *online* para múltiplos jogadores e sistemas financeiros. Poderíamos ter escolhido qualquer uma dessas áreas e apresentado um estudo de caso atraente e esclarecedor, mas optamos por nos concentrar na primeira delas: pesquisa na Web (e, na verdade, vamos além da pesquisa na Web, examinando o suporte mais geral para serviços em nuvem baseados na Web). Em particular, apresentamos neste capítulo um estudo de caso sobre a infraestrutura de sistema distribuído que serve de base para o Google (daqui em diante referida como infraestrutura do Google). O Google é um dos maiores sistemas distribuídos em uso atualmente e sua infraestrutura tem lidado com êxito com uma variedade de requisitos exigentes, conforme discutido a seguir. A arquitetura subjacente e a escolha de conceitos também são muito interessantes,

Engenharia de software para sistemas distribuídos

Remetemos o leitor aos avanços significativos feitos em áreas como:

- projeto orientado a objetos, incluindo o uso de notações de modelagem, como UML [Booch *et al.* 2005];
- engenharia de *software* baseada em componentes (CBSE) e sua relação com as arquiteturas empresariais [Szyperski 2002];
- padrões arquitetônicos destinados aos sistemas distribuídos [Bushmann *et al.* 2007];
- engenharia baseada em modelos, que busca gerar sistemas complexos, incluindo sistemas distribuídos, a partir de abstrações de nível mais alto (modelos) [France e Rumpe 2007].

reunindo muitos dos tópicos básicos apresentados neste livro. Portanto, um estudo da infraestrutura do Google oferece uma maneira perfeita de encerrar nosso estudo sobre sistemas distribuídos. Note que, além de dar um exemplo de como suportar pesquisa na Web, a infraestrutura do Google também se revelou um exemplo importante de computação em nuvem, conforme ficará claro nas descrições a seguir.

A Seção 21.2 apresenta o estudo de caso, fornecendo informações básicas sobre o Google. Em seguida, a Seção 21.3 apresenta o projeto global da infraestrutura do Google, considerando o modelo físico subjacente e a arquitetura de sistema associada. A Seção 21.4 examina o nível mais baixo da arquitetura de sistema, os paradigmas de comunicação suportados pela infraestrutura do Google; as duas seções subsequentes, Seções 21.5 e 21.6, discutem os serviços básicos fornecidos pela infraestrutura do Google, apresentando os serviços de armazenamento e processamento de grandes volumes de dados. Juntas, as Seções 21.3 a 21.6 descrevem um *middleware* completo para pesquisa na Web e computação em nuvem. Por fim, a Seção 21.7 resume os principais pontos que surgem de nossa discussão sobre a infraestrutura do Google. Por toda a apresentação, será dada ênfase na identificação e na justificativa das principais decisões de projeto e os compromissos inerentes associados.

21.2 Introdução ao estudo de caso: Google

O Google [www.google.com III] é uma corporação dos Estados Unidos, sediada em Mountain View, Califórnia (a Googleplex), oferecendo pesquisa na Internet e aplicações Web mais gerais, obtendo seus lucros principalmente com os anúncios ligados a tais serviços. O nome é um trocadilho com a palavra *googol*, o número 10^{100} (ou 1 seguido por cem zeros), enfatizando a enorme escala das informações disponíveis na Internet atualmente. A missão do Google é dominar esse enorme volume de informações: “*organizar as informações do mundo e torná-las universalmente acessíveis e úteis*” [www.google.com III].

O Google nasceu de um projeto de pesquisa na Universidade de Stanford, com a empresa sendo inaugurada em 1998. Desde então, ela cresceu até ter uma fatia dominante do mercado da pesquisa na Internet, principalmente devido à eficiência do algoritmo de classificação subjacente utilizado em seu mecanismo de busca (discutido com mais detalhes a seguir). De forma significativa, o Google diversificou e, além de fornecer um mecanismo de busca, agora tem uma importante participação na computação em nuvem.

Da perspectiva dos sistemas distribuídos, o Google oferece um fascinante estudo de caso, com requisitos extremamente exigentes, particularmente em termos de escalabilidade, confiabilidade, desempenho e abertura (veja a discussão sobre esses desafios na Seção 1.5). Por exemplo, em termos de pesquisa, vale notar que o sistema subjacente tem se adaptado muito bem ao crescimento da empresa. Desde seu sistema de produção inicial (em 1998), até lidar com mais de 88 bilhões de pesquisas por mês (em 2010), seu principal mecanismo de busca nunca sofreu uma interrupção em todo esse tempo e seus usuários podem esperar resultados de consulta em torno de 0,2 segundos [googleblog.blogspot.com I]. O estudo de caso apresentado aqui examinará as estratégias e decisões de projeto por trás desse sucesso e dará uma ideia do projeto de sistemas distribuídos complexos.

Antes de passarmos para o estudo de caso, contudo, é instrutivo ver com mais detalhes o mecanismo de busca e também o Google como provedor de nuvem.

O mecanismo de busca do Google • A função do mecanismo de busca do Google, assim como para qualquer mecanismo de pesquisa na Web, é receber determinada consulta e retornar uma lista ordenada com os resultados mais relevantes que satisfazem essa consulta, pesquisando o conteúdo da Web. Os desafios derivam do tamanho da Web e de sua taxa de mudança, assim como do requisito de fornecer os resultados mais relevantes do ponto de vista de seus usuários.

Fornecemos a seguir uma breve visão geral do funcionamento da pesquisa no Google; uma descrição mais completa do funcionamento do mecanismo de busca do Google pode ser encontrado em Langville e Meyer [2006]. Como exemplo, consideraremos como o mecanismo de busca responde à consulta “livro sistemas distribuídos”.

O mecanismo de busca consiste em um conjunto de serviços para esquadrinhar a Web e indexar e classificar as páginas descobertas, conforme discutido a seguir.

Esquadrinhamento (crawling): a tarefa do *crawler* é localizar e recuperar o conteúdo da Web e passá-lo para o subsistema de indexação. Isso é realizado por um serviço de *software* chamado Googlebot, que lê determinada página Web recursivamente, coletando todos os *links* dessa página Web e programando mais operações de esquadrinhamento para os *links* coletados (uma técnica conhecida como *pesquisa em profundidade*, altamente eficiente em buscar praticamente todas as páginas na Web).

No passado, devido ao tamanho da Web, o esquadrinhamento geralmente era feito uma vez a cada poucas semanas. Entretanto, para certas páginas Web isso era insuficiente. Por exemplo, é importante que os mecanismos de busca sejam capazes de informar precisamente sobre notícias de última hora ou mudanças em preços de ações. Portanto, o Googlebot tomava nota do histórico de alterações em páginas Web e revisitava frequentemente as que mudavam, com um período aproximadamente proporcional à frequência das mudanças. Com a introdução do Caffeine, em 2010 [[googleblog.blogspot.com II](http://googleblog.blogspot.com/2010/02/introducing-caffeine.html)], o Google mudou de uma estratégia em lotes para um processo mais contínuo de esquadrinhamento, destinado a oferecer resultados de busca mais atualizados. O Caffeine foi construído usando um novo serviço de infraestrutura, chamado de Percolator, que suporta a atualização incremental de grandes conjuntos de dados [Peng e Dabek 2010].

Indexação: embora o esquadrinhamento seja uma função importante em termos de reconhecer o conteúdo da Web, ele não nos ajuda na busca de ocorrências de “livro sistemas distribuídos”. Para entendermos como isso é processado, precisamos examinar mais de perto a indexação.

A função da indexação é produzir um índice para o conteúdo da Web que seja semelhante a um índice do final de um livro, mas em uma escala muito maior. Mais precisamente, a indexação produz o que conhecido como *índice invertido*, mapeando as palavras que aparecem em páginas Web e outros recursos Web textuais (incluindo documentos nos formatos *.pdf*, *.doc* e outros) nas posições em que ocorrem nos documentos, incluindo a posição exata no documento e outras informações relevantes, como tamanho da fonte e se há uso de letras maiúsculas (que são usadas para determinar a importância, conforme veremos a seguir). O índice também é classificado para suportar consultas eficientes de palavras em relação às localizações.

Além de manter um índice de palavras, o mecanismo de busca do Google também mantém um índice de *links*, monitorando quais páginas estão ligadas a determinado *site*. Isso é usado pelo algoritmo PageRank, conforme discutido a seguir.

Vamos voltar ao nosso exemplo de consulta. Esse índice invertido nos permitirá descobrir páginas Web que incluem os termos de busca “distribuídos”, “sistemas” e “livro” e, por meio de uma análise cuidadosa, poderemos descobrir páginas que incluem todos os termos. Por exemplo, o mecanismo de busca poderá identificar que todos os três termos podem ser

encontrados em amazon.com, www.cdk5.net e em muitos outros *sites*. Assim, usando o índice é possível reduzir o conjunto de páginas Web candidatas de bilhões para talvez dezenas de milhares, dependendo do nível de diferenciação nas palavras-chave escolhidas.

Classificação: o problema da indexação é que ela não fornece nenhuma informação sobre a importância relativa das páginas Web que contêm um conjunto de palavras-chave em particular – apesar de isso ser fundamental para se determinar a relevância em potential de uma página. Portanto, todos os mecanismos de busca modernos dão ênfase significativa a um sistema de classificação por meio do qual uma classificação mais alta seja uma indicação da importância de uma página e seja usada para garantir que a página será retornada mais perto da parte superior da lista de resultados do que as páginas com classificação mais baixa. Conforme mencionado anteriormente, grande parte do sucesso do Google pode ser creditado à eficiência de seu algoritmo de classificação, *PageRank* [Langville e Meyer 2006].

O algoritmo PageRank foi inspirado no sistema de classificação de artigos acadêmicos baseado na análise de citações. No mundo acadêmico, um artigo é considerado importante se tem muitas citações feitas por outros acadêmicos da área. Analogamente, no PageRank, uma página será considerada importante se for ligada a um grande número de outras páginas (usando os dados de *link* mencionados anteriormente). O PageRank também vai além de uma simples análise de “citação”, examinando a importância dos *sites* que contêm *links* para determinada página. Por exemplo, um *link* para *bbc.co.uk* será considerado como mais importante do que um *link* para a página Web pessoal de Gordon Blair.

No Google, a classificação leva diversos outros fatores em conta, incluindo a proximidade de palavras-chave em uma página e se estão em uma fonte grande ou se as letras são maiúsculas (com base nas informações armazenadas no índice invertido).

Voltando ao nosso exemplo, após fazer uma pesquisa de índice em cada uma das três palavras da consulta, a função de busca classifica todas as referências de página resultantes, de acordo com a importância percebida. Por exemplo, a classificação escolheria certas referências de página sob amazon.com e www.cdk5.net por causa do grande número de *links* para essas páginas em outros *sites* “importantes”. A classificação dará prioridade às páginas onde os termos “distribuídos”, “sistemas” e “livro” aparecem bem próximos. Analogamente, a classificação extraíria as páginas em que as palavras aparecem próximas ao início da página ou em letras maiúsculas, talvez indicando uma lista de livros-texto sobre sistemas distribuídos. O resultado final deve ser uma lista classificada de resultados na qual as entradas que estão no início são as mais importantes.

Anatomia de um mecanismo de busca: os fundadores do Google, Sergey Brin e Larry Page, escreveram um artigo de referência sobre a “anatomia” do mecanismo de busca do Google, em 1998 [Brin e Page 1998], fornecendo ideias interessantes sobre como seu mecanismo de busca foi implementado. A arquitetura global descrita nesse artigo está ilustrada na Figura 21.1, extraída do original. Nesse diagrama, fazemos distinção entre os serviços que suportam pesquisa na Web diretamente, desenhados como elipses, e os componentes da infraestrutura de armazenamento subjacente, ilustrados como retângulos.

Embora não seja o objetivo deste capítulo apresentar essa arquitetura em detalhes, uma breve visão geral ajudará na comparação com a infraestrutura do Google mais sofisticada disponível atualmente. A função básica do *esquadrinhamento* já foi descrita anteriormente. Ele recebe como entrada listas de URLs a serem buscados, fornecidos pelo *servidor de URL*, com as páginas buscadas resultantes colocadas no *servidor de armazenamento*. Então, esses dados são compactados e colocados no *repositório* para mais análise, em particular criando o índice para pesquisa. A função de indexação é executada em dois estágios. Primeiramente, o *indexador* descompacta os dados do repositório e produz um conjunto de *acessos (hits)*, no qual um *hit* é representado pelo ID do documento, a

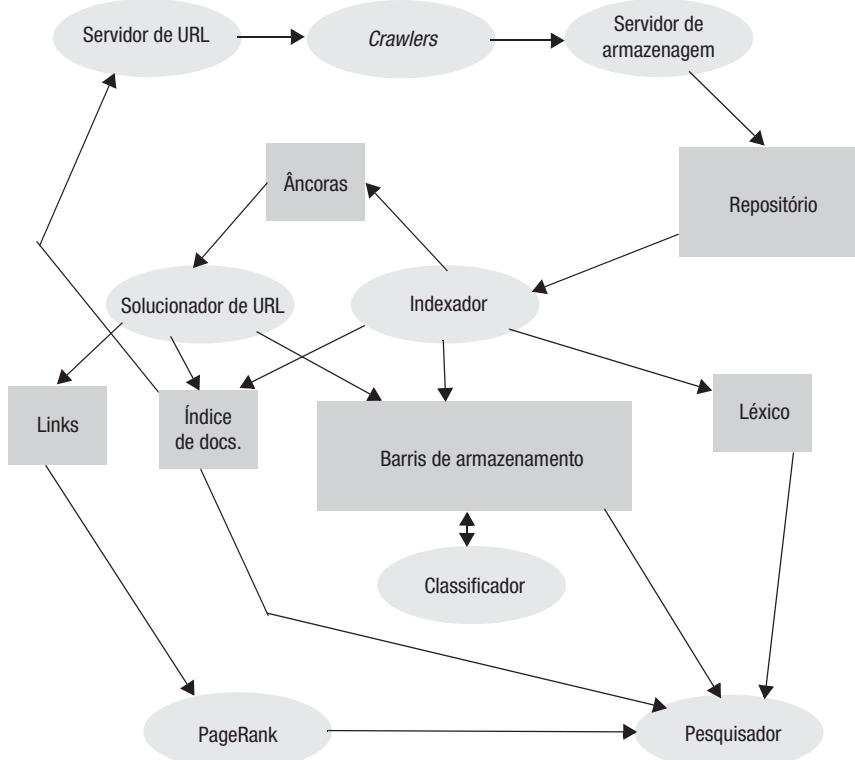


Figura 21.1 Esboço da arquitetura do mecanismo de busca original do Google [Brin e Page 1998].

palavra, a posição no documento e outras informações, como o tamanho da palavra e o uso de maiúsculas. Então, esses dados são armazenados em um conjunto de *barris*, um importante elemento de armazenamento na arquitetura inicial. Essas informações são armazenadas pelo ID do documento. Então, o *classificador* pega esses dados e os classifica por ID de palavra para produzir o índice invertido necessário (conforme discutido anteriormente). O indexador também executa outras duas funções muito importantes ao analisar os dados: ele extrai informações sobre *links* em documentos, armazenando essas informações em um arquivo de *âncoras*, e produz um *léxico* para os dados analisados (o qual, quando a arquitetura inicial foi usada, consistia em 14 milhões de palavras). O arquivo de âncoras é processado por um *solucionador de URL*, o qual executa várias funções sobre esses dados, incluindo transformar URLs relativos em URLs absolutos, antes de produzir um banco de dados de *links*, como uma entrada para os cálculos de *PageRank*. O solucionador de URL também cria um *doc index*, o qual fornece entrada para o servidor de URL em termos de mais páginas a esquadrinhar. Por fim, o *pesquisador* implementa o recurso de pesquisa básico do Google, recebendo entrada do *doc index*, de *PageRank*, do índice invertido mantido nos barris e também do léxico.

Algo extraordinário em relação a essa arquitetura é que, embora seus detalhes específicos tenham mudado, os principais serviços que suportam pesquisa na Web – isto é, esquadrinhamento, indexação (incluindo a ordenação), classificação (através de *PageRank*) – permanecem os mesmos.

<i>Aplicação</i>	<i>Descrição</i>
Gmail	Sistema de correio com mensagens hospedadas pelo Google, mas com gerenciamento de mensagens do tipo área de trabalho.
Google Docs	Conjunto de escritório baseado na web, suportando edição compartilhada de documentos mantidos nos servidores do Google.
Google Sites	<i>Sites</i> no estilo <i>wiki</i> , com recursos de edição compartilhada.
Google Talk	Superta troca de mensagens de texto instantâneas e Voice over IP.
Google Calendar	Calendário baseado na web, com todos os dados hospedados nos servidores do Google.
Google Wave	Ferramenta de colaboração que integra e-mail, troca de mensagens instantânea, <i>wikis</i> e redes sociais.
Google News	<i>Site</i> agregador de notícias totalmente automatizado.
Google Maps	Mapa-múndi baseado na Web, incluindo imagens de alta resolução e sobreposições ilimitadas, geradas pelos usuários.
Google Earth	Visão do globo terrestre quase tridimensional e escalonável, com sobreposições ilimitadas, geradas pelos usuários.
Google App Engine	Infraestrutura distribuída do Google, disponibilizada para terceiros como um serviço (plataforma como serviço).

Figura 21.2 Exemplo de aplicações do Google.

Igualmente notável é que, conforme ficará claro a seguir, a infraestrutura mudou radicalmente, desde as primeiras tentativas de identificar uma arquitetura para pesquisa na Web até o sofisticado suporte para sistemas distribuídos fornecido atualmente, tanto em termos de identificar blocos de construção mais reutilizáveis para comunicação, armazenamento e processamento, como em termos de generalização da arquitetura para além da pesquisa.

Google como provedor de nuvem • O Google diversificou significativamente e, agora, além de pesquisa oferece uma ampla variedade de aplicações baseadas na Web, incluindo o conjunto promovido como Google Apps [www.google.com I]. Mais geralmente, o Google agora é um forte participante na área da computação em nuvem. Lembre-se de que a computação em nuvem foi apresentada no Capítulo 1 e definida como “um conjunto de serviços de aplicativos, armazenamento e computação baseados na Internet, suficientes para suportar as necessidades da maioria dos usuários, permitindo, assim, que eles prescindam em grande medida ou totalmente do *software* local de armazenamento de dados ou de aplicativo”. É exatamente isso que agora o Google se esforça em oferecer, em particular com produtos significativos nas áreas de *software como serviço* e *plataforma como serviço* (conforme apresentado na Seção 7.7.1). Examinaremos cada uma dessas áreas por sua vez, a seguir.

Software como serviço: essa área ocupa-se em oferecer *software* em nível de aplicação pela Internet, como aplicativos Web. Um exemplo importante é o Google Apps, um conjunto de aplicações baseadas na Web que incluem Gmail, Google Docs, Google *Sites*, Google Talk e Google Calendar. O objetivo do Google é substituir os tradicionais conjuntos para escritório por aplicações que suportam preparação compartilhada de documentos, calendários *online* e diversas ferramentas de colaboração que suportam *e-mail*, *wikis*, *Voice over IP* e troca instantânea de mensagens.

Recentemente, foram desenvolvidas várias outras aplicações inovadoras baseadas na Web; elas e o Google Apps original estão resumidos na Figura 21.2. Uma das princi-

pais observações para os propósitos deste capítulo é que o Google estimula uma estratégia aberta para a inovação dentro da organização e, assim, novas aplicações estão surgindo do tempo todo. Isso impõe uma demanda específica sobre a infraestrutura de sistemas distribuídos subjacente, um ponto que será revisitado na Seção 21.3.2.

Plataforma como serviço: essa área ocupa-se em oferecer APIs de sistema distribuído como serviços pela Internet, sendo essas APIs usadas para suportar o desenvolvimento e a hospedagem de aplicações Web (note que, infelizmente, o uso do termo “plataforma” nesse contexto é incoerente com a maneira como é utilizado em outras partes deste livro, em que se refere ao *hardware* e ao sistema operacional). Com o lançamento do Google App Engine, o Google foi além do *software como serviço* e agora oferece sua infraestrutura de sistemas distribuídos, conforme discutido por todo este capítulo como serviço de nuvem. Mais especificamente, já está previsto que os negócios do Google utilizem essa infraestrutura de nuvem internamente para dar suporte a todas as suas aplicações e serviços, incluindo seu mecanismo de pesquisa na Web. Agora, o Google App Engine oferece acesso externo como parte dessa infraestrutura, permitindo que outras organizações executem suas próprias aplicações Web na plataforma Google.

Veremos mais detalhes da infraestrutura do Google no decorrer deste capítulo; para mais detalhes sobre o Google App Engine, consulte o [site do Google \[code.google.com IV\]](http://site do Google [code.google.com IV]).

21.3 Arquitetura global e filosofia de projeto

Esta seção considera a arquitetura global do sistema Google, examinando:

- a arquitetura física adotada pelo Google;
- a arquitetura de sistema associada que oferece serviços comuns para o mecanismo de busca na Internet e as muitas aplicações Web oferecidas pelo Google.

21.3.1 Modelo físico

A principal filosofia do Google em termos de infraestrutura física é o uso de números muito grandes de PCs comuns para produzir um ambiente de custo reduzido para armazenamento e computação distribuídos. As decisões de compra são baseadas na obtenção do melhor desempenho por dólar, em vez do desempenho absoluto, com um gasto típico em um único PC girando em torno de US\$1.000. Um PC normalmente terá cerca de 2 terabytes de armazenamento em disco, em torno de 16 gigabytes de memória DRAM (Dynamic Random Access Memory – memória de acesso aleatório dinâmico) e executará uma versão reduzida do núcleo Linux. (Essa filosofia de construção de sistemas a partir de PCs comuns reflete os primórdios do projeto de pesquisa original, quando Sergey Brin e Larry Page construíram o primeiro mecanismo de busca do Google a partir de *hardware* de refugo, recolhido próximo ao laboratório da Universidade de Stanford.)

Optando por tomar o caminho dos PCs comuns, o Google reconhecia que partes de sua infraestrutura falharia e, assim, conforme veremos a seguir, projetou a infraestrutura usando diversas estratégias para tolerar tais falhas. Hennessy e Patterson [2006] relatam as seguintes características de falha do Google:

- De longe, a fonte de falha mais comum é em razão do *software*, com cerca de 20 máquinas precisando ser reinicializadas por dia. (É interessante notar que o processo de reinicialização é totalmente manual.)

- As falhas de *hardware* representam cerca de 1/10 das falhas em razão do *software*, com cerca de 2 a 3% dos PCs falhando anualmente, devido ao *hardware*. Desses, 95% se devem a falhas nos discos ou na memória DRAM.

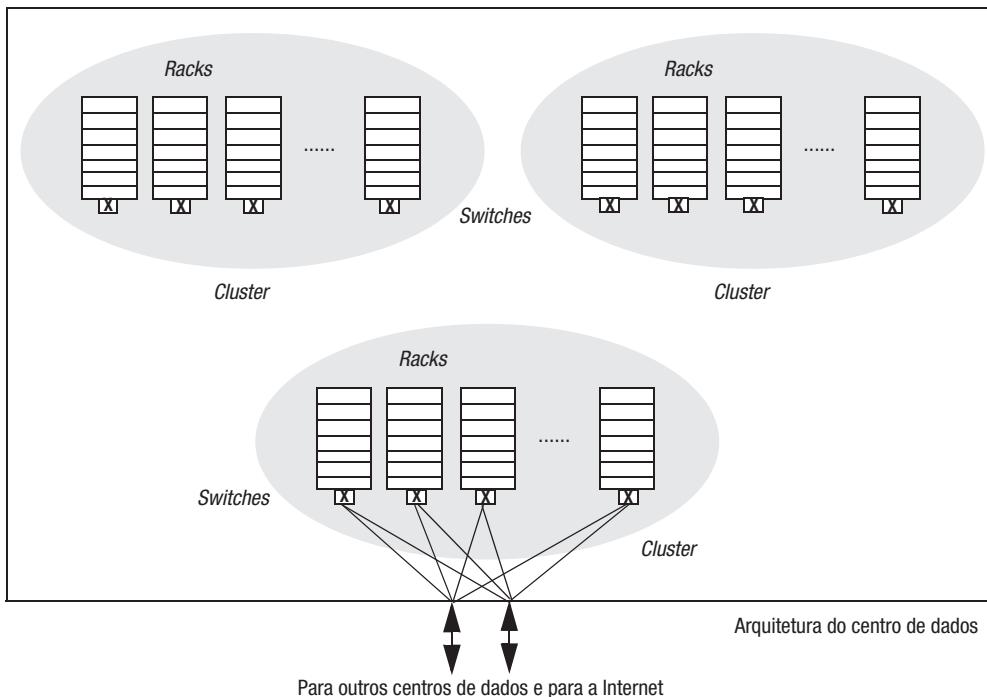
Isso justifica a decisão de adquirir PCs comuns; dado que a ampla maioria das falhas é em razão do *software*, não vale a pena investir em *hardware* mais caro e confiável. Outro artigo, de Pinheiro *et al.* [2007], também relata as características de falha dos discos comuns utilizados na infraestrutura física do Google, dando uma ideia interessante sobre os padrões de falha de armazenamento em disco em implantações de larga escala.

A arquitetura física é construída como segue [Hennessy e Patterson 2006]:

- Os PCs são organizados em *racks* contendo entre 40 e 80 PCs cada um. Os *racks* têm dois lados, com metade dos PCs em cada um. Cada *rack* tem um *switch* Ethernet que fornece conectividade para ele e também para o mundo externo (veja a seguir). Esse *switch* é modular, organizado como um número de lâminas (*blades*), com cada lâmina suportando 8 interfaces de rede de 100 Mbps ou uma interface de 1 Gbps. Para 40 PCs, 5 lâminas contendo cada uma oito interfaces de rede são suficientes para garantir a conectividade dentro da prateleira. Mais duas lâminas, cada uma suportando uma interface de rede de 1 Gbps, são usadas para conexão com o mundo externo.
- Os *racks* são organizados em *clusters* (conforme discutido na Seção 1.3.4), que são uma unidade de gerenciamento importante, determinando, por exemplo, a localização e a replicação de serviços. Normalmente, um *cluster* consiste em 30 ou mais *racks* e dois *switches* de alta largura de banda, fornecendo conectividade para o mundo externo (a Internet e outros centros do Google). Por redundância, cada *rack* é conectado aos dois *switches*; além disso, para ter ainda mais redundância, cada *switch* tem enlaces redundantes para o mundo externo.
- Os *clusters* ficam em centros de dados do Google espalhados pelo mundo. Em 2000, o Google contava com importantes centros de dados no Vale do Silício (dois centros) e na Virgínia, nos Estados Unidos. Quando este livro estava sendo produzido, o número de centros de dados tinha aumentado significativamente e agora existem centros em muitos lugares nos Estados Unidos, além de em Dublin (Irlanda), Saint-Ghislain (Bélgica), Zurique (Suiça), Tóquio (Japão) e Pequim (China). Um mapa dos centros de dados conhecidos em 2008 pode ser encontrado em [[royal.pingdom.com](#)].

Uma visão simplificada dessa organização global aparece na Figura 21.3. Essa infraestrutura física proporciona ao Google recursos de armazenamento e computacionais enormes, junto à redundância necessária para a construção de sistemas de larga escala e tolerantes a falhas (note que, para evitar confusão, essa figura mostra apenas as conexões Ethernet de um dos *clusters* para os enlaces externos).

Capacidade de armazenamento: consideremos a capacidade de armazenamento disponível no Google. Se cada PC oferece 2 terabytes de armazenamento, então um *rack* com 80 PCs fornecerá 160 terabytes, com um *cluster* de 30 *racks* oferecendo 4,8 petabytes. Não se sabe exatamente quantas máquinas o Google tem no total, pois a empresa mantém estrito sigilo a respeito desse aspecto de seu negócio, mas podemos supor que tenha perto de 200 *clusters*, oferecendo uma capacidade de armazenamento total de 960 petabytes ou quase 1 exabyte de armazenamento (10^{18} bytes). É provável que esse valor seja conservador, pois a vice-presidente do Google, Marissa Mayer, já fala sobre a explosão de dados à faixa da exaescala [[www.parc.com](#)].



(Para simplificar o diagrama, é mostrada a conexão Ethernet de um dos *clusters* com a Internet.)

Figura 21.3 Organização da infraestrutura física do Google.

No restante deste capítulo, veremos como o Google utiliza esse recurso de armazenamento e computacional extensivo e a redundância associada para oferecer serviços básicos.

21.3.2 Arquitetura de sistema global

Antes de examinarmos a arquitetura de sistema global, é interessante verificar os principais requisitos com mais detalhes:

Escalabilidade: o primeiro e mais óbvio requisito para a infraestrutura do Google é controlar a escalabilidade e, em particular, ter estratégias que se adaptem ao que é um sistema distribuído ULS (Ultra-Large Scale), conforme apresentado no Capítulo 2. Para o mecanismo de busca, o Google vê o problema da escalabilidade em termos de três dimensões: i) ser capaz de lidar com mais dados (por exemplo, à medida que o volume de informações na Web aumenta por meio de iniciativas como a digitalização de bibliotecas), ii) ser capaz de tratar de mais consultas (à medida que aumenta o número de pessoas que usam o Google em suas casas e locais de trabalho) e iii) procurar resultados melhores (particularmente importante, pois esse é um fator determinante no funcionamento de um mecanismo de pesquisa na Web). Essa visão do problema da escalabilidade está ilustrada na Figura 21.4.

A escalabilidade exige o uso de estratégias (sofisticadas) de sistemas distribuídos. Vamos ilustrar isso com uma análise simples extraída da ideia básica de

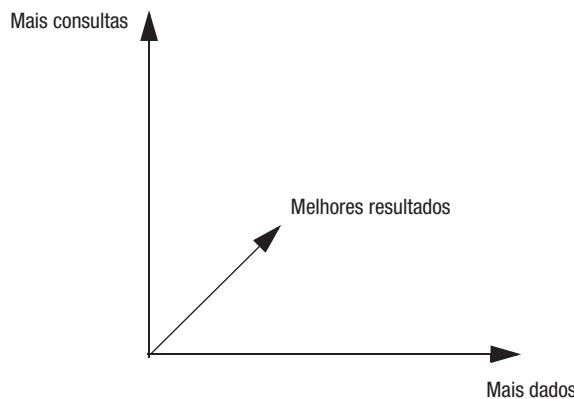


Figura 21.4 O problema da escalabilidade no Google.

Jeff Dean no PACT’06 [Dean 2006]. Ele presumiu que a Web consiste em cerca de 20 bilhões de páginas com 20 quilobytes cada uma. Isso significa um tamanho total de cerca de 400 terabytes. Examinar esse volume de dados exigiria mais de 4 meses para um único computador, supondo que esse computador possa ler 30 megabytes por segundo. Em contraste, 1.000 máquinas podem ler esse volume de dados em menos de 3 horas. Além disso, conforme vimos na Seção 21.2, pesquisar não envolve apenas esquadrinhar. Todas as outras funções, incluindo indexar, classificar e pesquisar, exigem soluções altamente distribuídas para que seja possível fazer a adaptação.

Confiabilidade: o Google tem requisitos de confiabilidade rigorosos, especialmente com relação à disponibilidade de serviços. Isso é particularmente importante para a funcionalidade de pesquisa, em que há necessidade de oferecer disponibilidade 24 horas por dia, sete dias por semana (notando, contudo, que é intrinsecamente fácil mascarar falhas em pesquisas, pois o usuário não tem como saber se todos os resultados foram retornados). Esse requisito também serve para outras aplicações Web e é interessante notar que o Google oferece um acordo de nível de serviço de 99,9% (efetivamente uma garantia de sistema) para clientes que pagam pelo Google Apps, cobrindo o Gmail, o Google Calendar, o Google Docs, o Google Sites e o Google Talk. A empresa tem um excelente recorde global em termos de disponibilidade de serviços, mas a conhecida interrupção do Gmail em 1º de setembro de 2009 serve como um lembrete dos contínuos desafios nessa área. (Essa interrupção, que durou 100 minutos, foi causada por um problema em cascata de servidores sobrecarregados durante um período de manutenção de rotina). Note que o requisito da confiabilidade deve ser satisfeito no contexto das escolhas de projeto na arquitetura física, o que significa que as falhas (de *software* e *hardware*) são antecipadas com frequência razoável. Isso exige detecção de falhas e a adoção de estratégias para mascarar ou tolerar tais falhas. Essas estratégias contam pesadamente com a redundância na arquitetura física subjacente. Veremos exemplos dessas estratégias à medida que os detalhes da arquitetura de sistema surgirem.

Desempenho: o desempenho global do sistema é fundamental para o Google, especialmente na obtenção de baixa latência de interações do usuário. Quanto melhor



Figura 21.5 A arquitetura global de sistemas do Google.

o desempenho, mais provável será que um usuário retorne com mais consultas que, por sua vez, aumentam a exposição de anúncios, potencialmente aumentando os lucros. A importância do desempenho é exemplificada, por exemplo, com a meta de concluir operações de pesquisa na Web em 0,2 segundos (conforme mencionado anteriormente) e obter o desempenho de saída exigido para responder a todas as requisições recebidas, enquanto lida com conjuntos de dados muito grandes. Isso se aplica a uma ampla variedade de funções associadas à operação do Google, incluindo esquadrinhamento da Web, indexação e classificação. Também é importante notar que o desempenho é uma propriedade fim-a-fim, exigindo que todos os recursos associados funcionem juntos, incluindo recursos de rede, armazenamento e computacionais.

Abertura: de muitas maneiras, os requisitos anteriores são óbvios para o Google suportar seus serviços e aplicações básicos. Há também um forte requisito de abertura, particularmente para suportar mais desenvolvimento na variedade de aplicações Web oferecidas. Sabe-se que o Google, como organização, estimula e cultiva a inovação, e isso fica mais evidente no desenvolvimento de novas aplicações Web. Isso só é possível com uma infraestrutura extensível e que dê suporte para o desenvolvimento de novas aplicações.

O Google tem respondido a essas necessidades desenvolvendo a arquitetura de sistema global mostrada na Figura 21.5. Essa figura mostra a plataforma de computação na parte inferior (isto é, a arquitetura física descrita anteriormente) e os conhecidos serviços e aplicações do Google na parte superior. A camada do meio define uma infraestrutura distribuída comum que fornece suporte de *middleware* para pesquisa e computação em nuvem. Isso é fundamental para o sucesso do Google. A infraestrutura fornece os serviços de sistema distribuído comuns para os desenvolvedores de serviços e aplicações Google e encapsula as principais estratégias para lidar com escalabilidade, confiabilidade e desempenho. O fornecimento de uma infraestrutura comum bem projetada como essa pode promover o desenvolvimento de novas aplicações e serviços por meio da reutilização dos serviços de sistema subjacentes e, mais sutilmente, fornece uma coerência global para o crescimento da base de código do Google, impondo estratégias e princípios de projeto comuns.

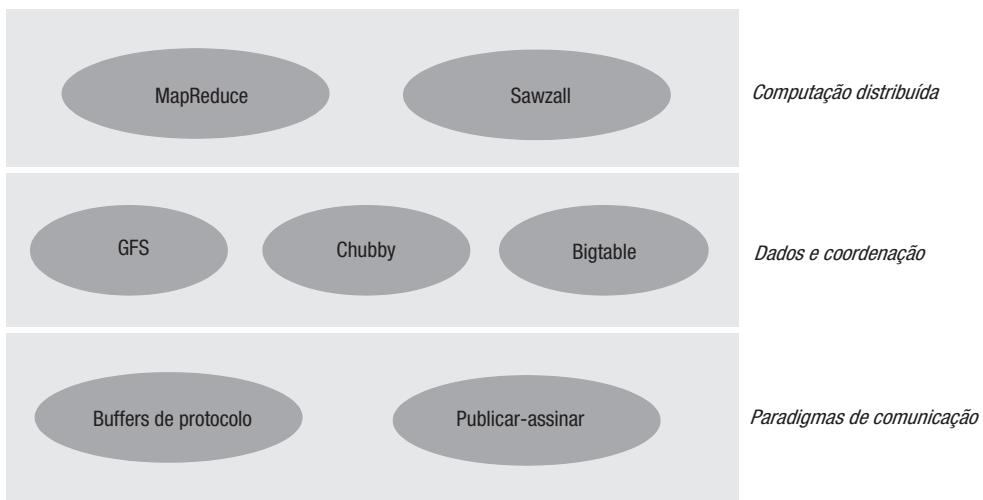


Figura 21.6 Infraestrutura do Google.

Infraestrutura do Google • O sistema é construído como um conjunto de serviços distribuídos que oferecem funcionalidade básica para os desenvolvedores (veja a Figura 21.6). Esse conjunto de serviços é naturalmente dividido nos seguintes subconjuntos:

- os paradigmas de comunicação subjacentes, incluindo serviços para invocação remota e comunicação indireta:
 - o componente *buffers de protocolo* oferece um formato de serialização comum para o Google, incluindo a serialização de requisições e respostas na invocação remota.
 - o serviço de *publicar-assinar* do Google suporta a disseminação eficiente de eventos para números potencialmente grandes de assinantes.
- serviços de dados e coordenação, fornecendo abstrações não estruturadas e semiestruturadas para o armazenamento de dados acoplado a serviços para suportar o acesso coordenado aos dados:
 - O GFS oferece um sistema de arquivos distribuído otimizado para os requisitos específicos de aplicações e serviços do Google (incluindo o armazenamento de arquivos muito grandes).
 - O Chubby suporta serviços de coordenação e a capacidade de armazenar volumes de dados menores.
 - A Bigtable fornece um banco de dados distribuído que dá acesso a dados semiestruturados.
- serviços de computação distribuída, que oferecem meios para executar computação paralela e distribuída na infraestrutura física:
 - O MapReduce suporta computação distribuída em conjuntos de dados potencialmente muito grandes (por exemplo, armazenados na Bigtable).

- A Sawzall é uma linguagem de nível mais alto para a execução de tais computações distribuídas.

Examinaremos cada um desses componentes por vez, nas Seções 21.4 a 21.6. Primeiro, contudo, é instrutivo refletir sobre os principais princípios de projeto associados à arquitetura como um todo.

Princípios de projeto • Para se entender completamente o projeto da infraestrutura do Google, é importante entender também as principais filosofias de projeto que permeiam a organização:

- O princípio de projeto mais importante por trás do *software* Google é a *simplicidade*: o *software* deve fazer apenas uma coisa e fazê-la bem, evitando projetos cheios de recursos, quando possível. Por exemplo, Bloch [2006] discute como esse princípio se aplica no projeto da API, implicando que a API deva ser a menor possível (um exemplo da aplicação de Razor de Occam).
- Outro importante princípio de projeto é uma forte ênfase no *desempenho* no desenvolvimento de *software* de sistemas, capturada na frase “cada milissegundo conta” [www.google.com/IV]. Em uma exposição de princípios básicos na LADIS’09, Jeff Dean (membro do *Google Systems Infrastructure Group*) enfatizou a importância de ser possível estimar o desempenho de um projeto de sistema conhecendo-se os custos do desempenho de operações primitivas, como acesso à memória e ao disco, envio de pacotes por uma rede, travamento e destravamento de um mutex, etc., acoplado ao que ele se referiu como cálculos no “verso do envelope” [www.cs.cornell.edu].
- Um último princípio é defender regimes de *teste* rigorosos no *software*, capturado pelo slogan “se não quebrou, você não se esforçou o bastante” [googletesting.blogspot.com]. Isso é complementado por uma forte ênfase no *registro* e *rastreamento* para detectar e resolver falhas no sistema.

Com essas informações, estamos prontos para examinar as várias partes constituintes da infraestrutura do Google, começando com os paradigmas de comunicação subjacentes. Para cada área, apresentaremos o projeto global e destacaremos as principais decisões de projeto e compromissos associados.

21.4 Paradigmas de comunicação

Recordando os Capítulos 3 a 6, fica claro que a escolha do paradigma (ou paradigmas) de comunicação subjacente é fundamental para o sucesso de um projeto de sistema global. As opções incluem:

- usar diretamente um serviço de comunicação entre processos subjacente, como o oferecido pelas abstrações de soquete (descritas no Capítulo 4 e suportadas por todos os sistemas operacionais modernos);
- usar um serviço de invocação remota (como um protocolo de requisição-resposta, chamadas de procedimento remoto ou invocação a método remoto, conforme discutido no Capítulo 5), oferecendo suporte para interações cliente-servidor;
- usar um paradigma de comunicação indireta, como comunicação em grupo, estratégias baseadas em eventos distribuídos, espaços de tupla ou estratégias de memória compartilhada distribuída (conforme discutido no Capítulo 6).

De conformidade com os princípios de projeto identificados na Seção 21.3, o Google adota um serviço de invocação remota simples, mínimo e eficiente que é uma variante de uma estratégia de chamada de procedimento remoto.

Os leitores se lembrarão de que a comunicação por chamada de procedimento remoto exige um componente de serialização para converter os dados da invocação de procedimento (nome do procedimento e parâmetros, possivelmente estruturados) de sua representação binária interna para um formato *plano* ou *serializado* neutro quanto ao processador, pronto para transmissão ao parceiro remoto. A serialização para RPC Java foi descrita na Seção 4.3.2. XML surgiu mais recentemente como um formato de dados serializados “universal”, mas sua generalidade apresenta sobrecargas substantiais. Portanto, o Google desenvolveu um componente de serialização simplificado e de alto desempenho, conhecido como *buffers de protocolo*, que é usado pela maioria significativa das interações dentro da infraestrutura. Isso pode ser usado em qualquer mecanismo de comunicação subjacente para fornecer um recurso de RPC. Existe uma versão de código-fonte aberto do componente *buffers de protocolo* [code.google.com II].

Um *serviço de publicar-assinar* separado também é usado, reconhecendo a importante função que esse paradigma pode oferecer em muitas áreas do projeto de sistema distribuído, incluindo a disseminação de eventos eficiente e em tempo real para vários participantes. Em comum com muitas outras plataformas de sistema distribuído, a infraestrutura do Google oferece uma solução mista, permitindo aos desenvolvedores selecionar o melhor paradigma de comunicação para seus requisitos. O protocolo de publicação-assinatura não é uma alternativa ao componente *buffers de protocolo* na infraestrutura do Google, mas um acréscimo, oferecendo um serviço de valor agregado onde for mais apropriado.

Examinaremos o projeto dessas duas estratégias, a seguir, com ênfase no componente *buffers de protocolo* (detalhes completos sobre o protocolo de publicar-assinar ainda não estão publicamente disponíveis).

21.4.1 Invocação remota

O componente *buffers de protocolo* enfatiza a descrição e a subsequente serialização dos dados e, assim, o conceito é mais bem comparado com as alternativas diretas, como XML. O objetivo é fornecer uma maneira, neutra quanto à linguagem e quanto à plataforma, de especificar e serializar dados de modo simples, altamente eficiente e extensível; os dados serializados podem, então, ser usados para armazenamento subsequente ou para transmissão, usando um protocolo de comunicação subjacente, ou mesmo para qualquer outro propósito que exija um formato de serialização para dados estruturados. Veremos, posteriormente, como isso pode ser utilizado como base para troca estilo RPC.

No componente *buffers de protocolo*, é fornecida uma linguagem para a especificação de *mensagens*. Apresentamos os principais recursos dessa linguagem (simples) por meio de um exemplo, com a Figura 21.7 mostrando como uma mensagem *Book* poderia ser especificada.

Como pode ser visto, a mensagem *Book* consiste em uma série de campos numerados exclusivamente, cada um representado por um nome e pelo tipo do valor associado. O tipo pode ser:

- um *tipo de dados primitivo* (incluindo inteiro, ponto flutuante, booleano, *string* ou bytes brutos);
- um *tipo enumerado*;
- uma *mensagem aninhada*, permitindo uma estruturação de dados hierárquica.

```
message Book {
    required string title = 1;
    repeated string author = 2;
    enum Status {
        IN_PRESS = 0;
        PUBLISHED = 1;
        OUT_OF_PRINT = 2;
    }
    message BookStats {
        required int32 sales = 1;
        optional int32 citations = 2;
        optional Status bookstatus = 3 [default = PUBLISHED];
    }
    optional BookStats statistics = 3;
    repeated string keyword = 4;
}
```

Figura 21.7 Exemplo de *buffers* de protocolo.

Podemos ver exemplos de cada um na Figura 21.7.

Os campos são anotados com um de três rótulos:

- os campos *required* devem estar presentes na mensagem;
- os campos *optional* podem estar presentes na mensagem;
- os campos *repeated* podem existir zero ou mais vezes na mensagem (os desenvolvedores do componente *buffers* de protocolo veem isso como um tipo de vetor dimensionado dinamicamente).

Novamente, podemos ver o uso de cada uma dessas anotações no formato da mensagem *Book* ilustrados na Figura 21.7.

A numeração exclusiva (=1, =2, etc.) representa a marca (*tag*) que um campo em particular tem na codificação binária da mensagem.

Essa especificação está contida em um arquivo *.proto* e é compilada pela ferramenta *protoc*. A saída dessa ferramenta é código gerado que permite aos programadores manipular o tipo de mensagem específico, em particular atribuir/extrair valores a/de mensagens. Em mais detalhes, a ferramenta *protoc* gera uma classe *construtor* que fornece métodos *getter* e *setter* para cada campo, junto a métodos adicionais para *testar* se um método foi ativado e para *limpar* um campo com o valor nulo associado. Por exemplo, os seguintes métodos seriam gerados para o campo *title*:

```
public boolean hasTitle();
public java.lang.String getTitle();
public Builder setTitle(String value);
public Builder clearTitle();
```

A importância da classe construtor é que, embora as mensagens sejam imutáveis no componente *buffers* de protocolo, os construtores são mutáveis e usados para construir e manipular novas mensagens.

Para campos *repeated*, o código gerado é um pouco mais complicado, com métodos fornecidos para retornar uma *contagem* do número de elementos na lista associada, para *obter* ou *configurar* campos específicos na lista, para *anexar* um novo elemento

em uma lista e para adicionar um conjunto de elementos na lista (o método *addAll*). Ilustramos isso por meio de um exemplo, listando os métodos gerados para o campo *keyword*:

```
public List<string> getKeywordList();
public int getKeywordCount();
public string getKeyword(int index);
public Builder setKeyword(int index, string value);
public Builder addKeyword(string value);
public Builder addAllKeyword(Iterable<string> value);
public Builder clearKeyword();
```

O código gerado também fornece diversos outros métodos para manipular mensagens, incluindo métodos como *toString* para fornecer uma representação legível da mensagem (frequentemente usada para depuração, por exemplo) e também uma série de métodos para analisar as mensagens recebidas.

Como se pode ver, esse é um formato muito simples, comparado com XML (por exemplo, compare a especificação anterior com as especificações equivalentes em XML, mostradas na Seção 4.3.3), e que seus desenvolvedores dizem ser de 3 a 10 vezes menor do que os equivalentes em XML e de 10 a 100 vezes mais rápido na operação. A interface de programação associada que dá acesso aos dados também é consideravelmente mais simples do que as equivalentes para XML.

Note que essa é uma comparação um tanto injusta, por dois motivos. Primeiramente, a infraestrutura do Google é um sistema relativamente fechado e, assim, ao contrário de XML, não trata da operação conjunta de sistemas abertos. Em segundo lugar, XML é significativamente mais rica, pois gera mensagens autodescritivas que contêm os dados e os metadados associados descrevendo a estrutura das mensagens (veja a Seção 4.3.3). O componente *buffers* de protocolo não fornece essa facilidade diretamente (embora seja possível obter esse efeito, conforme descrito nas páginas Web relevantes, em uma seção sobre técnicas [[code.google.com II](#)]). Em linhas gerais, isso é obtido pedindo-se para que o compilador *protoc* gere um *FileDescriptorSet* contendo autodescrições das mensagens e, então, incluindo-se isso explicitamente nas descrições da mensagem. Contudo, os desenvolvedores do componente *buffers* de protocolo enfatizam que isso não é visto como um recurso particularmente útil e que raramente é usado no código da infraestrutura do Google.

Suporte para RPC • Conforme mencionado anteriormente, o componente *buffers* de protocolo é um mecanismo geral que pode ser usado para armazenamento ou comunicação. Contudo, o uso mais comum de *buffers* de protocolo é na especificação de trocas RPC na rede, e isso é realizado com uma sintaxe extra na linguagem. Novamente, ilustraremos a sintaxe por meio de um exemplo:

```
service SearchService {
    rpc Search (RequestType) returns (ResponseType);
}
```

Esse trecho de código especifica uma interface de serviço chamada *SearchService* contendo uma operação remota, *Search*, que recebe um parâmetro de tipo *RequestType* e retorna um parâmetro de tipo *ResponseType*. Por exemplo, os tipos poderiam corresponder a uma lista de palavras-chave e uma lista de livros (*Books*) correspondendo a esse con-

junto de palavras-chave respectivamente. O compilador *protoc* pega essa especificação e produz uma interface abstrata *SearchService* e um *stub* que suporta chamadas no estilo RPC seguras quanto ao tipo para o serviço remoto, usando *buffers* de protocolo.

Além de ser neutro quanto à linguagem e quanto à plataforma, o componente *buffers* de protocolo também é agnóstico com relação ao protocolo RPC subjacente. Em particular, o *stub* presume que existem implementações para duas interfaces abstratas *RpcChannel* e *RpcController*, a primeira oferecendo uma interface comum para as implementações RPC subjacentes e a última oferecendo uma interface de controle comum, por exemplo, para manipular as configurações associadas a essa implementação. O programador deve fornecer implementações dessas interfaces abstratas, efetivamente selecionando a implementação de RPC desejada. Por exemplo, isso poderia passar mensagens serializadas usando HTTP ou TCP, ou poderia mapear em uma de várias implementações de RPC de terceiros disponíveis e vinculadas do local do componente *buffers* de protocolo [code.google.com III].

Note que uma interface de serviço pode suportar várias operações remotas, mas cada operação deve obedecer ao padrão de pegar um único parâmetro e retornar apenas um resultado (sendo ambas mensagens de *buffer* de protocolo). Isso é incomum, comparado aos projetos de sistemas RPC e RMI – como vimos no Capítulo 5 –, pois as invocações remotas podem ter um número arbitrário de parâmetros e, no caso da RMI, os parâmetros ou resultados podem ser objetos ou mesmo referências de objeto (embora deva ser notado que a RPC da Sun, conforme documentado na Seção 5.3.3, adota uma estratégia semelhante ao componente *buffers* de protocolo). O fundamento lógico para se ter uma requisição e uma resposta é para dar suporte à capacidade de extensão e à evolução do *software*; embora os estilos mais gerais de interface possam mudar significativamente com o passar do tempo, há uma maior probabilidade de que este estilo mais restrito de interface permaneça constante. Essa estratégia também joga a complexidade para os dados, de uma maneira que lembra a filosofia REST, com seu conjunto de operações restrito e sua ênfase nos recursos de manipulação (veja a Seção 9.2).

21.4.2 Publicar-assinar

O componente *buffers* de protocolo é usado extensivamente, mas não exclusivamente, como paradigma de comunicação na infraestrutura do Google. Para complementar o componente *buffers* de protocolo, a infraestrutura também suporta um sistema de publicação-assinatura destinado a ser usado quando eventos distribuídos precisam ser disseminados em tempo real e com garantias de confiabilidade para números potencialmente grandes de destinatários. Conforme mencionado anteriormente, o serviço de publicação-assinatura é acréscimo ao componente *buffers* de protocolo e, na verdade, utiliza esse componente para sua comunicação subjacente.

Um uso importante do sistema de publicar-assinar, por exemplo, é servir de base para o sistema Google Ads, reconhecendo que, no Google, os anúncios têm abrangência mundial e que sistemas anunciantes de qualquer lugar na rede precisam saber, em uma fração de segundo, a elegibilidade de certas propagandas que podem ser mostradas em resposta a uma consulta.

O sistema RPC descrito anteriormente seria claramente inadequado e altamente ineficiente para esse estilo de interação, especialmente por causa dos números potencialmente grandes de assinantes e das garantias exigidas pelas aplicações associadas.

Em particular, o remetente precisaria conhecer a identidade de todos os outros sistemas anunciantes, o que poderia ser muito grande. As RPCs precisariam ser enviadas para todos os sistemas individuais, consumindo muitas conexões e um enorme espaço de *buffer* associado no remetente, sem mencionar os requisitos de largura de banda para enviar os dados pelos enlaces de rede de longo alcance. Em contraste, uma solução publicar-assinar, com seu inerente desacoplamento temporal e espacial, supera essas dificuldades e também oferece suporte natural para falha e recuperação de assinantes (veja a Seção 6.1).

O Google não tornou disponível publicamente os detalhes do sistema de publicar-assinar. Portanto, restringiremos nossa discussão a alguns recursos de alto nível do sistema.

O Google adota um sistema de publicar-assinar *baseado em tópicos*, fornecendo vários *canais* para fluxos de evento, com os canais correspondendo a tópicos em particular. Foi escolhido um sistema baseado em tópicos por sua facilidade de implementação e sua relativa previsibilidade em termos de desempenho, em comparação com as estratégias baseadas em conteúdo – isto é, a infraestrutura pode ser configurada e personalizada para distribuir eventos relacionados a determinado tópico. O inconveniente é a falta de poder expressivo na especificação de eventos de interesse. Como um compromisso, o sistema de publicar-assinar do Google permite assinaturas aprimoradas, definidas não somente pela seleção de um canal, mas também pela seleção de subconjuntos de eventos dentro desse canal. Em particular, um evento consiste em um cabeçalho, um conjunto de palavras-chave associadas e uma carga útil, que é opaca para o programador. As requisições de assinatura especificam o canal, junto a um filtro definido sobre o conjunto de palavras-chave. Os canais se destinam a ser usados para fluxos de dados relativamente estáticos e brutos, exigindo alto desempenho de saída de eventos (pelo menos 1 Mbps), de modo que a capacidade adicional de expressar assinaturas refinadas usando filtros ajuda muito. Por exemplo, se um tópico gerar menos do que esse volume de dados, eles serão incluídos dentro de outro tópico, mas poderão ser identificados pela palavra-chave.

O sistema de publicar-assinar é implementado como uma sobreposição de *despachantes* na forma de um conjunto de árvores, onde cada árvore representa um tópico. A raiz da árvore é o publicador e os nós-folha representam os assinantes. Quando filtros são introduzidos, eles são colocados o mais abaixo na árvore possível para minimizar tráfego desnecessário.

Ao contrário dos sistemas de publicar-assinar discutidos no Capítulo 6, há uma forte ênfase na entrega confiável e oportunista:

- Em termos de confiabilidade, o sistema mantém árvores redundantes; em particular, são mantidas duas sobreposições de árvore separados por canal (tópico) lógico.
- Em termos de entrega dentro de prazos, o sistema implementa uma técnica de gerenciamento de qualidade de serviço para controlar os fluxos de mensagem. Em particular, um esquema de controle de taxa simples é introduzido, com base em um limite de taxa obrigatório, imposto de acordo com o usuário/tópico. Isso substitui uma estratégia mais complexa e gerencia a utilização antecipada de recursos na árvore em termos de memória, CPU e taxas de mensagem e bytes.

Inicialmente, as árvores são construídas e constantemente reavaliadas de acordo com um algoritmo de caminho mais curto (veja o Capítulo 3).

21.4.3 Resumo das principais escolhas de projeto para a comunicação

As escolhas de projeto globais relacionadas aos paradigmas de comunicação no Google estão resumidas na Figura 21.8. Essa tabela destaca as decisões mais importantes associadas ao projeto global e aos elementos constituintes (*buffers* de protocolo e o sistema de publicar-assinar) e resume o fundamento lógico e os compromissos específicos associados a cada escolha.

No todo, vimos uma estratégia mista que oferece dois paradigmas de comunicação distintos, projetados para suportar diferentes estilos de interação dentro da arquitetura. Isso permite aos desenvolvedores escolher o melhor paradigma para cada domínio de problema em particular.

Vamos repetir esse estilo de análise ao final de cada uma das seções a seguir, fornecendo, assim, uma perspectiva global das principais decisões de projeto relacionadas à infraestrutura do Google.

<i>Elemento</i>	<i>Escolha de projeto</i>	<i>Fundamento lógico</i>	<i>Compromissos</i>
<i>Buffers de protocolo</i>	O uso de uma linguagem para especificar formatos de dados	Flexível, pois a mesma linguagem pode ser usada para serializar dados para armazenamento ou comunicação	–
	Simplicidade da linguagem	Implementação eficiente	Falta de expressividade quando comparado, por exemplo, com XML
	Supporte para um estilo de RPC (recebendo uma única mensagem como parâmetro e retornando uma única mensagem como resultado)	Mais eficiente, extensível e suporta evolução do serviço	Falta de expressividade quando comparado com outros pacotes de RPC ou RMI
	Projeto agnóstico quanto ao protocolo	Podem ser usadas diferentes implementações de RPC	Nenhuma semântica comum para trocas RPC
<i>Publicar-assinar</i>	Estratégia baseada em tópicos	Suporta implementação eficiente	Menos expressiva do que as estratégias baseadas em conteúdo (atenuado pelos recursos de filtragem adicionais)
	Garantias de tempo real e confiabilidade	Suporta a manutenção de modos de exibição consistentes de maneira oportunamente	Suporte para algoritmo adicional exigido com sobregarga associada

Figura 21.8 Resumo das escolhas de projeto relacionadas aos paradigmas de comunicação.

21.5 Serviços de armazenamento de dados e coordenação

Apresentaremos agora os três serviços que, juntos, fornecem serviços de dados e coordenação para aplicações e serviços de nível mais alto: o Google File System, o Chubby e a Bigtable. São serviços complementares na infraestrutura do Google:

- O Google File System é um sistema de arquivos distribuído que oferece serviço semelhante ao do NFS e do AFS, conforme discutido no Capítulo 12. Ele dá acesso a dados não estruturados na forma de arquivos, mas otimizados para os estilos de dados e acesso a dados exigidos pelo Google (arquivos muito grandes, por exemplo).
- O Chubby é um serviço multifacetado, suportando, por exemplo, travas distribuídas para coordenação no ambiente distribuído e o armazenamento de quantidades muito pequenas de dados, complementando o armazenamento em larga escala oferecido pelo Google File System.
- A Bigtable dá acesso a dados mais estruturados na forma de tabelas, que podem ser indexadas de várias maneiras, inclusive por linha ou coluna. Portanto, a Bigtable é um estilo de banco de dados distribuído, mas, ao contrário de muitos bancos de dados, não suporta operadores totalmente relacionais (eles são vistos pelo Google como desnecessariamente complexos e não escaláveis).

Esses três serviços também são interdependentes. Por exemplo, a Bigtable usa o Google File System para armazenamento e o Chubby, para coordenação.

Examinaremos cada serviço em detalhes a seguir.

21.5.1 O Google File System (GFS)

O Capítulo 12 apresentou um estudo detalhado sobre o tema dos sistemas de arquivos distribuídos, analisando seus requisitos, sua arquitetura global e examinando dois estudos de caso em detalhes, a saber, NFS e AFS. Esses sistemas de arquivos são sistemas de arquivos distribuídos de *propósito geral* que oferecem abstrações de arquivo e diretório para uma ampla variedade de aplicações dentro e entre organizações. O Google File System (GFS) também é um sistema de arquivos distribuído; ele oferece abstrações semelhantes, mas especializadas para os requisitos específicos do Google, em termos de armazenamento e acesso a quantidades de dados muito grandes [Ghemawat *et al.* 2003]. Esses requisitos levaram a decisões de projeto muito diferentes das tomadas no NFS e no AFS (e, na verdade, em outros sistemas de arquivos distribuídos), conforme veremos a seguir. Começaremos nossa discussão sobre o GFS examinando os requisitos específicos identificados pelo Google.

Requisitos do GFS • O objetivo global do GFS é atender às necessidades exigentes e rapidamente crescentes do mecanismo de busca do Google e das diversas outras aplicações Web oferecidas pela empresa. A partir do entendimento desse domínio de operação em particular, o Google identificou os seguintes requisitos para o GFS (consulte Ghemawat *et al.* [2003]):

- O primeiro requisito é que o GFS deve funcionar de forma confiável na arquitetura física discutida na Seção 21.3.1 – ou seja, um sistema muito grande, construído a partir de *hardware* comum. Os projetistas do GFS começaram com a suposição de que os componentes falharão (não apenas componentes de *hardware*, mas também de *software*) e de que o projeto deve ser suficientemente tolerante a tais falhas para

permitir que serviços em nível de aplicação continuem sua operação diante de qualquer combinação provável de condições de falha.

- O GFS é otimizado para os padrões de utilização dentro do Google, tanto em termos dos tipos de arquivos armazenados como dos padrões de acesso a esses arquivos. O número de arquivos armazenados no GFS não é grande em comparação com outros sistemas, mas os arquivos tendem a ser pesados. Por exemplo, Ghemawat *et al* [2003] relatam a necessidade de talvez um milhão de arquivos tendo em média 100 megabytes de tamanho, mas com alguns arquivos na faixa dos gigabytes. Os padrões de acesso também são atípicos dos sistemas de arquivos em geral. Os acessos são dominados por leituras sequenciais em arquivos grandes e escritas sequenciais que anexam dados em arquivos, e o GFS é muito adequado a esse estilo de acesso. Pequenas leituras e escritas aleatórias ocorrem (estas últimas muito raramente) e são suportadas, mas o sistema não é otimizado para esses casos. Esses padrões de arquivo são influenciados, por exemplo, pelo armazenamento sequencial de muitas páginas Web em arquivos únicos, que são examinados por uma variedade de programas de análise de dados. O nível de acesso concorrente também é alto no Google, com grandes números de anexações concomitantes sendo particularmente predominantes, frequentemente acompanhadas de leituras concorrentes.
- O GFS deve satisfazer todos os requisitos da infraestrutura do Google como um todo; isto é; deve ser flexível (particularmente em termos de volume de dados e número de clientes), deve ser confiável, apesar da suposição a respeito das falhas mencionada anteriormente, deve funcionar bem e deve ser aberto no sentido de suportar o desenvolvimento de novas aplicações Web. Em termos de desempenho e dados os tipos de arquivo de dados armazenados, o sistema é otimizado para rendimento alto e prolongado na leitura de dados e isso tem prioridade em relação à latência. Não quer dizer que a latência não seja importante, mas sim que esse componente (o GFS) em particular precisa ser otimizado para leitura de alto desempenho e anexação de grandes volumes de dados para a operação correta do sistema como um todo.

Esses requisitos são acentuadamente diferentes dos existentes para o NFS e o AFS (por exemplo), que devem armazenar grandes números de arquivos frequentemente pequenos e em que as leituras e escritas aleatórias são comuns. Essas distinções levaram às decisões de projeto muito particulares, discutidas a seguir.

Interface do GFS • O GFS fornece uma interface de sistema de arquivos convencional, oferecendo um espaço de nomes hierárquico com os arquivos individuais identificados por nomes de caminho. Embora o sistema de arquivos não ofereça total compatibilidade com POSIX, muitas das operações são conhecidas dos usuários desses sistemas de arquivos (veja, por exemplo, a Figura 12.4):

- create* – cria uma nova instância de um arquivo;
- delete* – exclui uma instância de um arquivo;
- open* – abre um arquivo nomeado e retorna um identificador (*handle*);
- close* – fecha um arquivo especificado por um identificador (*handle*);
- read* – lê dados de um arquivo especificado;
- write* – escreve dados em um arquivo especificado.

Pode-se ver que as principais operações do GFS são muito parecidas com as do serviço de arquivo plano, descritas no Capítulo 12 (veja a Figura 12.6). Devemos supor que as

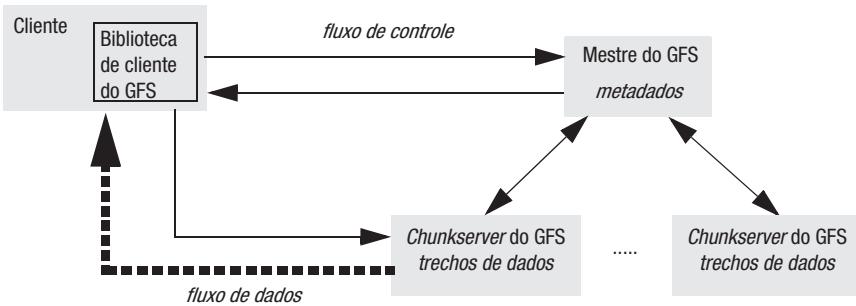


Figura 21.9 Arquitetura global do GFS.

operações *read* e *write* do GFS recebem um parâmetro especificando um deslocamento inicial dentro do arquivo, como acontece no serviço de arquivo plano.

A API também oferece duas operações mais especializadas, *snapshot* e *record append*. A primeira operação fornece um mecanismo eficiente para fazer uma cópia de um arquivo em particular ou da estrutura em árvore do diretório. A última suporta o padrão de acesso comum mencionado anteriormente, por meio do qual vários clientes realizam anexações concorrentes em determinado arquivo.

Arquitetura do GFS • A escolha de projeto mais influente no GFS é o armazenamento de arquivos em *trechos* de tamanho fixo, em que cada trecho tem 64 megabytes. Isso é muito grande, comparado aos outros projetos de sistema de arquivos. Em um nível, isso simplesmente reflete o tamanho dos arquivos armazenados no GFS. Em outro nível, essa decisão é fundamental para fornecer leituras sequenciais altamente eficientes e anexações de grandes volumes de dados. Voltaremos a esse ponto posteriormente, quando tivermos discutido mais detalhes da arquitetura do GFS.

Dada essa escolha de projeto, a tarefa do GFS é fornecer um mapeamento dos arquivos em trechos e, então, suportar operações padrão em arquivos, mapeando nas operações em trechos individuais. Isso é obtido com a arquitetura ilustrada na Figura 21.9, que mostra uma instância de um sistema de arquivos GFS mantida em um *cluster* físico. Cada *cluster* GFS tem um só *mestre* e vários *chunkservers* (*servidores de trecho*) (normalmente na ordem de centenas) que, juntos, fornecem um serviço de arquivo para grandes números de clientes que acessam os dados de forma concorrente.

A função do mestre é gerenciar *metadados* sobre o sistema de arquivos, definindo o espaço de nomes para arquivos, informações de controle de acesso e o mapeamento de cada arquivo em particular no conjunto de trechos associado. Além disso, todos os trechos são replicados (por padrão, em três *chunkservers* independentes, mas o nível de replicação pode ser especificado pelo programador). A localização das réplicas é mantida no mestre. A replicação é importante no GFS, para fornecer a confiabilidade necessária no caso de falhas (esperadas) de *hardware* e *software*. Isso contrasta com o NFS e o AFS, que não fornecem replicação com atualizações (veja o Capítulo 12).

Os principais metadados são armazenados de forma persistente em um registro de operação que suporta recuperação em caso de colapsos (novamente, aumentando a confiabilidade). Em particular, todas as informações mencionadas anteriormente são registradas, com exceção da localização das réplicas (esta última é recuperada por eleição de *chunkservers* e perguntando-se a eles quais réplicas armazenam no momento).

Embora o mestre seja centralizado e, portanto, um ponto único de falha, o registro de operações é replicado em várias máquinas remotas, de modo que, em caso de falha, o mestre pode ser prontamente restaurado. A vantagem de se ter um único mestre centralizado é que ele tem uma *visão global* do sistema de arquivos e, assim, pode tomar as melhores decisões de gerenciamento, relacionadas, por exemplo, ao posicionamento de trechos. Esse esquema também é mais simples de implementar, permitindo ao Google desenvolver o GFS em um período de tempo relativamente curto. McKusick e Quinlan [2010] apresentam o fundamento lógico dessa escolha de projeto bastante incomum.

Quando os clientes precisarem acessar dados a partir de um deslocamento de bytes em particular dentro de um arquivo, a biblioteca de cliente GFS primeiramente transformará isso em um par nome de arquivo e índice de trecho (facilmente computados, dado o tamanho fixo dos trechos). Então, isso é enviado para o mestre na forma de uma requisição RPC (usando *buffers* de protocolo). O mestre responde com o identificador de trecho apropriado e a localização das réplicas, e essas informações são colocadas na cache do cliente e usadas para acessar os dados por invocação RPC direta para um dos *chunkservers* replicados. Desse modo, o mestre é envolvido no início e depois fica completamente fora do lago, implementando uma separação de fluxos de controle e de dados – uma separação fundamental para manter o alto desempenho dos acessos a arquivo. Combinado com o tamanho grande dos trechos, isso significa que, uma vez identificado e localizado um trecho, os 64 megabytes podem ser lidos tão rapidamente quanto o servidor de arquivos e a rede permitam, sem quaisquer outras interações com o mestre, até que outro trecho precise ser acessado. Portanto, as interações com o mestre são minimizadas e o desempenho de saída, otimizado. O mesmo argumento se aplica às anexações sequenciais.

Note que outra repercussão do tamanho grande dos trechos é que o GFS mantém proporcionalmente menos metadados (se fosse adotado um tamanho de trecho de 64 quilobytes, por exemplo, o volume de metadados aumentaria por um fator de 1.000). Isso, por sua vez, significa que os mestres GFS geralmente podem manter todos os seus metadados na memória principal (mas veja a seguir), diminuindo significativamente a latência de operações de controle.

À medida que a utilização do sistema aumenta, têm surgido problemas com o esquema do mestre centralizado:

- Apesar da separação entre fluxo de controle e de dados e da otimização do desempenho do mestre, isso está aparecendo como um gargalo em seu projeto.
- Apesar do volume de metadados reduzido, decorrente do tamanho grande dos trechos, o volume de metadados armazenados em cada mestre está aumentando a um nível em que é difícil manter todos os metadados na memória principal.

Por esses motivos, agora o Google está trabalhando em um novo projeto que apresenta uma solução de mestre distribuído.

Uso de cache: como vimos no Capítulo 12, o *uso de cache* frequentemente desempenha um papel fundamental no desempenho e na escalabilidade de um sistema de arquivos (veja também a discussão mais geral sobre uso de cache na Seção 2.3.1). É interessante notar que o GFS não faz muito uso de cache. Conforme mencionado anteriormente, as informações sobre os locais de trechos são colocadas na cache dos clientes na primeira vez que são acessadas, para minimizar as interações com o mestre. Fora isso, o cliente não utiliza a cache. Em particular, os clientes GFS não colocam dados de arquivo na cache. Como a maioria dos acessos envolve fluxos sequenciais, por exemplo, leitura de conteúdo Web para produzir o índice invertido exigido, tais caches pouco contribuiriam

para o desempenho do sistema. Além disso, limitando o uso de cache aos clientes, o GFS também evita a necessidade de protocolos de coerência de cache.

O GFS também não fornece nenhuma estratégia em particular para uso de cache no lado do servidor (isto é, nos *chunkservers*), contando com cache de *buffer* no Linux para manter dados frequentemente acessados na memória.

Logging: o GFS também é um exemplo importante do uso de *logs* no Google para suportar depuração e análise de desempenho. Em particular, todos os servidores do GFS mantêm extensivos *logs* de diagnóstico que armazenam eventos significativos do servidor e todas as requisições e respostas RPC. Esses *logs* são monitorados continuamente e usados em caso de problemas no sistema, para identificar as causas subjacentes.

Gerenciando a consistência no GFS • Dado que os trechos são replicados no GFS, é importante manter a consistência das réplicas diante de operações que alteram os dados – ou seja, as operações *write* e *record append*. O GFS fornece uma estratégia para gerenciamento de consistência que:

- mantém a separação mencionada anteriormente entre controle e dados e, assim, permite atualizações de alto desempenho nos dados com envolvimento mínimo dos mestres;
- fornece uma forma de consistência relaxada, reconhecendo, por exemplo, a semântica específica oferecida por *record append*.

A estratégia funciona como segue.

Quando uma mutação (isto é, uma operação *write*, *append* ou *delete*) é requisitada para um trecho, o mestre concede um *arrendamento* de trecho para uma das réplicas, a qual é, então, designada como *primária*. Essa réplica primária é responsável por fornecer uma ordem serial para todas as mutações concorrentes pendentes para esse trecho. Assim, uma ordem global é fornecida pela ordenação dos arrendamentos de trecho, combinados com a ordem determinada por essa primária. Em particular, o arrendamento permite que a primária faça mutações em suas cópias locais e controle a ordem das mutações nas cópias secundárias; então, outra primária receberá o arrendamento e assim por diante.

Portanto, as etapas envolvidas nas mutações são as seguintes (ligeiramente simplificadas):

- Ao receber uma requisição de um cliente, o mestre concede um arrendamento para uma das réplicas (a primária) e retorna a identidade da primária e de outras réplicas (secundárias) para o cliente.
- O cliente envia todos os dados para as réplicas e isso é armazenado temporariamente em uma cache de *buffer* e não é escrito até que venham mais instruções (novamente, mantendo a separação do fluxo de controle do fluxo de dados, acoplada a um regime de controle leve, baseado em arrendamentos).
- Uma vez que todas as réplicas tenham confirmado o recebimento desses dados, o cliente envia uma requisição de escrita para a primária; então, a primária determina uma ordem serial para as requisições concorrentes e aplica as atualizações, nessa ordem, no local onde a primária se encontra.
- A primária solicita que as mesmas mutações, na mesma ordem, sejam realizadas nas réplicas secundárias, e estas enviam de volta uma confirmação, quando as mutações tiverem ocorrido.
- Se todas as confirmações forem recebidas, a primária relata o sucesso de volta para o cliente; caso contrário, é informada uma falha, indicando que a mutação ocorreu

na primária e em algumas das réplicas, mas não em todas. Isso é tratado como uma falha e deixa as réplicas em um estado inconsistente. O GFS procura resolver essa falha tentando fazer as mutações malsucedidas novamente. No pior caso, isso não terá êxito e, portanto, a consistência não será garantida pela estratégia.

É interessante relacionar esse esquema com as técnicas de replicação discutidas no Capítulo 18. O GFS adota uma arquitetura de replicação passiva, com uma modificação importante. Na replicação passiva, as atualizações são enviadas para a réplica primária e, então, esta fica responsável por enviar as atualizações subsequentes para os servidores de *backup* e por garantir que elas sejam coordenadas. No GFS, o cliente envia dados para todas as réplicas, mas a requisição vai para a primária que, então, fica responsável por escalarizar as mutações reais (a separação entre fluxo de dados e fluxo de controle, mencionada anteriormente). Isso permite que a transmissão de grandes quantidades de dados seja otimizada, independentemente do fluxo de controle.

Nas mutações há uma distinção importante entre as operações *write* e *record append*. A operação *write* especifica um deslocamento no qual as mutações devem ocorrer, enquanto a operação *record append* não faz isso (as mutações são aplicadas no final do arquivo, onde quer seja isso em determinado ponto no tempo). No primeiro caso, a localização é predeterminada, enquanto no último, o sistema decide. Operações *write* correntes no mesmo ponto não são serializadas e podem resultar em regiões corrompidas no arquivo. Nas operações *record append*, o GFS garante que a anexação ocorrerá pelo menos uma vez e atomicamente (isto é, como uma sequência contínua de bytes); contudo, o sistema não garante que todas as cópias do trecho serão idênticas (algumas podem ter dados duplicados). Novamente, é útil relacionar isso com a matéria do Capítulo 18. As estratégias de replicação do Capítulo 18 são todas de propósito geral, enquanto esta estratégia é específica do domínio e flexibiliza as garantias de consistência, sabendo-se que a semântica resultante pode ser tolerada pelas aplicações e serviços do Google (outro exemplo de replicação específica do domínio – o algoritmo de replicação de Xu e Liskov [1989] para espaços de tuplas – pode ser encontrado na Seção 6.5.2).

21.5.2 Chubby

O Chubby [Burrows 2006] é um serviço fundamental da infraestrutura do Google, oferecendo serviços de armazenamento e coordenação para outros serviços da infraestrutura, incluindo o GFS e a Bigtable. O Chubby é um serviço multifacetado que oferece quatro recursos distintos:

- Ele fornece travas distribuídas para sincronizar atividades distribuídas no que é um ambiente assíncrono de larga escala.
- Ele fornece um sistema de arquivos que oferece armazenamento confiável de pequenos arquivos (complementando o serviço oferecido pelo GFS).
- Ele pode ser usado para suportar a eleição da réplica primária em um conjunto de réplicas (conforme é necessário, por exemplo, pelo GFS, conforme discutido na Seção 21.5.1).
- Ele é usado como serviço de nomes dentro Google.

À primeira vista, pode parecer que isso contradiz o princípio de projeto global da simplicidade (fazer apenas uma coisa e fazê-la bem). Entretanto, à medida que desvendarmos o projeto do Chubby, veremos que no centro há um serviço básico que oferece uma solução para o *consenso distribuído* e que as outras facetas surgem desse serviço básico, que é otimizado para o estilo de utilização dentro do Google.

Iniciaremos nosso estudo do Chubby examinando a interface que ele oferece; em seguida, veremos a arquitetura de um sistema Chubby em detalhes e como isso é mapeado na arquitetura física. Concluiremos o exame vendo em detalhes a implementação do algoritmo de consenso que há no centro do Chubby, o *Paxos*.

Interface do Chubby • O Chubby fornece uma abstração baseada em um sistema de arquivos, adotando a visão inicialmente promovida no Plan 9 [Pike *et al.* 1993], de que todo objeto de dados é um arquivo. Os arquivos são organizados em um espaço de nomes hierárquico, utilizando estruturas de diretório com os nomes tendo a seguinte forma:

/ls/célula_chubby/nome_diretório/.../nome_arquivo

onde */ls* se refere ao serviço de travas, designando que isso faz parte do sistema Chubby, *célula_chubby* é o nome de uma instância em particular de um sistema Chubby (o termo *célula* é usado no Chubby para denotar uma instância do sistema). Isso é seguido por uma série de nomes de diretório, culminando em um *nome_arquivo*. Um nome especial */ls/local*, será solucionado na célula mais local em relação ao aplicativo ou serviço que fez a chamada.

O Chubby começou sua existência como um serviço de travas e a intenção era que tudo fosse uma trava no sistema. Entretanto, rapidamente ficou claro que seria útil associar quantidades (normalmente pequenas) de dados às entidades do Chubby – veremos um exemplo disso a seguir, quando examinarmos como o Chubby é usado em eleições de réplica primária. Assim, no Chubby as entidades compartilham a funcionalidade de travas e arquivos; elas podem ser usadas exclusivamente como travas, para armazenar pequenas quantidades de dados ou para associar pequenas quantidades de dados (na verdade, metadados) a operações de trava.

Uma versão ligeiramente simplificada da API oferecida pelo Chubby aparece na Figura 21.10. *Open* e *Close* são operações padrão, com *Open* recebendo um arquivo ou diretório nomeado e retornando um *identificador* Chubby para essa entidade. O cliente pode especificar vários parâmetros associados a *Open*, incluindo declarar a utilização pretendida (por exemplo, leitura, escrita ou trava), e as verificações de permissões são realizadas nesse estágio, usando listas de controle de acesso. A operação *Close* simplesmente abre mão do uso do identificador. A operação *Delete* é usada para remover o arquivo ou diretório (essa operação falhará se for aplicada a um diretório com filhos).

Na função de sistema de arquivos, o Chubby oferece um pequeno conjunto de operações para leitura e escrita de *arquivos inteiros*; são operações simples que retornam os dados completos do arquivo e escrevem os dados completos no arquivo. Essa estratégia de arquivo inteiro é adotada para desencorajar a criação de arquivos grandes, pois não é esse uso pretendido para o Chubby. A primeira operação, *GetContentsAndStat*, retorna o conteúdo do arquivo e quaisquer metadados associados ao arquivo (uma operação associada, *GetStat*, retorna apenas os metadados; também é fornecida uma operação *ReadDir* para ler nomes e metadados associados aos filhos de um diretório. *SetContents* escreve o conteúdo de um arquivo e *SetACL* fornece uma maneira de configurar os dados da lista de controle de acesso. A leitura e a escrita de arquivos inteiros são operações *atômicas*.

Na função de ferramenta de gerenciamento de travas, as principais operações fornecidas são *Acquire*, *TryAcquire* e *Release*. *Acquire* e *Release* correspondem às operações de mesmo nome apresentadas na Seção 16.4; *TryAcquire* é uma variante não bloqueante de *Acquire*. Note que, embora as travas sejam *consultivas* no Chubby, um aplicativo ou serviço deve passar pelo protocolo correto de adquirir e liberar travas. Os desenvolvedores do Chubby consideraram uma alternativa de travas obrigatórias, segundo a qual os dados

Função	Operação	Efeito
Geral	<i>Open</i>	Abre um arquivo ou diretório dado e retorna um identificador
	<i>Close</i>	Fechá o arquivo associado ao identificador
	<i>Delete</i>	Exclui o arquivo ou diretório
Arquivo	<i>GetContentsAndStat</i>	Retorna (atomicamente) o conteúdo do arquivo inteiro e os metadados associados ao arquivo
	<i>GetStat</i>	Retorna apenas os metadados
	<i>ReadDir</i>	Retorna o conteúdo de um diretório – isto é, os nomes e metadados de quaisquer filhos
	<i>SetContents</i>	Escreve todo o conteúdo de um arquivo (atomicamente)
	<i>SetACL</i>	Escreve novas informações na lista de controle de acesso
Trava	<i>Acquire</i>	Adquire uma trava sobre um arquivo
	<i>TryAcquire</i>	Tenta adquirir uma trava sobre um arquivo
	<i>Release</i>	Libera a trava

Figura 21.10 API do Chubby.

travados ficam inacessíveis para todos os outros usuários e isso é imposto pelo sistema, mas a flexibilidade extra e resiliência das travas consultivas foram preferidas, deixando para o programador a responsabilidade pela verificação de conflitos [Burrows 2006].

Se uma aplicação precisa proteger um arquivo contra acesso concorrente, ela pode usar as duas funções juntas, armazenando dados no arquivo e adquirindo travas antes de acessar esses dados.

O Chubby também pode ser usado para suportar uma *eleição de primária* em sistemas distribuídos – isto é, a eleição de uma réplica como a primária no gerenciamento de replicação passiva (consulte as Seções 15.3 e 18.3.1 para discussões sobre algoritmos de eleição e replicação passiva, respectivamente). Primeiramente, todas as primárias candidatas tentam adquirir uma trava associada à eleição. Somente uma terá sucesso. Essa candidata se torna a primária, com todas as outras candidatas se tornando secundárias. A primária registra sua vitória escrevendo sua identidade no arquivo associado, e os outros processos podem então determinar a identidade da primária lendo esses dados. Conforme mencionado anteriormente, esse é um importante exemplo de combinação das funções de trava e arquivo para um propósito útil em um sistema distribuído. Isso também mostra como a eleição de primária pode ser implementada em cima de um serviço de consenso, como uma alternativa a algoritmos como a estratégia baseada em anel ou o algoritmo valentão (*bully*), apresentado na Seção 15.3.

Por fim, o Chubby suporta um *mecanismo de evento* simples, permitindo aos clientes se registrarem ao abrir um arquivo, para receber mensagens de evento relacionadas ao arquivo. Mais especificamente, o cliente pode se inscrever em diversos eventos, como uma opção na chamada de *Open*. Então, os eventos associados são entregues de forma assíncrona, por meio de *callbacks*. Exemplos de eventos incluem a modificação do conteúdo de um arquivo, um identificador se tornando inválido, etc.

O Chubby é uma interface de programação de sistema de arquivos reduzida, comparada, por exemplo, com o POSIX. O Chubby não apenas exige que as operações de leitura e atualização sejam aplicadas em arquivos inteiros, como também não suporta operações para mover arquivos entre diretórios nem vínculos simbólicos (*softlinks*) ou restritos (*hard links*). Além disso, o Chubby mantém apenas metadados limitados (relacionados a controle de acesso, controle de versão e uma soma de verificação para proteger a integridade dos dados).

Arquitetura do Chubby • Conforme mencionado anteriormente, uma única instância de um sistema Chubby é conhecida como célula; cada célula consiste em um número relativamente pequeno de réplicas (normalmente, cinco), com uma delas designada como mestra. Os aplicativos clientes acessam esse conjunto de réplicas por meio de uma biblioteca Chubby, a qual se comunica com os servidores remotos usando o serviço RPC descrito na Seção 21.4.1. As réplicas são colocadas em locais independentes de falha para minimizar o potencial de falhas correlacionadas – por exemplo, elas não estarão contidas dentro do mesmo *rack*. Normalmente, todas as réplicas estariam contidas dentro de determinado *cluster* físico, embora isso não seja exigido para o funcionamento correto do protocolo e tenham sido criadas células experimentais que abrangem os centros de dados do Google.

Cada réplica mantém um pequeno banco de dados, cujos elementos são entidades no espaço de nomes do Chubby – isto é, diretórios e arquivos/travas. A consistência do banco de dados replicado é obtida usando-se um protocolo de consenso subjacente (uma implementação do algoritmo Paxos de Lamport [Lamport 1989, Lamport 1998]), baseado na manutenção de *logs de operação* (veremos a implementação desse protocolo, a seguir). Como os *logs* se tornam muito grandes com o passar do tempo, o Chubby também suporta a criação de *instantâneos* – visões completas do estado do sistema em determinado ponto no tempo. Uma vez tirado um instantâneo, os *logs* anteriores podem ser excluídos, com o estado consistente do sistema em qualquer ponto determinado pelo instantâneo anterior, junto às aplicações do conjunto de operações no *log*. Essa estrutura global está mostrada na Figura 21.11.

Uma *sessão* do Chubby é uma relação entre um cliente e uma célula do Chubby. Isso é mantido usando-se *handshakes KeepAlive* entre as duas entidades. Para aumentar o desempenho, a biblioteca do Chubby implementa *cache no cliente*, armazenando dados de arquivo, metadados e informações em tratadores. Em contraste com o GFS (com suas grandes leituras e anexações sequenciais), a cache no cliente é eficiente no Chubby, com seus arquivos pequenos que provavelmente serão acessados repetidamente. Por causa desse uso de cache, o sistema precisa manter a consistência entre um arquivo e uma cache, assim como entre as diferentes réplicas do arquivo. A *consistência de cache* exigida no Chubby é obtida como segue. Quando uma mutação está para ocorrer, a operação associada (por exemplo, *SetContents*) é bloqueada até que todas as caches associadas sejam invalidadas (por eficiência, as requisições de invalidação vão “de carona” nas respostas de *KeepAlive* da mestra, com as respostas enviadas imediatamente quando ocorre uma invalidação). Os dados colocados em cache também nunca são atualizados diretamente.

O resultado final é um protocolo de consistência de cache muito simples que apresenta semântica determinística para os clientes do Chubby. Compare isso com o regime de cache no cliente do NFS, por exemplo, no qual as mutações não acarretam a atualização imediata das cópias colocadas em cache, resultando em versões potencialmente diferentes de arquivos em diferentes nós clientes. Também é interessante comparar isso com o protocolo de consistência de cache do AFS, mas deixamos como exercício para o leitor (veja o Exercício 21.7).

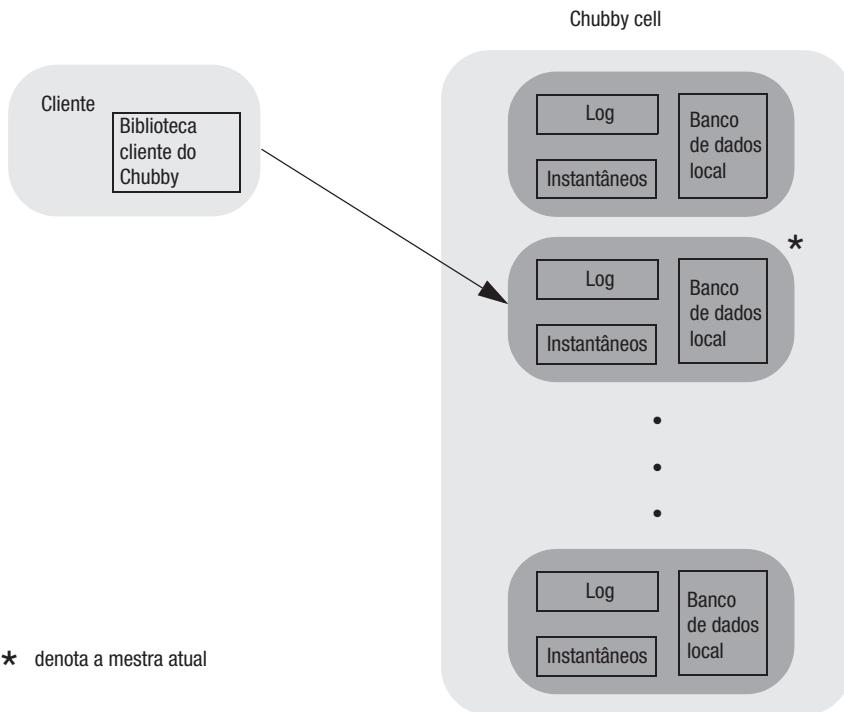


Figura 21.11 Arquitetura global do Chubby.

Esse determinismo é importante para muitas das aplicações e serviços de cliente que usam o Chubby (por exemplo, a Bigtable, conforme discutido na Seção 21.5.3) para armazenar listas de controle de acesso. A Bigtable exige atualização consistente de listas de controle de acesso, entre todas as réplicas e em termos de cópias colocadas na cache. Note que foi esse determinismo que levou ao uso do Chubby como servidor de nomes dentro do Google. Mencionamos, na Seção 13.2.3, que o DNS permite que a atribuição de nomes de dados se torne inconsistente. Embora isso seja tolerável na Internet, os desenvolvedores da infraestrutura do Google preferiram a visão mais consistente oferecida pelo Chubby, usando arquivos Chubby para manter os mapeamentos de nomes para endereços. Burrows [2006] discute em mais detalhes o uso do Chubby como serviço de nomes.

Implementação do Paxos • Paxos é uma família de protocolos que fornece consenso distribuído (veja a Seção 15.5 para uma discussão mais ampla sobre protocolos de consenso distribuído). Os protocolos de consenso operam sobre um conjunto de réplicas com o objetivo de obter um acordo entre os servidores que gerenciam as réplicas, para atualizá-las com um valor comum. Isso é obtido em um ambiente onde:

- Os servidores de réplica podem operar em velocidade arbitrária e podem falhar (e subsequentemente se recuperar).
- Os servidores de réplica têm acesso a um armazenamento estável e persistente que sobrevive aos colapsos.
- Mensagens podem ser perdidas, reordenadas ou duplicadas. Elas são entregues sem corrupção, mas podem demorar um tempo arbitrariamente longo para serem entregues.

Portanto, o Paxos é fundamentalmente um protocolo de consenso distribuído para *sistemas assíncronos* (veja a Seção 2.4.1) e, de fato, é a oferta dominante nesse universo. Os desenvolvedores do Chubby enfatizam que as suposições anteriores refletem a verdadeira natureza dos sistemas baseados na Internet, como o Google, e alertam os profissionais do setor sobre algoritmos de consenso que fazem suposições mais fortes (por exemplo, algoritmos para sistemas síncronos) [Burrows 2006].

Lembre-se, do Capítulo 15, de que é impossível garantir a consistência em sistemas assíncronos, mas que foram propostas várias técnicas para contornar esse problema. O Paxos funciona garantindo a exatidão, mas não há garantias de que o Paxos termine a execução (voltaremos a essa questão, a seguir, quando tivermos visto os detalhes do algoritmo).

O algoritmo foi apresentado pela primeira vez por Leslie Lamport, em 1989, em um artigo chamado *The Part-Time Parliament* (O parlamento de meio-expediente) [Lamport 1989, Lamport 1998]. Inspirado por sua descrição dos generais bizantinos (conforme discutido na Seção 15.5.1), ele novamente apresentou o algoritmo fazendo uma analogia, desta vez se referindo ao comportamento de um parlamento mitológico na ilha grega de Paxos. Em seu site [research.microsoft.com], Lamport escreve de forma engraçada sobre a reação a essa apresentação.

No algoritmo, qualquer réplica pode submeter um valor com o objetivo de obter consenso sobre um valor final. No Chubby, o acordo equivale a todas as réplicas terem esse valor como a próxima entrada em seus *logs* de atualização, obtendo, assim, uma visão consistente dos *logs* de todos os locais. É garantido que o algoritmo obtenha um consenso, se a maioria das réplicas executar o bastante, com suficiente estabilidade da rede. Mais formalmente, Kirsch e Amir [2008] apresentam as seguintes propriedades para o Paxos:

Paxos-L1 (Progresso): se existe uma maioria estável em um conjunto de servidores, então, se um servidor do conjunto inicia uma atualização, algum membro do conjunto executará a atualização.

Paxos-L2 (Replicação Eventual): se um servidor s executa uma atualização e existe um conjunto de servidores contendo s e r , e um tempo após o qual o conjunto não experimenta falhas de comunicação ou processo, então r executará a atualização.

A intuição aqui diz que o algoritmo não pode garantir que a consistência seja atingida quando a rede se comporta de forma assíncrona, mas atingirá a consistência quando forem experimentadas condições mais síncronas (ou estáveis).

O algoritmo Paxos: O algoritmo Paxos funciona como segue:

Etapa 1: o algoritmo conta com a capacidade de eleger um *coordenador* para determinada decisão consensual. Reconhecendo que os coordenadores podem falhar, é adotado um processo de eleição flexível que pode resultar na coexistência de vários coordenadores, antigos e novos, com o objetivo de reconhecer e rejeitar mensagens de antigos coordenadores. Para identificar o coordenador correto, os coordenadores são ordenados por um *número sequencial*. Cada réplica mantém o número sequencial mais alto visto até o momento e, se estiver fazendo uma oferta para ser um coordenador, escolherá um número *exclusivo* mais alto e divulgará isso para todas as réplicas em uma mensagem de *proposta*.

Claramente, é importante que o número sequencial escolhido por um coordenador em potencial seja realmente exclusivo – dois (ou mais) coordenadores não devem ser capazes de escolher o mesmo valor. Vamos supor que tenhamos n réplicas. Um número sequencial exclusivo pode ser garantido se toda réplica receber

um identificador exclusivo, i_r , entre 0 e $n-1$ e, então, selecionar o menor número sequencial s maior do que quaisquer números sequenciais vistos até o momento, de modo que $s \bmod n = i_r$ (por exemplo, se o número de réplicas é 5, examinamos a réplica com o identificador exclusivo 3 e o último número sequencial visto foi 20, então essa réplica escolherá o número sequencial 23 para sua próxima oferta).

Se outras réplicas não tiverem visto um licitante maior, elas respondem com uma mensagem de *promessa*, indicando que se comprometem a não negociar com outros coordenadores (isto é, mais antigos) com números sequenciais menores, ou enviam uma confirmação negativa, indicando que não votarão nesse coordenador. Cada mensagem de *promessa* contém também o valor mais recente que o remetente recebeu como proposta de consenso; esse valor pode ser nulo, caso nenhuma outra proposta tenha sido observada. Se a maioria das mensagens de *promessa* foi recebida, a réplica receptora é eleita como coordenador, com a maioria das réplicas que apóiam esse coordenador conhecida como *quórum*.

Etapa 2: o coordenador eleito deve selecionar um valor e , subsequentemente, enviar uma mensagem *aceito* com esse valor para o quórum associado. Se alguma das mensagens de *promessa* continha um valor, então o coordenador deve escolher um valor (qualquer valor) do conjunto de valores que tiver recebido; caso contrário, o coordenador está livre para selecionar seu próprio valor. Qualquer membro do quórum que receba a mensagem *aceito* deve aceitar o valor e , então, *confirmar* a aceitação. O coordenador espera, possivelmente indefinidamente no algoritmo, que a maioria das réplicas confirme a mensagem de *aceito*.

Etapa 3: se a maioria das réplicas confirmar, então um consenso foi efetivamente obtido. O coordenador divulga uma mensagem de *confirmação* para notificar as réplicas sobre esse acordo. Caso contrário, o coordenador abandona a proposta e começa novamente.

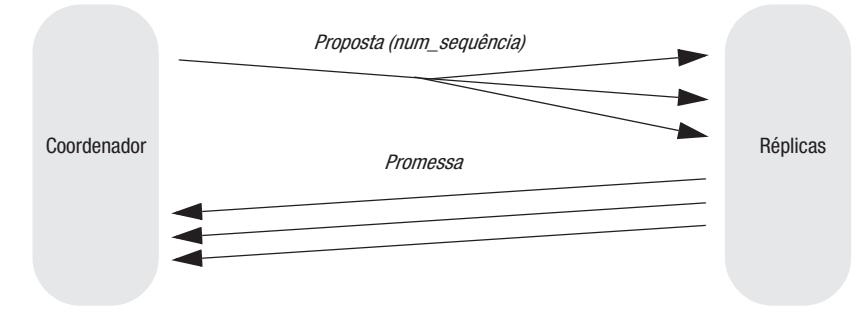
Note que a terminologia anterior é a utilizada pelo Google, por exemplo, em [Chandra *et al.* 2007]. Na literatura, as descrições do protocolo podem usar outra terminologia, por exemplo, baseada nas funções dos proponentes, aceitantes, apredizes, etc.

Portanto, na ausência de falhas o consenso é obtido com as trocas de mensagem mostradas na Figura 21.12. O algoritmo também é seguro na presença de falhas – por exemplo, a falha de um coordenador ou de outra réplica ou problemas com mensagens perdidas, reordenadas, duplicadas ou atrasadas, conforme discutido anteriormente. Uma prova de exatidão está fora dos objetivos deste livro, mas conta pesadamente com a ordenação imposta pela etapa 1, acoplada com o fato de que, graças ao mecanismo de quórum, se duas maiorias tiverem concordado com um valor proposto, haverá pelo menos uma réplica em comum que concordou com ambas. O mecanismo de quórum também garante comportamento correto se a rede particionar, pois somente no máximo uma partição será capaz de compor uma maioria.

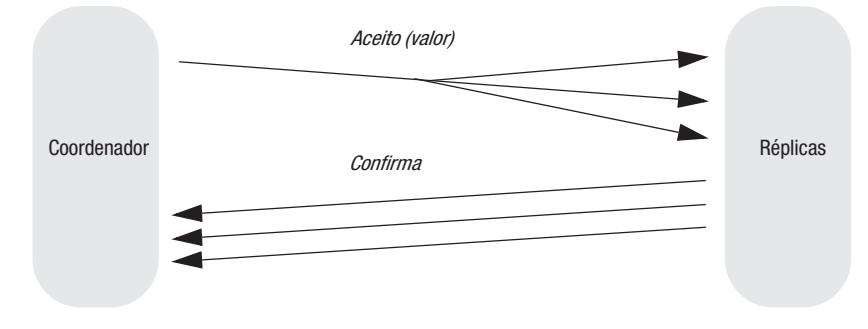
Voltando à questão do término, é possível que o Paxos não termine se dois proponentes competirem entre si e fizerem ofertas indefinidamente mais altas em termos de números sequenciais. Isso é coerente com o resultado da impossibilidade de Fischer *et al.* [1985], a respeito de garantias de consenso absolutas em sistemas assíncronos.

Problemas de implementação adicionais: no Chubby, não é suficiente chegar a um acordo sobre um único valor; há necessidade de chegar a um acordo sobre uma sequência de valores. Portanto, na prática o algoritmo precisa repetir um conjunto de entradas no *log*. Isso é referido como Multi-Paxos por Chandra *et al.* [2007]. No Multi-Paxos, certas otimizações são possíveis, incluindo a eleição de um coordenador por um período de tempo (potencialmente longo), evitando, assim, execuções repetidas da etapa 1.

Etapa 1: elegendo um coordenador



Etapa 2: buscando o consenso



Etapa 3: obtendo o consenso

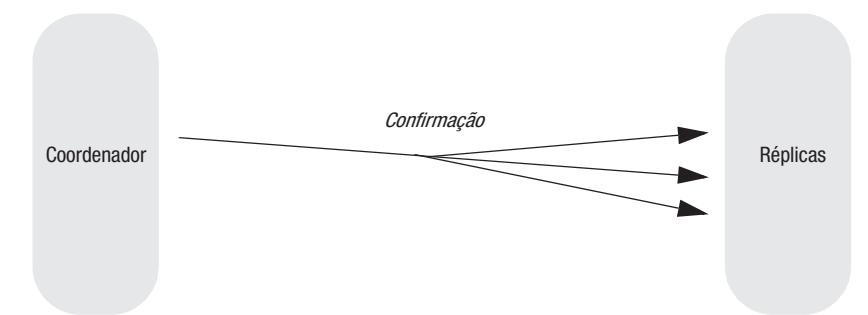


Figura 21.12 Trocas de mensagem no Paxos (na ausência de falhas).

O artigo de Chandra *et al.* [2007] também discute os desafios de engenharia da implementação do Paxos em um cenário real, em particular no complexo ambiente de sistema distribuído oferecido pela infraestrutura do Google. Em um texto divertido e instrutivo, eles discutem os desafios de passar da descrição algorítmica e da prova formal para fazer o algoritmo funcionar eficientemente como parte do sistema Chubby, incluindo tratar de corrupções de disco e outros eventos contextuais, como atualizações de sistema. O artigo enfatiza a importância de um regime rigoroso de teste, especialmente para esses blocos de construção importantes de sistemas tolerantes a falhas, em consonância com o princípio global de testes extensivos do Google, mencionado na Seção 21.3.

21.5.3 Bigtable

O GFS oferece um sistema para armazenar e acessar arquivos “planos” grandes, cujo conteúdo é acessado em relação a deslocamentos de byte dentro de um arquivo, permitindo aos programas armazenar grandes quantidades de dados e executar operações de leitura e escrita (especialmente anexação) otimizadas para o uso típico dentro da organização. Embora esse seja um bloco de construção importante, ele não é suficiente para atender a todas as necessidades de dados do Google. Existe uma forte necessidade de um sistema de armazenamento distribuído que dê acesso a dados indexados de maneiras mais sofisticadas em relação ao seu conteúdo e estrutura. A pesquisa na Web e praticamente todas as outras aplicações do Google, incluindo a infraestrutura de esquadrinhamento, Google Earth/Maps, Google Analytics e pesquisas personalizadas, usam acesso a dados estruturados. O Google Analytics, por exemplo, armazena em uma tabela as informações sobre cliques brutos associados a usuários visitando um *site* e resume as informações analisadas em uma segunda tabela. A primeira tem cerca de 200 terabytes de tamanho e a última, 20 terabytes. (A análise é feita com MapReduce, descrito na Seção 21.6, a seguir).

Uma escolha para o Google seria implementar (ou reutilizar) um banco de dados distribuído, por exemplo, um banco de dados relacional com um conjunto completo de operadores relacionais fornecidos (por exemplo, *union*, *selection*, *projection*, *intersection* e *join*). No entanto, obter bom desempenho e escalabilidade em tais bancos de dados distribuídos é reconhecidamente um problema difícil e, fundamentalmente, os estilos de aplicação oferecidos pelo Google não exigem essa funcionalidade completa. Portanto, o Google introduziu a Bigtable [Chang *et al.* 2008], a qual mantém o modelo de *tabela* oferecido pelos bancos de dados relacionais, mas com uma interface muito mais simples, conveniente para o estilo de aplicações e serviços oferecidos pelo Google e projetado para suportar o armazenamento eficiente e a recuperação de conjuntos de dados estruturados bastante grandes. Descreveremos essa interface com alguns detalhes a seguir, antes de examinarmos a arquitetura interna da Bigtable, destacando como essas propriedades são obtidas.

Interface da Bigtable • A Bigtable é um sistema de armazenamento distribuído que suporta o armazenamento de volumes potencialmente enormes de dados estruturados. O nome (“tabela extensa”) é uma forte indicação do que ela oferece, fornecendo armazenamento para o que são tabelas muito grandes (frequentemente na faixa dos terabytes). Mais precisamente, a Bigtable suporta o armazenamento tolerante a falhas, a criação e a exclusão de tabelas, no qual uma tabela é uma estrutura tridimensional contendo células indexadas por uma chave de linha, uma chave de coluna e um carimbo de tempo:

Linhas: cada linha em uma tabela tem uma chave associada que é um *string* arbitrário de até 64 quilobytes de tamanho, embora a maioria das chaves seja significativamente menor. Uma chave de linha é mapeada pela Bigtable no endereço de uma linha. Uma linha contém potencialmente grandes volumes de dados sobre uma entidade, como uma página Web. Dado que dentro Google é normal processar informações sobre páginas Web, é bastante comum, por exemplo, as chaves de linha serem URLs, com a linha contendo, então, informações sobre os recursos referenciados pelos URLs. A Bigtable mantém uma ordenação lexicográfica de determinada tabela pela chave de linha e isso tem algumas repercussões interessantes. Em particular, conforme veremos a seguir, quando examinarmos a arquitetura subjacente, subsequências de linhas são mapeadas em *tablets*, as quais representam a unidade de distribuição e posicionamento. Assim, é vantajoso gerenciar a localidade atribuindo chaves de linha que estarão próximas ou mesmo adjacentes na ordem lexicográfica. Isso significa que os URLs podem ser uma má escolha para chaves, mas URLs com a parte do domínio invertida fornecem localidade muito mais forte para acessos a

dados, pois os domínios comuns serão classificados juntos, suportando análise de domínio. Para ilustrar isso, considere as informações relacionadas a esportes armazenadas no site da BBC. Se essas informações forem armazenadas sob URLs como www.bbc.co.uk/sports e www.bbc.co.uk/football, então a classificação resultante será bastante aleatória e dominada pela ordem lexicográfica dos primeiros campos. No entanto, se forem armazenadas sob uk.co.bbc.www/sport e uk.co.bbc.www/football, é provável que as informações sejam armazenadas no mesmo *tablet*. Deve-se enfatizar que essa atribuição de chaves fica totalmente por conta do programador, de modo que eles devem conhecer essa propriedade (ordenação) para explorar o sistema da melhor forma. Para tatar dos problemas da concorrência, todos os acessos às linhas são atômicos (ecoando decisões de projeto semelhantes no GFS e no Chubby).

Colunas: a atribuição de nomes de colunas é mais estruturada do que para linhas. As colunas são organizadas em várias *famílias de coluna* – agrupamentos lógicos nos quais os dados sob uma família tendem a ser do mesmo tipo, com colunas individuais designadas por *qualificadores* dentro das famílias. Em outras palavras, determinada coluna é referenciada usando-se a sintaxe *família:qualificador*, onde *família* é um *string* imprimível e *qualificador* é um *string* arbitrário. O objetivo é ter um número de famílias relativamente pequeno para determinada tabela, mas um número potencialmente grande de colunas (designadas por qualificadores distintos) dentro de uma família. Usando o exemplo de Chang *et al.* [2008], isso pode ser usado para estruturar dados associados a páginas Web, com as famílias válidas sendo o *conteúdo*, quaisquer âncoras associadas à página e a *linguagem* utilizada na página Web. Se um nome de família se refere a apenas uma coluna, é possível omitir o qualificador. Por exemplo, uma página Web terá um campo de conteúdo e isso pode ser referenciado usando-se o nome de chave *contents*:

Carimbos de tempo: qualquer célula dentro da Bigtable também pode ter várias versões indexadas pelo carimbo de tempo (*timestamp*), em que o carimbo de tempo está relacionado ao tempo real ou pode ser um valor arbitrário atribuído pelo programador (por exemplo, um *tempo lógico*, conforme discutido na Seção 14.4, ou um identificador de versão). As várias versões são classificadas pelo carimbo de tempo inverso, com a versão mais recente disponível primeiro. Esse recurso pode ser usado, por exemplo, para armazenar diferentes versões dos mesmos dados, incluindo o conteúdo de páginas Web, permitindo que seja feita uma análise sobre dados históricos, assim como sobre dados atuais. As tabelas podem ser configuradas para aplicar automaticamente coleta de lixo em versões mais antigas, reduzindo, assim, a carga sobre o programador para gerenciar os grandes conjuntos de dados e versões associadas. Essa abstração de tabela tridimensional está ilustrada na Figura 21.13.

A Bigtable suporta uma API que fornece uma grande variedade de operações, incluindo:

- a criação e a exclusão de tabelas;
- a criação e a exclusão de famílias de coluna dentro de tabelas;
- o acesso a dados de determinadas linhas;
- escrever ou excluir valores de célula;
- realizar mutações atômicas de linha, incluindo acessos a dados e operações de escrita e exclusão associadas (transações mais globais entre linhas não são suportadas);
- iterar sobre diferentes famílias de coluna, incluindo o uso de expressões regulares para identificar intervalos de coluna;
- associar metadados, como informações de controle de acesso, a tabelas e famílias de coluna.

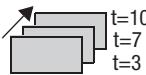
		Famílias de coluna e qualificadores				
		FC ₁ :	FC ₂ :q ₁	FC ₂ :q ₂	FC ₃ :q ₁	FC ₃ :q ₂
Linhas	L ₁					
	L ₂				t=10 t=7 t=3	carimbos de tempo
	L ₃					
	L ₄					
	L ₅					

Figura 21.13 A abstração de tabela na Bigtable.

Como pode-se ver, isso é consideravelmente mais simples do que um banco de dados relacional, mas bem conveniente para os estilos de aplicação dentro do Google. Chang *et al.* [2008] discutem como essa interface suporta o armazenamento de tabelas de dados em páginas Web (em que as linhas representam páginas Web individuais e as colunas representam dados e metadados associados a essa página Web dada), o armazenamento de dados brutos e processados do Google Earth (com as linhas representando segmentos geográficos e as colunas sendo as diferentes imagens disponíveis para esse segmento) e também dados para suportar o Google Analytics (por exemplo, mantendo uma tabela de cliques brutos, em que as linhas representam uma sessão de usuário final e as colunas, a atividade associada).

A arquitetura global do sistema subjacente está apresentada a seguir.

Arquitetura da Bigtable • Uma Bigtable é decomposta em *tablets*, com determinado *tablet* tendo um tamanho de aproximadamente 100–200 megabytes. Portanto, as principais tarefas da infraestrutura da Bigtable são gerenciar *tablets* e suportar as operações descritas anteriormente para acessar e alterar os dados estruturados associados. A implementação também tem a tarefa de mapear a estrutura de *tablets* no sistema de arquivos subjacente (GFS) e garantir o balanceamento de carga eficiente no sistema. Conforme veremos a seguir, a Bigtable faz uso pesado do GFS e do Chubby para o armazenamento de dados e a coordenação distribuída.

Uma única instância de uma implementação de Bigtable é conhecida como *grupo (cluster)*, e cada grupo pode armazenar várias tabelas. A arquitetura de um grupo da Bigtable é semelhante à do GFS, consistindo em três componentes principais (como mostrado na Figura 21.14):

- um componente *biblioteca* no lado do cliente;
- um *servidor mestre*;
- um número potencialmente grande de *servidores de tablets*.

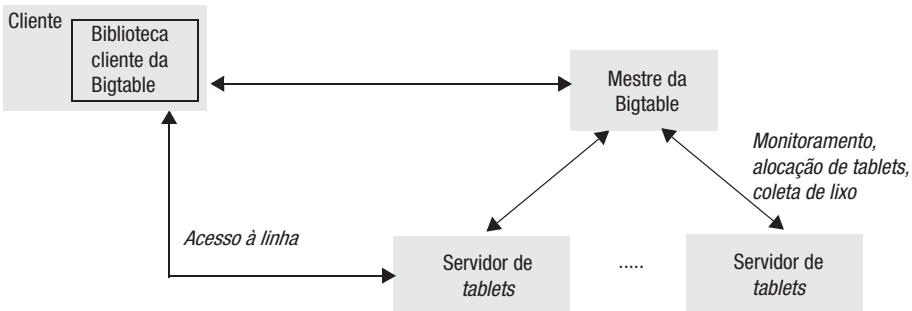


Figura 21.14 Arquitetura global da Bigtable.

Em termos de escala, Chang *et al.* relatam que, em 2008, 388 grupos de produção funcionavam em vários *clusters* de máquina do Google, com uma média em torno de 63 servidores de *tablets* por grupo, mas com muitos sendo significativamente maiores (alguns com mais de 500 servidores de *tablets* por grupo). O número de servidores de *tablets* por grupo também é dinâmico e é comum adicionar novos servidores de *tablets* no sistema em tempo de execução para aumentar o desempenho de saída.

Duas das principais decisões de projeto na Bigtable são idênticas às tomadas para o GFS. Primeiramente, a Bigtable adota uma estratégia de *mestre único*, exatamente pelos mesmos motivos – ou seja, manter uma visão centralizada do estado do sistema, suportando assim as melhores decisões de posicionamento e balanceamento de carga, e por causa da inherente simplicidade na implementação dessa estratégia. Em segundo lugar, a implementação mantém uma rigorosa *separação entre controle e dados*, com o regime de controle leve mantido pelo mestre e o acesso aos dados feito inteiramente por intermédio de servidores de *tablets* apropriados, sem envolvimento do mestre nesse estágio (para garantir o máximo desempenho de saída ao acessar conjuntos de dados grandes, interagindo diretamente com os servidores de *tablets*). Em particular, as tarefas de controle associadas ao mestre são as seguintes:

- monitorar o *status* dos servidores de *tablets* e reagir à disponibilidade de novos servidores de *tablets* e à falha dos já existentes;
- atribuir *tablets* a servidores de *tablets* e garantir um balanceamento de carga eficiente;
- coleta de lixo dos arquivos armazenados no GFS.

A Bigtable vai mais longe do que o GFS no sentido de não envolver o mestre na tarefa básica de mapear *tablets* nos dados persistentes (que são, conforme mencionado anteriormente, armazenados no GFS). Isso significa que os clientes da Bigtable não precisam se comunicar com o mestre (compare isso com a operação *open* no GFS, que envolve o mestre), uma decisão de projeto que reduz significativamente a carga no mestre e a possibilidade de o mestre se tornar um gargalo.

Veremos, agora, como a Bigtable usa o GFS para armazenar seus dados e usa o Chubby de maneiras bastante inovadoras para implementar monitoramento e balanceamento de carga.

Armazenamento de dados na Bigtable: o mapeamento de tabelas da Bigtable para o GFS envolve vários estágios, conforme resumido a seguir:

- Uma tabela é decomposta em vários *tablets*, dividindo-se a tabela por linha, pegando-se um intervalo de linhas até um tamanho de cerca de 100–200 megabytes e

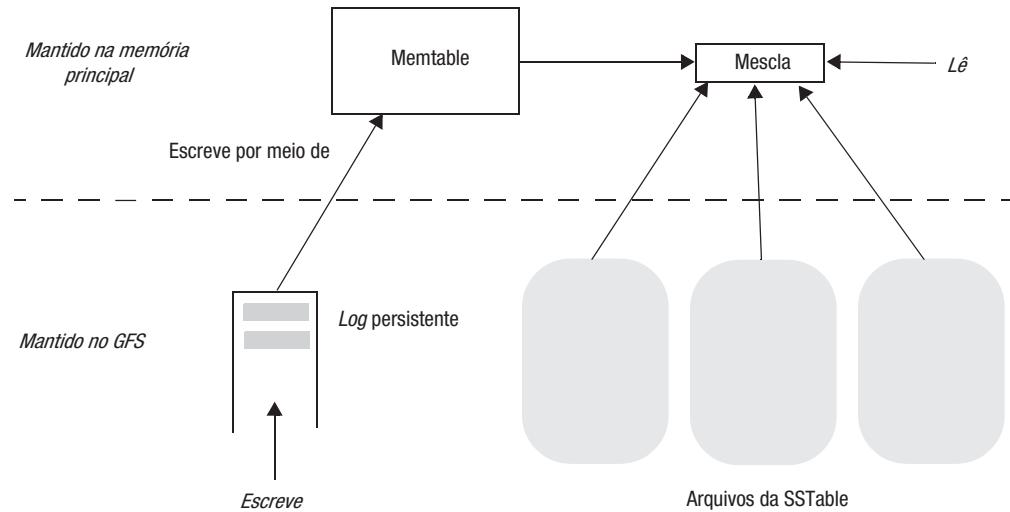


Figura 21.15 A arquitetura de armazenamento na Bigtable.

mapeando-se isso em um *tablet*. Portanto, uma tabela consistirá em vários *tablets*, dependendo de seu tamanho. À medida que as tabelas aumentarem, *tablets* extras serão adicionados.

- Cada *tablet* é representado por uma estrutura de armazenamento que consiste em um conjunto de arquivos que armazenam dados em um formato específico (a SSTable), junto a outras estruturas de armazenamento que implementam o *log*.
- O mapeamento de *tablets* para SSTables é fornecido por um esquema de índice hierárquico inspirado em árvores B⁺.

Veremos a representação do armazenamento e o mapeamento com mais detalhes a seguir.

A representação precisa do armazenamento de um *tablet* na Bigtable está mostrada na Figura 21.15. A principal unidade de armazenamento na Bigtable é a SSTable (um formato de arquivo que também é usado em outras partes da infraestrutura do Google). Uma SSTable é organizada como um mapeamento ordenado e imutável de chaves para valores, sendo ambos *strings* arbitrários. São fornecidas operações para ler eficientemente o valor associado à determinada chave e para iterar pelo conjunto de valores em determinado intervalo de chaves. O índice de uma SSTable é escrito no final do arquivo SSTable e lido na memória quando uma SSTable é acessada. Isso significa que determinada entrada pode ser lida com uma única leitura de disco. Opcionalmente, uma SSTable inteira pode ser armazenada na memória principal.

Um *tablet* é representado por diversas SSTables. Em vez de fazer mutações diretamente nas SSTables, primeiramente as escritas são confirmadas em um *log* para suportar a recuperação (veja o Capítulo 17), com o *log* também mantido no GFS. As entradas de *log* são escritas na *memtable* mantida na memória principal. Portanto, as SSTables atuam como um instantâneo do estado de um *tablet* e, em caso de falha, a recuperação é implementada pela reprodução das entradas de *log* mais recentes desde o último instantâneo. As leituras são feitas fornecendo-se uma visão mesclada dos dados das SSTables.

combinadas com a *memtable*. Diferentes níveis de compactação são realizados nessa estrutura de dados para manter a operação eficiente, conforme relatado em Chang *et al.* [2008]. Note que as SSTables também podem ser compactadas para reduzir os requisitos de armazenamento de tabelas específicas na Bigtable. Os usuários podem especificar se as tabelas devem se compactadas e também o algoritmo de compactação a ser usado.

Conforme mencionado anteriormente, o mestre não é envolvido no mapeamento de tabelas para dados armazenados. Particularmente, isso é conseguido percorrendo-se um índice baseado no conceito de árvores B⁺ (uma forma de árvore na qual todos os dados são mantidos em nós-folha, com os outros nós contendo dados de indexação e metadados).

Um cliente da Bigtable que esteja procurando o local de um *tablet* começa a pesquisa examinando um arquivo específico no Chubby que contém a localização de um *tablet-raiz* – isto é, um tablet contendo o índice-raiz da estrutura em árvore. Esse *tablet-raiz* contém metadados sobre outros *tablets*, especificamente sobre outros *tablets* de metadados que, por sua vez, contêm o local dos *tablets* dos dados reais. O *tablet-raiz*, junto a outros *tablets* de metadados, forma uma tabela de metadados, com a única distinção sendo que as entradas do *tablet-raiz* contêm metadados sobre *tablets* de metadados que, por sua vez, contêm metadados sobre os *tablets* de dados reais. Com esse esquema, a profundidade da árvore é limitada em três. As entradas da tabela de metadados mapeiam partes dos *tablets* em informações de localização, incluindo informações sobre a representação de armazenamento desse *tablet* (inclusive o conjunto de SSTables e o *log* associado).

Essa estrutura global está representada na Figura 21.16. Para reduzir essa hierarquia de três níveis, os clientes colocam as informações de localização na cache e também os metadados associados buscados previamente com outras tabelas ao acessar a estrutura de dados acima.

Monitoramento: a Bigtable usa o Chubby de maneira bastante interessante para monitorar os servidores de *tablets*. A Bigtable mantém no Chubby um diretório contendo arquivos que representam cada um dos servidores de *tablets* disponíveis. Quando um novo servidor de *tablets* aparece, ele cria um novo arquivo nesse diretório e, fundamentalmente, obtém uma trava exclusiva nesse arquivo. A existência desse arquivo atua como um sinal de que o servidor de *tablets* está totalmente operacional e pronto para receber *tablets* do mestre, com a trava fornecendo um meio de comunicação entre as duas partes:

No lado do servidor de tablets: cada servidor de *tablets* monitora sua trava exclusiva e, se ela é perdida, ele para de servir seus *tablets*. A causa mais provável disso é uma partição de rede que comprometa a sessão do Chubby. O servidor de *tablets* tentará readquirir a trava exclusiva se o arquivo ainda existir (veja a seguir) e, se o arquivo desaparecer, o servidor terminará a si mesmo. Caso um servidor termine por outro motivo, por exemplo, porque é informado de que sua máquina é necessária para outro propósito, o servidor de *tablets* pode entregar sua trava exclusiva, disparando, assim, uma reatribuição.

No lado do mestre: o mestre solicita o *status* da trava periodicamente. Se a trava é perdida ou se um servidor de *tablets* não responde, então claramente há um problema no servidor de *tablets* ou no Chubby. O mestre tenta adquirir a trava e, se tiver sucesso, pode deduzir que o Chubby está ativo e que o problema está no servidor de *tablets*. Então, o mestre exclui o arquivo do diretório, o que resultará no servidor

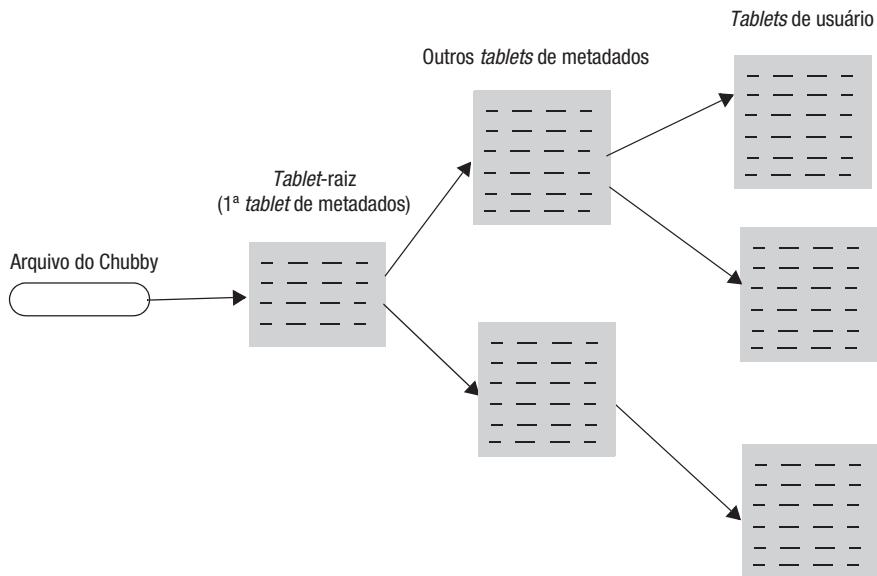


Figura 21.16 O esquema de indexação hierárquico adotado pela Bigtable.

de *tablets* terminando a si mesmo, caso não tenha falhado. Em seguida, o mestre precisa reatribuir todos os *tablets* para servidores de *tablets* alternativos.

O fundamento lógico é a reutilização do Chubby, que é um serviço bem testado e confiável, para obter o nível extra de monitoramento, em vez de fornecer um serviço de monitoramento especificamente para esse propósito.

Balanceamento de carga: para atribuir *tablets*, o mestre precisa mapear os *tablets* disponíveis no grupo para os servidores de *tablets* apropriados. A partir do algoritmo acima, o mestre tem uma lista precisa dos servidores de *tablets* que estão prontos e querem hospedar *tablets* e uma lista de todos os *tablets* associados ao grupo. O mestre também mantém as informações de mapeamento correntes, junto a uma lista de *tablets* não atribuídos (que é preenchida, por exemplo, quando um servidor de *tablets* é removido do sistema). Tendo essa visão global do sistema, o mestre garante que os *tablets* não atribuídos sejam designados para os servidores de *tablets* apropriados, com base nas respostas das requisições de carga, atualizando as informações de mapeamento correspondente mente.

Note que um mestre também tem sua própria trava exclusiva (a trava do mestre) e, se ela for perdida porque a sessão do Chubby foi comprometida, o mestre deve terminar a si mesmo (novamente, reutilizando o Chubby para implementar a funcionalidade adicional). Isso não interrompe o acesso aos dados, mas impede o prosseguimento de operações de controle. Portanto, nesse estágio a Bigtable ainda está disponível. Quando o mestre reinicia, precisa recuperar o *status* atual. Ele faz isso criando primeiramente um novo arquivo e obtendo a trava exclusiva, garantindo ser o único mestre no grupo, e, então, percorrendo o diretório para encontrar servidores de *tablets*, solicitando informações sobre atribuições de *tablet* dos servidores de *tablets* e também construindo uma lista de todos os *tablets* sob sua responsabilidade para descobrir os que não foram atribuídos. Então, o mestre prossegue com sua operação normal.

<i>Elemento</i>	<i>Decisão de projeto</i>	<i>Fundamentação lógica</i>	<i>Compromissos</i>
GFS	O uso de um trecho de tamanho grande (64 megabytes)	Conveniente para o tamanho dos arquivos no GFS; eficiente para grandes leituras e anexações sequenciais; minimiza o volume de metadados	Seria muito ineficiente para acesso aleatório a pequenas partes de arquivos
	O uso de um mestre centralizado	O mestre mantém uma visão global que informa as decisões de gerenciamento; mais simples de implementar	Ponto de falha único (atenuado pela manutenção de réplicas de registros de operação)
	Separação de fluxos de controle e dados	Acesso a arquivo de alto desempenho com mínimo envolvimento do mestre	Complica a biblioteca cliente, pois precisa lidar com o mestre e com <i>chunckservers</i>
Chubby	Modelo de consistência relaxado	Alto desempenho, explorando a semântica de operações do GFS	Os dados podem ser inconsistentes, em particular, duplicados
	Trava e abstração de arquivo combinadas	Uso variado, por exemplo, suportando eleições	Necessidade de entender e diferenciar entre diferentes facetas
	Leitura e escrita de arquivos inteiros	Muito eficiente para arquivos pequenos	Inadequado para arquivos grandes
Bigtable	Uso de cache no cliente, com consistência restrita	Semântica determinista	Sobrecarga de manter a consistência restrita
	O uso de uma abstração de tabela	Suporta dados estruturados eficientemente	Menos expressiva do que um banco de dados relacional
	O uso de um mestre centralizado	Como acima, o mestre tem uma visão global; mais simples de implementar	Ponto de falha único; possível gargalo
Coordenação	Separação de fluxos de controle e dados	Acesso a dados de alto desempenho com mínimo envolvimento do mestre	–
	Ênfase no monitoramento e no balanceamento de carga	Capacidade de suportar números muito grandes de clientes em paralelo	Sobrecarga associada à manutenção de estados globais

Figura 21.17 Resumo das escolhas de projeto relacionadas ao armazenamento de dados e à coordenação.

21.5.4 Resumo das principais escolhas de projeto

As escolhas de projeto globais relacionadas aos serviços de armazenamento de dados e coordenação estão resumidas na Figura 21.17.

A característica que mais se destaca a partir dessa análise é a escolha de projeto de fornecer três serviços separados que, individualmente, são relativamente simples e volta-

dos a determinado estilo de utilização, mas que juntos oferecem excelente cobertura para as necessidades das aplicações e serviços do Google. Isso fica mais evidente nos estilos complementares oferecidos pelo GFS e pelo Chubby, com a Bigtable fornecendo dados estruturados que ampliam os serviços oferecidos pelos dois serviços subjacentes. Essa escolha de projeto também ecoa a estratégia adotada para paradigmas de comunicação (veja a Seção 21.4.3), segundo a qual várias técnicas são oferecidas, cada uma otimizada para o estilo de aplicação planejado.

21.6 Serviços de computação distribuída

Para complementar os serviços de armazenamento e coordenação, também é importante suportar computação distribuída de alto desempenho sobre os grandes conjuntos de dados armazenados no GFS e na Bigtable. A infraestrutura do Google suporta computação distribuída por meio do serviço MapReduce e também da linguagem de nível mais alto Sawzall. Veremos o MapReduce em detalhes e depois examinaremos brevemente os principais recursos da linguagem Sawzall.

21.6.1 MapReduce

Dados os enormes conjuntos de dados em uso no Google, é um forte requisito a capacidade de fazer computação distribuída decompondo os dados em fragmentos menores e analisar tais fragmentos em paralelo, fazendo uso dos recursos computacionais oferecidos pela arquitetura física descrita na Seção 21.3.1. Tal análise inclui tarefas comuns como classificar, pesquisar e construir índices invertidos (índices que contêm um mapeamento de palavras para locais em diferentes arquivos, sendo essa a chave na implementação de funções de pesquisa). O MapReduce [Dean e Ghemawat 2008] é um modelo de programação simples para suportar o desenvolvimento dessas aplicações, ocultando do programador os detalhes subjacentes, inclusive os relacionados à paralelização da computação, do monitoramento e recuperação de falhas, do gerenciamento de dados e do balanceamento de carga na infraestrutura física subjacente.

Veremos os detalhes do modelo de programação oferecido pelo MapReduce, antes de examinarmos como o sistema é implementado.

Interface do MapReduce • O principal princípio por trás do MapReduce é o reconhecimento de que muitas computações paralelas compartilham o mesmo padrão global – ou seja:

- dividem os dados de entrada em vários trechos;
- fazem o processamento inicial nesses trechos de dados para produzir resultados intermediários;
- combinam os resultados intermediários para produzir a saída final.

A especificação do algoritmo associado pode, então, ser expressa em termos de duas funções, uma para realizar o processamento inicial e a segunda para produzir os resultados finais a partir dos valores intermediários. Então, é possível suportar vários estilos de computação, fornecendo diferentes implementações dessas duas funções. Fundamentalmente, considerando essas duas funções, o restante da funcionalidade pode ser compartilhado entre as diferentes computações, obtendo-se enormes reduções na complexidade ao se construir essas aplicações.

Mais especificamente, o MapReduce especifica uma computação distribuída em termos de duas funções, *map* e *reduce* (uma estratégia parcialmente influenciada pelo projeto de linguagens de programação funcionais, como a Lisp, que fornece funções de mesmo nome, embora na programação funcional a motivação não seja a computação paralela):

- *map* recebe como entrada um conjunto de pares chave/valor e produz como saída um conjunto de pares chave/valor intermediários.
- Então, os pares intermediários são classificados pelo valor da chave para que todos os resultados intermediários sejam ordenados pela chave intermediária. Isso é decomposto em grupos e passado para instâncias de *reduce*, que faz seu processamento para produzir uma lista de valores para cada grupo (para alguns cálculos, esse poderia ser um único valor).

Para ilustrar o funcionamento de MapReduce, vamos considerar um exemplo simples. Na Seção 21.2, ilustramos os vários aspectos de uma pesquisa na Web em busca de “livro sistemas distribuídos”. Vamos simplificar isso ainda mais, procurando apenas esse *string* completo – ou seja, uma busca pela frase “livro sistemas distribuídos” conforme ela aparece em um corpo de conteúdo grande, como o conteúdo esquadrinhado da Web. Neste exemplo, as funções *map* e *reduce* executariam as seguintes tarefas:

- Supondo que sejam fornecidos como entrada um nome de página Web e seu conteúdo, a função *map* pesquisa o conteúdo de forma linear, emitindo um par chave/valor que consiste (digamos) na frase seguida pelo nome do documento Web que a contém, sempre que encontra os *strings* “livro”, seguidos por “sistemas”, seguidos por “distribuídos” (o exemplo pode ser estendido para emitir também uma posição dentro do documento).
- A função *reduce* é trivial, neste caso, simplesmente emitindo os resultados intermediários, prontos para serem combinados em um índice completo.

A implementação de MapReduce é responsável por decompor os dados em trechos, criar várias instâncias das funções *map* e *reduce*, alocá-las e ativá-las em máquinas disponíveis na infraestrutura física, monitorar as computações para verificar quaisquer falhas e implementar as estratégias de recuperação apropriadas, despachar os resultados intermediários e garantir o melhor desempenho do sistema inteiro.

Com essa estratégia, é possível fazer economias significativas em termos de linhas de código, reutilizando a estrutura subjacente do MapReduce. Por exemplo, em 2003, o Google reimplementou o principal sistema de indexação de produção e reduziu o número de linhas de código C++ no MapReduce de 3.800 para 700 – uma redução significativa, embora em um sistema relativamente pequeno. Isso também resulta em outras vantagens importantes, incluindo tornar mais fácil a atualização de algoritmos, pois há uma clara separação de preocupações entre o que é efetivamente a lógica da aplicação e o gerenciamento da computação distribuída associada (um princípio semelhante à separação de preocupações intrínseca nos sistemas baseados em contêineres, conforme relatado na Seção 8.4). Além disso, melhorias na implementação subjacente do MapReduce beneficiam imediatamente todas as suas aplicações. A contrapartida é uma estrutura mais prescritiva, embora possa ser personalizada pela especificação de *map* e *reduce* e mesmo de outras funções, conforme ficará claro a seguir.

Para ilustrar melhor o uso de MapReduce, fornecemos na Figura 21.18 um conjunto de exemplos de funções comuns e como elas seriam implementadas com as funções

Função	Etapa inicial	Fase map	Etapa intermediária	Fase reduce
Contagem de palavras		Para cada ocorrência da palavra na partição de dados, emite <palavra, 1>		Para cada palavra no conjunto intermediário, conta o número de valores 1
Grep		Produz uma linha na saída, caso corresponda ao padrão dado		Nulo
Classificação (Isso conta pesadamente com a etapa intermediária)	Particiona os dados em trechos de tamanho fixo para processamento	Para cada entrada nos dados inseridos, produz na saída os pares chave-valor a serem classificados	Mescla/classifica todos os pares chave-valor, de acordo com sua chave intermediária	Nulo
Índice invertido		Analisa os documentos associados e produz na saída um par <palavra, ID do documento>, quando essa palavra existe		Para cada palavra, produz uma lista de IDs de documento (classificadas)

Figura 21.18 Exemplos de uso de MapReduce.

map e *reduce*. Para sermos completos, também são mostradas as etapas compartilhadas na computação realizada pela estrutura MapReduce. Mais detalhes sobre esses exemplos podem ser encontrados em Dean e Ghemawat [2004].

Arquitetura do MapReduce • MapReduce é implementado por uma biblioteca que, conforme mencionado anteriormente, oculta os detalhes associados à paralelização e à distribuição e permite ao programador se concentrar na especificação das funções *map* e *reduce*. Essa biblioteca é construída em cima de outros aspectos da infraestrutura do Google, em particular usando RPC para comunicação e GFS para o armazenamento de valores intermediários. Também é comum o MapReduce receber seus dados de entrada da Bigtable e produzir uma tabela como resultado, por exemplo, como o exemplo de Google Analytics mencionado anteriormente (Seção 21.5.3).

A execução global de um programa MapReduce está ilustrada na Figura 21.19, que mostra as principais fases envolvidas:

- O primeiro estágio é dividir o arquivo de entrada em M partes, com cada uma delas tendo normalmente um tamanho de 16-64 megabytes (portanto, não maior do que um único trecho no GFS). O tamanho real é ajustado pelo programador e, portanto, o programador é capaz de otimizar isso para o processamento paralelo particular a seguir. O espaço de chaves associado aos resultados intermediários também é partitionado em R partes, usando-se uma função de partição (programável). Assim, a computação global envolve M execuções de *map* e R execuções de *reduce*.
- Então, a biblioteca MapReduce inicia um conjunto de máquinas trabalhadoras (*workers*) do conjunto disponível no *cluster*, sendo uma designada como *mestra* e

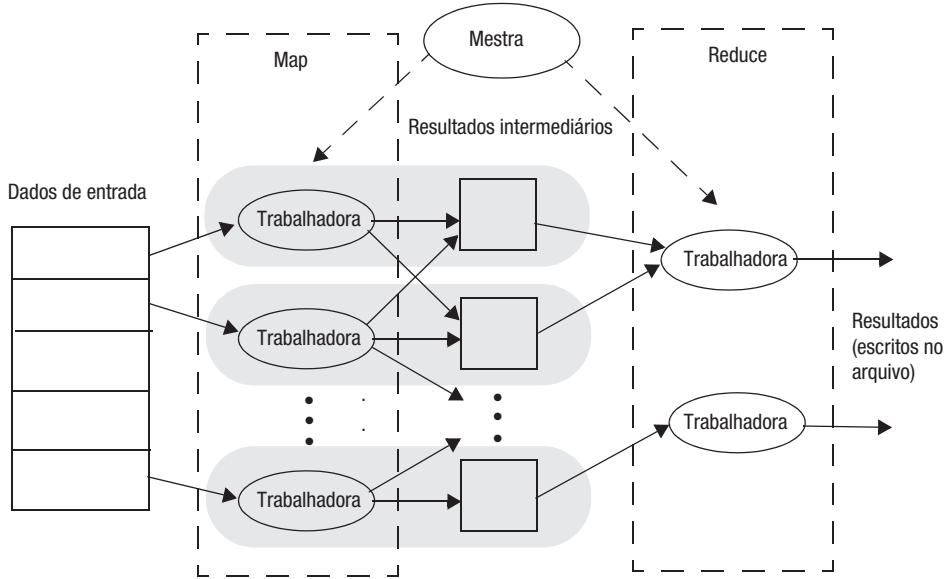


Figura 21.19 A execução global de um programa MapReduce

as outras sendo usadas para executar as etapas de *map* ou *reduce*. O número de trabalhadoras normalmente é muito menor do que $M+R$. Por exemplo, Dean e Ghemawat [2008] relatam valores típicos de $M=200.000$, $R=5.000$, com 2.000 máquinas trabalhadoras alocadas para a tarefa. O objetivo da mestra é monitorar o estado das trabalhadoras e alocar trabalhadoras ociosas para tarefas, a execução de funções *map* ou *reduce*. Mais precisamente, a mestra monitora o *status* de tarefas de *map* e *reduce* em termos de estarem *ociosas*, *em andamento* ou *concluídas* e também mantém informações sobre o local de resultados intermediários para passar uma tarefa de *reduce* às trabalhadoras alocadas.

- Uma trabalhadora que recebeu uma tarefa de *map* primeiramente lerá o conteúdo do arquivo de entrada alocado para essa tarefa, extraírá os pares chave/valor e os fornecerá como entrada para a função *map*. A saída da função *map* é um conjunto processado de pares chave/valor que são mantidos em um *buffer* intermediário. Quando os dados de entrada forem armazenados no GFS, o arquivo será replicado em (digamos) três máquinas. A mestra tenta alocar uma trabalhadora em uma dessas três máquinas para garantir a localidade e minimizar o uso de largura de banda da rede. Se isso não for possível, será selecionada uma máquina próxima aos dados.
- Os *buffers* intermediários são escritos periodicamente em um arquivo local para a computação de *map*. Nesse estágio, os dados são particionados de acordo com a função de partição, resultando em R regiões. Essa função de partição, que é fundamental para a operação de MapReduce, pode ser especificada pelo programador, mas o padrão é a execução de uma função de *hashing* na chave e, então, aplicação do módulo R no valor do *hashing* para produzir R partições, com o resultado final de que os resultados intermediários são agrupados de acordo com o valor de *hashing*. Dean e Ghemawat [2004] fornecem o exemplo alternativo no qual as

chaves são URLs e o programador quer agrupar os resultados intermediários pela máquina associada: $\text{hash}(\text{Hostname}(key)) \bmod R$. A mestra é notificada quando o particionamento está concluído e, então, é capaz de solicitar a execução das funções *reduce* associadas.

- Quando uma trabalhadora é designada para executar uma função *reduce*, ela lê sua partição correspondente no disco local das trabalhadoras de *map*, usando RPC. Esses dados são classificados pela biblioteca MapReduce prontos para processamento pela função *reduce*. Uma vez concluída a classificação, a trabalhadora de *reduce* percorre os pares chave/valor da partição, aplicando a função *reduce* para produzir um conjunto de resultado acumulado, que é então escrito em um arquivo de saída. Isso continua até que todas as chaves da partição sejam processadas.

Obtenção de tolerância a falhas: a implementação de MapReduce fornece um alto nível de tolerância a falhas, em particular garantindo que, se as operações de *map* e *reduce* forem *determinísticas* com relação às suas entradas (isto é, elas sempre produzem as mesmas saídas para determinado conjunto de entradas), então a tarefa global de MapReduce produzirá a mesma saída que a execução sequencial do programa, ainda que em face de falhas.

Para lidar com falhas, a mestra envia uma mensagem *ping* periodicamente para verificar se uma trabalhadora está ativa e executando sua operação planejada. Se nenhuma resposta é recebida, supõe-se que a trabalhadora falhou e isso é registrado pela mestra. A ação subsequente depende, então, se a tarefa que estava sendo executada era *map* ou *reduce*:

- Se a trabalhadora estava executando uma tarefa *map*, essa tarefa é marcada como *ociosa*, implicando que será reagendada. Isso acontece independentemente de a tarefa associada estar em andamento ou concluída. Lembre-se de que os resultados são armazenados em discos locais e, assim, se a máquina tiver falhado, os resultados ficarão inacessíveis.
- Se a trabalhadora estava executando uma tarefa *reduce*, essa tarefa é marcada como *ociosa* somente se ainda estava em andamento; se foi concluída, os resultados estarão disponíveis, pois são escritos no sistema de arquivos global (e replicado).

Note que, para se obter a semântica desejada, é importante que as saídas das tarefas *map* e *reduce* sejam escritas atomicamente, uma propriedade garantida pela biblioteca MapReduce em cooperação com o sistema de arquivos subjacente. Detalhes sobre como isso é conseguido podem ser encontrados em Dean e Ghemawat [2008].

MapReduce também implementa uma estratégia para lidar com trabalhadoras que podem estar demorando muito para terminar (conhecidas como *vagabundas – stragglers*). O Google tem observado que é relativamente comum algumas trabalhadoras funcionarem lentamente, por exemplo, por causa de um disco defeituoso que pode funcionar mal devido às várias etapas de correção de erro envolvidas nas transferências de dados. Para tratar disso, quando a execução de um programa está perto do fim, a mestra rotineiramente inicia trabalhadoras de *backup* para todas as tarefas *em andamento* restantes. As tarefas associadas são marcadas como *concluídas* quando a trabalhadora original, ou a nova, termina. Relata-se que isso tem um impacto significativo nos tempos de conclusão, novamente contornando o problema de se trabalhar com máquinas comuns que podem, e vão, falhar.

Conforme mencionado anteriormente, o MapReduce é projetado para operar junto à Bigtable no processamento de grandes volumes de dados estruturados. De fato, dentro do Google é comum encontrar aplicativos que usam uma mistura de todos os elementos da infraestrutura. No quadro a seguir, descrevemos o suporte fornecido para os aplicativos Google Maps e Google Earth pelo MapReduce, pela Bigtable e pelo GFS.

Suprimento a Google Maps e Google Earth

As programações clientes do Google Maps [maps.google.com] e do Google Earth [earth.google.com] contam com a disponibilidade de enormes conjuntos de *peças de mapa (image tiles)* para carregamento por parte dos clientes dos servidores Google. As peças de mapa são vetores quadrados de valores de *pixel* contendo imagens renderizadas de aspectos geográficos e são organizadas em camadas contendo diferentes tipos desses aspectos. Uma camada de base de peças mostrando um mapa de rua é construída a partir de um banco de dados geográfico atualizado e outra camada é construída a partir de imagens de satélite e aéreas em escala, mostrando as características físicas da superfície da Terra. Outras camadas, parcialmente transparentes, também são mantidas e podem ser ativadas para mostrar redes de transporte público e outros recursos de infraestrutura, contornos de elevação e até fluxos de tráfego em tempo real. Os conjuntos de peças de cada camada cobrem toda a superfície terrestre do planeta e são replicados para mostrar diferentes níveis de detalhes com até 20 níveis de zoom (isto é, de escalas).

Grande parte dos dados geográficos básicos muda apenas lentamente, mas os dados que definem ruas novas e alteradas e outra infraestrutura física se tornam disponíveis o tempo todo, e a consequente regeneração de conjuntos de peças de mapa exige a execução, nos servidores do Google, de um aplicativo distribuído de alto desempenho que converte vetores geográficos, pontos e dados de imagem brutos nas peças. A implementação faz pesado uso da Bigtable para armazenar os dados. Os dados geográficos básicos são armazenados em um formato XML conhecido como KML (Keyhole Markup Language – Keyhole é o nome da empresa que desenvolveu o software, adquirido pelo Google em 2004). Os dados vetoriais e de imagem brutos são recebidos em muitos formatos e em resoluções de diversas fontes, incluindo imagens de satélite e aéreas, e são armazenados com metadados KML em uma única tabela, na qual as linhas representam locais geográficos em particular e as colunas representam diferentes aspectos geográficos e imagens brutais, organizados em famílias de colunas. O esquema de atribuição de nomes de linhas garante que as características físicas que estão próximas sejam armazenadas em linhas adjacentes, de modo que os dados necessários para gerar cada peça de mapa ficarão em um *tablet* ou em não mais do que um pequeno número deles. O tamanho dessa tabela é de cerca de 70 terabytes e ela tem 9 bilhões de células. Ela é relativamente esparsa, pois normalmente existem poucos aspectos ou imagens por localização geográfica.

Dados são continuamente adicionados à tabela, à medida que mais dados geográficos e imagens se tornam disponíveis. Em pontos do tempo selecionados, a adição de dados é suspensa, e uma atualização de peças de mapa começa. Um conjunto de processos concorrentes de *Map* (como no MapReduce) trabalha nos dados brutos para transformar e corrigir todas as coordenadas de georreferenciamento para a apresentação plana dos dados e para combinar as imagens. Esse estágio de *Map* gera uma estrutura tabular contendo dados geográficos classificados pelo local, que são passados para um conjunto de processos concorrentes de *Reduce*, que representa as peças de mapa como imagens rasterizadas. A tarefa inteira do MapReduce leva cerca de 8 horas para gerar um conjunto completo de peças, processando dados brutos a cerca de 1 Megabyte por segundo [Chang *et al.* 2008].

As peças de mapa resultantes são armazenadas no GFS, com um índice associado armazenado em outra tabela na Bigtable. Essa tabela é pesadamente replicada por centenas de servidores de *tablets* em grupos de vários centros de dados, possibilitando, assim, atender a números muito grandes de usuários concomitantes de Google Maps e Google Earth. O índice tem em torno de 500 gigabytes de tamanho e as partes significativas são mantidas na memória principal, reduzindo a latência associada às leituras.

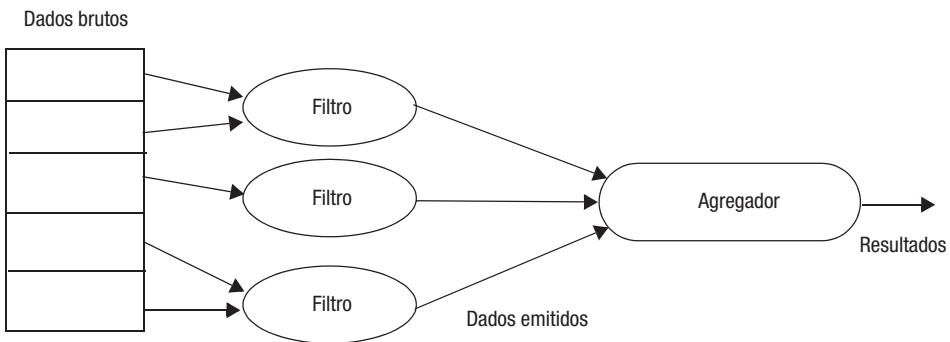


Figura 21.20 A execução global de um programa em Sawzall.

21.6.2 Sawzall

Sawzall [Pike *et al.* 2005] é uma linguagem de programação interpretada para análise de dados em paralelo sobre conjuntos de dados muito grandes em ambientes altamente distribuídos, como o fornecido pela infraestrutura física do Google. Embora o MapReduce suporte prontamente a construção de programas altamente paralelos e distribuídos, o objetivo da Sawzall é simplificar a construção desses programas. Isso é confirmado na prática, com programas em Sawzall frequentemente sendo de 10 a 20 vezes menores do que os programas equivalentes escritos para MapReduce [Pike *et al.* 2005]. A implementação da linguagem Sawzall compõe grande parte da infraestrutura existente do Google fazendo uso de MapReduce para criar e gerenciar as execuções paralelas subjacentes, do GFS para armazenar dados associados à computação e de *buffers* de protocolo para fornecer um formato de dados comum para registros armazenados.

Assim como o MapReduce, a Sawzall presume que as computações paralelas seguem determinado padrão, que resumimos na Figura 21.20. A Sawzall presume que a entrada de uma computação consiste em *dados brutos*, os quais, por sua vez, consistem em um conjunto de registros a serem processados. Então, as computações prosseguem executando *filtros*, os quais processam cada registro em paralelo, produzindo resultados *emitidos*. *Agregadores* recebem os dados emitidos e produzem os resultados globais da computação.

A Sawzall também faz duas suposições sobre a execução de filtros e agregadores:

- A execução de filtros e agregadores deve ser *comutativa* entre todos os registros; isto é, os filtros podem ser executados em qualquer ordem e o resultado será o mesmo.
- As operações de agregador devem ser *associativas*. Isto é, os parênteses (implícitos) na execução não importam, permitindo mais graus de liberdade na execução.

Como se poderia esperar a partir do exame do MapReduce, os programas em Sawzall que expressam operações de filtro e emissões de dados são executados na fase *map* do MapReduce, com os agregadores correspondendo à fase *reduce*. Um conjunto de agregadores predefinidos é fornecido pela linguagem, incluindo agregadores que fazem a soma de todos os valores emitidos (*sum*) ou constróem uma coleção de todos os valores emitidos (*collection*). Outros agregadores são mais estatísticos, construindo, por exemplo, uma distribuição de probabilidade cumulativa (*quantile*) ou estimando os valores que são

<i>Elemento</i>	<i>Decisão de projeto</i>	<i>Fundamentação lógica</i>	<i>Compromissos</i>
<i>MapReduce</i>	O uso de uma estrutura comum	Oculto do programador os detalhes da paralelização e distribuição; melhorias na infraestrutura imediatamente exploradas por todas as aplicações MapReduce	As escolhas de projeto dentro da estrutura podem não ser adequadas para todos os estilos de computação distribuída
	Programação de sistema por meio de duas operações, <i>map</i> e <i>reduce</i>	Modelo de programação muito simples, permitindo o rápido desenvolvimento de computações distribuídas complexas	Novamente, pode não ser adequado a todos os domínios de problema
	Suporte inerente para computações distribuídas tolerantes a falhas	O programador não precisa se preocupar com o tratamento de falhas (particularmente importante para tarefas longas executando em infraestrutura física, em que falhas são esperadas)	Sobrecarga associada às estratégias de recuperação de falha
<i>Sawzall</i>	Fornecimento de uma linguagem de programação especializada para computação distribuída	Novamente, suporte para o rápido desenvolvimento de computações distribuídas complexas, com a complexidade ocultada para o programador (ainda mais do que com MapReduce)	Presume que os programas podem ser escritos no estilo suportado (em termos de filtros e agregadores)

Figura 21.21 Resumo das escolhas de projeto relacionadas à computação distribuída.

mais comuns (*top*). Também é possível um programador desenvolver novos agregadores, embora isso deva ser relativamente raro.

Ilustramos o uso de Sawzall por meio de um exemplo simples de Pike *et al.* [2005], apresentando os recursos acima:

```
count: table sum of int;
total: table sum of float;
x: float = input;
emit count <- 1;
emit total <- x;
```

Esse programa recebe como entrada registros simples de tipo *float* (um fluxo de valores acessados por meio da variável local *x*). O programa também define dois agregadores, introduzidos com a palavra-chave *table*, com a palavra-chave adicional *sum* indicando que são agregadores de adição (essa palavra-chave poderia igualmente ter sido *collection*, *quantile* ou *top*, por exemplo). As chamadas de *emit* produzem um fluxo de valores que são processados pelos agregadores, gerando os resultados desejados (neste caso, uma contagem de todos os valores do fluxo de entrada, junto a uma soma de todos esses valores).

Uma descrição completa da linguagem Sawzall e mais exemplos podem ser encontrados em Pike *et al.*

21.6.3 Resumo das principais escolhas de projeto

As escolha de projeto globais relacionadas ao MapReduce e à Sawzall estão resumidas na Figura 21.21.

As vantagens globais nas duas estratégias derivam do estímulo a um estilo de computação distribuída em particular e do fornecimento de uma infraestrutura comum para permitir a implementação eficiente de sistemas desenvolvidos com esse estilo. Essa estratégia se mostrou eficaz nas aplicações e serviços do Google, incluindo o suporte para a funcionalidade de pesquisa básica, e na exigente área de suporte para aplicativos em nuvem, como o Google Earth.

Esse trabalho tem estimulado um interessante debate na comunidade de gerenciamento de dados, sobre se tais abstrações são suficientes para todas as classes de aplicação. Para ter uma ideia sobre esse debate, consulte os artigos de Dean e Ghemawat [2010] e Stonebraker *et al.* [2010] em *Communications of the ACM*.

21.7 Resumo

Este capítulo finaliza o livro, tratando do importante problema de como uma empresa de Internet muito grande tem encarado o projeto de um sistema distribuído para suportar um exigente conjunto de aplicações do mundo real. Esse é um assunto muito desafiador e exige entendimento completo das escolhas tecnológicas disponíveis para desenvolvedores de sistemas distribuídos em todos os níveis do desenvolvimento de sistemas, incluindo paradigmas de comunicação, serviços disponíveis e algoritmos distribuídos associados. Os inevitáveis compromissos associados às escolhas de projeto exigem o entendimento completo do domínio de aplicação.

A estratégia adotada neste capítulo foi destacar a arte do projeto de sistemas distribuídos por meio de um estudo de caso importante – ou seja, o exame do projeto da infraestrutura do Google, da plataforma e do *middleware* utilizados pelo Google para suportar seu mecanismo de busca e expandir o conjunto de aplicações e serviços. Esse é um estudo de caso atraente, pois trata do que é o sistema distribuído mais complexo e de grande escala já construído e que tem demonstrado satisfazer seus requisitos de projeto.

Esse estudo de caso examinou a arquitetura global do sistema, junto a estudos aprofundados dos principais serviços subjacentes – especificamente *buffers* de protocolo, o serviço de publicar-assinar, GFS, Chubby, Bigtable, MapReduce e Sawzall –, todos os quais funcionam juntos para suportar aplicações e serviços distribuídos complexos, incluindo o mecanismo de busca básico e o Google Earth. Uma lição fundamental a ser tirada desse estudo de caso é a importância de realmente entender seu domínio de aplicação, extrair um conjunto básico de princípios de projeto subjacentes e aplicá-los consistentemente. No caso do Google, isso se manifesta em uma forte defesa da simplicidade e de estratégias de baixa sobrecarga, acopladas a uma ênfase nos testes, registros e monitoramento. O resultado final é uma arquitetura altamente flexível, confiável, de alto desempenho e aberta em termos de suporte para novas aplicações e serviços.

A infraestrutura do Google é uma de várias soluções de *middleware* para computação em nuvem surgida recentemente (embora disponível completamente apenas dentro do Google). Outras soluções incluem o AWS (Amazon Web Services) [aws.amazon.com], Azure da Microsoft [www.microsoft.com/IV] e soluções de código-fonte aberto, incluindo Hadoop (que contém uma implementação de MapReduce) [hadoop.apache.org], Eu-

calyptus [open.eucalyptus.com], o Google App Engine (disponível externamente e fornecendo uma janela para parte da funcionalidade oferecida pela infraestrutura do Google) [code.google.com IV] e Sector/Sphere [sector.sourceforge.net]. Também foi desenvolvido o OpenStreetMap [www.openstreetmap.org], uma alternativa aberta ao Google Maps que funciona de maneira semelhante, usando *software* desenvolvido por voluntários e servidores não comerciais. Detalhes dessas implementações estão geralmente disponíveis, e estimulamos o leitor a estudar uma seleção dessas arquiteturas, comparando as escolhas de projeto com as apresentadas no estudo de caso anterior.

Fora isso, há uma verdadeira escassez de estudos de caso publicados relacionados ao projeto de sistemas distribuídos, e isso é uma pena, dado o valor educativo em potencial de estudar arquiteturas de sistemas distribuídos globais e seus princípios de projeto associados. Portanto, a principal contribuição deste capítulo foi fornecer um primeiro estudo de caso aprofundado ilustrando as complexidades do projeto e implementação de uma solução de sistema distribuído completo.

Exercícios

- 21.1 Até que ponto o Google agora é uma empresa provedor de nuvem? Consulte a definição do Capítulo 1 e repetida na Seção 21.2. *Capítulo 1, página 921*
- 21.2 Os principais requisitos da infraestrutura do Google são escalabilidade, confiabilidade, desempenho e abertura. Dê três exemplos de onde esses requisitos poderiam estar em conflito e discuta como o Google lida com esses conflitos em potencial. *página 924*
- 21.3 Uma especificação de uma estrutura *Person* em XML foi apresentada no Capítulo 4 (Figura 4.12) Reescreva essa especificação usando *buffers* de protocolo. *Capítulo 4, página 929*
- 21.4 Discuta até que ponto o estilo RPC suportado pelo componente *buffers* de protocolo melhora a capacidade de extensão (especialmente a decisão de projeto de ter apenas um argumento e um resultado). *página 931*
- 21.5 Explique por que a infraestrutura do Google suporta três recursos de armazenamento de dados distintos. Por que o Google não adota apenas um banco de dados distribuído comercial, em vez dos três serviços separados? *página 935*
- 21.6 O GFS e a Bigtable fazem a mesma escolha básica de projeto – ter um único mestre. Quais as repercussões de uma falha desse mestre único em cada caso? *páginas 937-938, 950-952*
- 21.7 Na Seção 21.5.2, comparamos a estratégia de consistência de cache do Chubby com o NFS, concluindo que o NFS oferece semântica muito mais fraca, em termos de ver diferentes versões de arquivos em diferentes nós. Faça uma comparação semelhante entre as estratégias de consistência de cache adotadas no Chubby e no AFS. *Seção 12.1.3, páginas 943-944*
- 21.8 Conforme descrito na Seção 21.5.2, a implementação do Paxos depende da geração de números sequenciais cada vez maiores e globalmente exclusivos. Essa seção também descreveu uma possível implementação. Descreva uma estratégia alternativa para implementar esses números sequenciais. *páginas 945-946*
- 21.9 A Figura 21.18 lista várias aplicações possíveis de MapReduce. Descreva outra aplicação possível e esboce como isso seria implementado no MapReduce, fornecendo, em particular, esboços de implementações das funções *map* e *reduce*. *página 958*
- 21.10 Dê um exemplo de computação distribuída que seria difícil implementar no MapReduce, fornecendo as razões para sua resposta. *página 958*

Referências

Referências online

Esta lista de referências está disponível na Web, no endereço www.cdk5.net/refs. Ela fornece *links* em que se pode clicar para ver documentos que existem somente na Web. Nesta lista impressa, os itens identificados por caracteres sublinhados, por exemplo, www.omg.org, apontam para uma página de índices que levam ao documento; *links* diretos podem ser encontrados na lista de referências online.

As referências para RFCs dizem respeito à série de padrões e especificações da Internet, chamados “requests for comments”, que estão disponíveis na Internet Engineering Task Force, no endereço www.ietf.org/rfc/ e em vários outros sites conhecidos.

A lista de referências online também pode ser usada como auxílio na busca de cópias de outros documentos na web, pesquisando-se por autores ou títulos com o Google ou com o Citeseer, no endereço citeseer.ist.psu.edu.

O material online escrito ou editado pelos autores para complementar o livro está referenciado pela identificação www.cdk5.net, mas não foi incluído na lista. Por exemplo, www.cdk5.net/ipc se refere a material adicional sobre comunicação entre processos em nossas páginas web.

- | | |
|-------------------------------|---|
| Abadi and Gordon 1999 | Abadi, M. and Gordon, A.D. (1999). A calculus for cryptographic protocols: The spi calculus. <i>Information and Computation</i> , Vol. 148, No. 1, pp. 1–70. |
| Abadi <i>et al.</i> 1998 | Abadi, M., Birrell, A.D., Slata, R. and Wobber, E.P. (1998). Secure Web tunneling. In <i>Proceedings of the 7th International World Wide Web Conference</i> , pp. 531–9. Elsevier, in <i>Computer Networks and ISDN Systems</i> , Volume 30, Nos 1–7. |
| Abrossimov <i>et al.</i> 1989 | Abrossimov, V., Rozier, M. and Shapiro, M. (1989). Generic virtual memory management for operating system kernels. <i>Proceedings of 12th ACM Symposium on Operating System Principles</i> , December, pp. 123–36. |
| Accetta <i>e0t al.</i> 1986 | Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A. and Young, M. (1986). Mach: A new kernel foundation for UNIX development. In <i>Proceedings of the Summer 1986 USENIX Conference</i> , pp. 93–112. |

- Adjie-Winoto *et al.* 1999 Adjie-Winoto, W., Schwartz, E., Balakrishnan, H. and Lilley, J. (1999). The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, published as *Operating Systems Review*, Vol. 34, No. 5, pp. 186–201.
- Agrawal *et al.* 1987 Agrawal, D., Bernstein, A., Gupta, P. and Sengupta, S. (1987). Distributed optimistic concurrency control with reduced rollback. *Distributed Computing*, Vol. 2, No 1, pp. 45–59.
- Ahamad *et al.* 1992 Ahamad, M., Bazzi, R., John, R., Kohli, P. and Neiger, G. (1992). *The Power of Processor Consistency*. Technical report GIT-CC-92/34, Georgia Institute of Technology, Atlanta, GA.
- Al-Muhtadi *et al.* 2002 Al-Muhtadi, J., Campbell, R., Kapadia, A., Mickunas, D. and Yi, S. (2002). Routing through the mist: Privacy preserving communication in ubiquitous computing environments. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, July, pp. 74–83.
- Albanna *et al.* 2001 Albanna, Z., Almeroth, K., Meyer, D. and Schipper, M. (2001). *IANA Guidelines for IPv4 Multicast Address Assignments*. Internet RFC 3171.
- Alonso *et al.* 2004 Alonso, G., Casata, C., Kuno, H. and Machiraju, V. (2004). *Web Services, Concepts, Architectures and Applications*. Berlin, Heidelberg: Springer-Verlag.
- Anderson 1993 Anderson, D.P. (1993). Metascheduling for continuous media. *ACM Transactions on Computer Systems*, Vol. 11, No. 3, pp. 226–52.
- Anderson 1996 Anderson, R. J. (1996). The Eternity Service. In *Proceedings of Pragocrypt '96*, pp. 242–52.
- Anderson 2008 Anderson, R.J. (2008). *Security Engineering*, 2nd edn. John Wiley & Sons.
- Anderson *et al.* 1990 Anderson, D.P., Herrtwich, R.G. and Schaefer, C. (1990). *SRP – A Resource Reservation Protocol for Guaranteed-Performance Communication in the Internet*. Technical report 90-006, International Computer Science Institute, Berkeley, CA.
- Anderson *et. al.* 1991 Anderson, T., Bershad, B., Lazowska, E. and Levy, H. (1991). Scheduler activations: Efficient kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pp. 95–109.
- Anderson *et al.* 1995 Anderson, T., Culler, D., Patterson, D. and the NOW team. (1995). A case for NOW (Networks Of Workstations). *IEEE Micro*, Vol. 15, No. 1, pp. 54–64.

- Anderson *et al.* 1996 Anderson, T.E., Dahlin, M.D., Neefe, J.M., Patterson, D.A., Roselli, D.S. and Wang, R.Y. (1996). Serverless Network File Systems. *ACM Trans. on Computer Systems*, Vol. 14, No. 1, pp. 41–79.
- Anderson *et al.* 2002 Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M. and Werthimer, D. (2002). SETI@home: An experiment in public-resource computing. *Communications of the ACM*, Vol. 45, No. 11, pp. 56–61.
- Anderson *et al.* 2004 Anderson, R., Chan, H. and Perrig, A. (2004). Key infection: Smart trust for smart dust. In *Proceedings of the IEEE 12th International Conference on Network Protocols* (ICNP 2004), Berlin, Germany, October, pp. 206–215.
- ANSA 1989 ANSA (1989). *The Advanced Network Systems Architecture (ANSA) Reference Manual*. Castle Hill, Cambridge, England: Architecture Project Management.
- ANSI 1985 American National Standards Institute (1985). *American National Standard for Financial Institution Key Management*. Standard X9.17 (revised).
- Armand *et al.* 1989 Armand, F., Gien, M., Herrman, F. and Rozier, M. (1989). Distributing UNIX brings it back to its original virtues. In *Proc. Workshop on Experiences with Building Distributed and Multiprocessor Systems*, October, pp. 153–174.
- Arnold *et al.* 1999 Arnold, K., O’Sullivan, B., Scheifler, R.W., Waldo, J. and Wollrath, A. (1999). *The Jini Specification*. Reading, MA: Addison-Wesley.
- associates.amazon.com Amazon Web Service FAQs.
- Attiya and Welch 1998 Attiya, H. and Welch, J. (1998). *Distributed Computing – Fundamentals, Simulations and Advanced Topics*. Maidenhead, England: McGraw-Hill.
- aws.amazon.com Amazon Web Services. *Home page*.
- Babaoglu *et al.* 1998 Babaoglu, O., Davoli, R., Montresor, A. and Segala, R. (1998). System support for partition-aware network applications. In *Proceedings of the 18th International Conference on Distributed Computing Systems* (ICDCS ‘98), pp. 184–191.
- Bacon 2002 Bacon, J. (2002). *Concurrent Systems*, 3rd edn. Harlow, England: Addison-Wesley.
- Baker 1997 Baker, S. (1997). *CORBA Distributed Objects Using Orbix*. Harlow, England: Addison-Wesley.
- Bakken and Schlichting 1995 Bakken, D.E. and Schlichting, R.D. (1995). Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions Parallel and Distributed Systems*, Vol. 6, No. 3, pp. 287–302.
- Balakrishnan *et al.* 1995 Balakrishnan, H., Seshan, S. and Katz, R.H. (1995). Improving reliable transport and hand-off performance in cellular wireless networks. In *Proceedings of the ACM Mobile Computing and Networking Conference*, pp. 2–11.

- Balakrishnan *et al.* 1996 Balakrishnan, H., Padmanabhan, V., Seshan, S. and Katz, R. (1996). A comparison of mechanisms for improving TCP performance over wireless links. In *Proceedings of the ACM SIGCOMM '96 Conference*, pp. 256–69.
- Balan *et al.* 2003 Balan, R.K., Satyanarayanan, M., Park, S., Okoshi, T. (2003). Tactics-based remote execution for mobile computing. In *Proceedings of the First USENIX International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, San Francisco, CA, May, pp. 273–286.
- Balazinska *et al.* 2002 Balazinska, M., Balakrishnan, H. and Karger, D. (2002). INS/Twine: A scalable peer-to-peer architecture for intentional resource discovery. In *Proceedings of the International Conference on Pervasive Computing*, Zurich, Switzerland, August, pp. 195–210.
- Baldoni *et al.* 2005 Baldoni, R., Beraldì, R., Cugola, G., Migliavacca, M. and Querzoni, L. (2005). Structure-less content-based routing in mobile ad hoc networks. In *Proceedings of the International Conference on Pervasive Services*, pp. 37–46.
- Baldoni and Virgillito 2005 Baldoni, R. and Virgillito, A. (2005). *Distributed event routing in publish/subscribe communication systems: A survey*. Technical Report 15-05, Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”.
- Baldoni *et al.* 2007 Baldoni, R., Beraldì, R., Quema, V., Querzoni, L. and Tucci-Piergiovanni, S. (2007). TERA: Topic-based event routing for peer-to-peer architectures. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems*. Toronto, Ontario, Canada, June, pp. 2–13.
- Balfanz *et al.* 2002 Balfanz, D., Smetters, D.K., Stewart, P. and Wong, H.C. (2002). Talking to strangers: Authentication in ad-hoc wireless networks. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February.
- Banerjea and Mah 1991 Banerjea, A. and Mah, B.A. (1991). The real-time channel administration protocol. *Second International Workshop on Network and Operating System Support for Digital Audio and Video*, Heidelberg, Germany.
- Baran 1964 Baran, P. (1964). *On Distributed Communications*. Research Memorandum RM-3420-PR, Rand Corporation.
- Barborak *et al.* 1993 Barborak, M., Malek, M. and Dahbura, A. (1993). The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, Vol. 25, No. 2, pp. 171–220.
- Barghouti and Kaiser 1991 Barghouti, N.S. and Kaiser, G.E. (1991). Concurrency control in advanced database applications. *ACM Computing Surveys*, Vol. 23, No. 3, pp. 269–318.

- Barham *et al.* 2003a Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A. (2003). Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, October, pp. 164–177.
- Barham *et al.* 2003b Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Kotsovinos, E., Madhavapeddy, A.V.S., Neugebauer, R., Pratt, I. and Warfield, A. (2003). *Xen 2002*. Technical Report UCAM-CL-TR-553, Computing Laboratory, University of Cambridge.
- Barr and Asanovic 2003 Barr, K. and Asanovic, K. (2003). Energy aware lossless data compression. In *Proceedings of the First USENIX International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, San Francisco, CA, May, pp. 231–244.
- Barton *et al.* 2002 Barton, J., Kindberg, T. and Sadalgi, S. (2002). Physical registration: Configuring electronic directories using handheld devices. *IEEE Wireless Communications*, Vol. 9, No. 1, pp. 30–38.
- Baset and Schulzrinne 2006 Baset, S.A. and Schulzrinne, H.G. (2006). An analysis of the Skype peer-to-peer Internet telephony protocol. In *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM'06)*, pp. 1–11.
- Bates *et al.* 1996 Bates, J., Bacon, J., Moody, K. and Spiteri, M. (1996). Using events for the scalable federation of heterogeneous components. *European SIGOPS Workshop*.
- Baude *et al.* 2009 Baude, F., Caromel, D., Dalmasso, C., Danelutto, M., Getov, V., Henrio, L. and Pérez, C. (2009). GCM: A grid extension for Fractal autonomous distributed components. *Annals of Telecommunications*, Springer, Vol. 64, No. 1, pp. 5–24.
- Bell and LaPadula 1975 Bell, D.E. and LaPadula, L.J. (1975). *Computer Security Model: Unified Exposition and Multics Interpretation*. Mitre Corporation.
- Bellman 1957 Bellman, R.E. (1957). *Dynamic Programming*. Princeton, NJ: Princeton University Press.
- Bellovin and Merritt 1990 Bellovin, S.M. and Merritt, M. (1990). Limitations of the Kerberos authentication system. *ACM Computer Communications Review*, Vol. 20, No. 5, pp. 119–32.
- Bellwood *et al.* 2003 Bellwood, T., Clément, L. and von Riegen, C. (eds.) (2003). *UDDI Version 3.0.1*. Oasis Corporation.
- Beresford and Stajano 2003 Beresford, A. and Stajano, F. (2003). Location privacy in pervasive computing. *IEEE Pervasive Computing*, Vol. 2, No. 1, pp. 46–55.
- Berners-Lee 1991 Berners-Lee, T. (1991). World Wide Web Seminar.
- Berners-Lee 1999 Berners-Lee, T. (1999). *Weaving the Web*. New York: HarperCollins.

- Berners-Lee *et al.* 2005 Berners Lee, T., Fielding, R. and Masinter, L. (2005). Uniform Resource Identifiers (URI): Generic syntax. Internet RFC 3986.
- Bernstein *et al.* 1980 Bernstein, P.A., Shipman, D.W. and Rothnie, J.B. (1980). Concurrency control in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, Vol. 5, No. 1, pp. 18–51.
- Bernstein *et al.* 1987 Bernstein, P., Hadzilacos, V. and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley. Texto disponível online.
- Bershad *et al.* 1990 Bershad, B., Anderson, T., Lazowska, E. and Levy, H. (1990). Lightweight remote procedure call. *ACM Transactions on Computer Systems*, Vol. 8, No. 1, pp. 37–55.
- Bershad *et al.* 1991 Bershad, B., Anderson, T., Lazowska, E. and Levy, H. (1991). User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, Vol. 9, No. 2, pp. 175–198.
- Bershad *et al.* 1995 Bershad, B., Savage, S., Pardyak, P., Sirer, E., Fiuczynski, M., Becker, D., Chambers, C. and Eggers, S. (1995). Safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 267–84.
- Bessani *et al.* 2008 Bessani, A. N., Alchieri, E. P., Correia, M. and Fraga, J. S. (2008). DepSpace: A Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM Sigops/Eurosys European Conference on Computer Systems*. Glasgow, Scotland, April, pp. 163–176.
- Bhagwan *et al.* 2003 Bhagwan, R., Savage, S. and Voelker, G. (2003). Understanding availability, In *Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, February.
- Bhatti and Friedrich 1999 Bhatti, N. and Friedrich, R. (1999). *Web Server Support for Tiered Services*. Hewlett-Packard Corporation Technical Report HPL-1999-160.
- Birman 1993 Birman, K.P. (1993). The process group approach to reliable distributed computing. *Comms. ACM*, Vol. 36, No. 12, pp. 36–53.
- Birman 2004 Birman, K.P. (2004). Like it or not, web services are distributed Objects! *Comms. ACM*. Vol. 47, No. 12, pp. 60–62.
- Birman 2005 Birman, K.P. (2005). *Reliable Distributed Systems: Technologies, Web Services and Applications*. Springer-Verlag.
- Birman and Joseph 1987a Birman, K.P. and Joseph, T.A. (1987). Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, Vol. 5, No. 1, pp. 47–76.

- Birman and Joseph 1987b Birman, K. and Joseph, T. (1987): Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pp. 123–38.
- Birman *et al.* 1991 Birman, K.P., Schiper, A. and Stephenson, P. (1991). Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, Vol. 9, No. 3, pp. 272–314.
- Birrell and Needham 1980 Birrell, A.D. and Needham, R.M. (1980). A universal file server. *IEEE Transactions Software Engineering*, Vol. SE-6, No. 5, pp. 450–3.
- Birrell and Nelson 1984 Birrell, A.D. and Nelson, B.J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39–59.
- Birrell *et al.* 1982 Birrell, A.D., Levin, R., Needham, R.M. and Schroeder, M.D. (1982). Grapevine: An exercise in distributed computing. *Comms. ACM*, Vol. 25, No. 4, pp. 260–73.
- Birrell *et al.* 1995 Birrell, A., Nelson, G. and Owicki, S. (1993). Network objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 217–30.
- Bisiani and Forin 1988 Bisiani, R. and Forin, A. (1988). Multilanguage parallel programming of heterogeneous machines. *IEEE Transactions Computers*, Vol. 37, No. 8, pp. 930–45.
- Black 1990 Black, D. (1990). Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, Vol. 23, No. 5, pp. 35–43.
- Blakley 1999 Blakley, R. (1999). *CORBA Security – An Introduction to Safe Computing with Objects*. Reading, MA: Addison-Wesley.
- Bloch 2006 Bloch, J. (2006). How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. (OOPSLA'06), Portland, Oregon, pp. 506–507.
- Bolosky *et al.* 1996 Bolosky, W., Barrera, J., Draves, R., Fitzgerald, R., Gibson, G., Jones, M., Levi, S., Myhrvold, N. and Rashid, R. (1996). The Tiger video fileserver. *6th NOSSDAV Conference*, Zushi, Japan, April.
- Bolosky *et al.* 1997 Bolosky, W., Fitzgerald, R. and Douceur, J. (1997). Distributed schedule management in the Tiger video fileserver. In *Proc. of the 16th ACM Symposium on Operating System Principles*, St Malo, France, October, pp. 212–23.
- Bolosky *et al.* 2000 Bolosky, W.J., Douceur, J.R., Ely, D. and Theimer, M. (2000). Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pp. 34–43.

- Bonnaire *et al.* 1995 Bonnaire, X., Baggio, A. and Prun, D. (1995). Intrusion free monitoring: An observation engine for message server based applications. In *Proceedings of the 10th International Symposium on Computer and Information Sciences* (ISCIS X), pp. 541-48.
- Booch *et al.* 2005 Booch, G., Rumbaugh, J. and Jacobson, I. (2005). *The Unified Modeling Language User Guide*, 2nd edn. Reading MA: Addison-Wesley.
- Borisov *et al.* 2001 Borisov, N., Goldberg, I. and Wagner, D. (2001). Intercepting mobile communications: The insecurity of 802.11. In *Proceedings of MOBICOM 2001*, pp. 180-9.
- Bowman *et al.* 1990 Bowman, M., Peterson, L. and Yeatts, A. (1990). Univers: An attribute-based name server. *Software—Practice and Experience*, Vol. 20, No. 4, pp. 403-24.
- Box 1998 Box, D. (1998). *Essential COM*. Reading, MA: Addison-Wesley.
- Box and Curbera 2004 Box, D. and Curbera, F. (2004). *Web Services Addressing (WS-Addressing)*, BEA Systems, IBM and Microsoft, August.
- boxee.tv Boxee. *Home page*.
- Boykin *et al.* 1993 Boykin, J., Kirschen, D., Langerman, A. and LoVerso, S. (1993). *Programming under Mach*. Reading, MA: Addison-Wesley.
- Bray and Sturman 2002 Bray, J. and Sturman, C.F. (2002). *Bluetooth: Connect Without Cables*, 2nd edn. Upper Saddle River, NJ: Prentice-Hall.
- Brin and Page 1998 Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, Vol. 30, Issue 1-7, pp. 107-117.
- Bruneton *et al.* 2006 Bruneton, E., Coupaye, T., LeClercq, M., Quema, V. and Stefani, J.B. (2006). The Fractal component model and its support in Java. *Software – Practice and Experience*, Vol. 36, Nos.1 1-21, pp. 1257-1284.
- Buford 1994 Buford, J.K. (1994). *Multimedia Systems*. Addison-Wesley.
- Burns and Wellings 1998 Burns, A. and Wellings, A. (1998). *Concurrency in Ada*. England: Cambridge University Press.
- Burrows *et al.* 1989 Burrows, M., Abadi, M. and Needham, R. (1989). A logic of authentication. Technical Report 39. Palo Alto, CA: Digital Equipment Corporation Systems Research Center.
- Burrows *et al.* 1990 Burrows, M., Abadi, M. and Needham, R. (1990). A logic of authentication. *ACM Transactions on Computer Systems*, Vol. 8, No. 1, pp. 18-36.
- Burrows 2006 Burrows, M. (2006). The Chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, pp. 335-350.

- Bush 1945 Bush, V. (1945). As we may think. *The Atlantic Monthly*, July.
- Bushmann *et al.* 2007 Bushmann, F., Henney, K. and Schmidt, D.C. (2007). *Pattern-Oriented Software Architecture: A Pattern for Distributed Computing*, New York: John Wiley & Sons.
- Busi *et al.* 2003 Busi, N., Manfredini, C., Montresor, A. and Zavattaro, G. (2003). PeerSpaces: Data-driven coordination in peer-to-peer networks. In *Proceedings of the 2003 ACM Symposium on Applied Computing*, Melbourne, Florida, March, pp. 380–386.
- Callaghan 1996a Callaghan, B. (1996). *WebNFS Client Specification*. Internet RFC 2054.
- Callaghan 1996b Callaghan, B. (1996). *WebNFS Server Specification*. Internet RFC 2055.
- Callaghan 1999 Callaghan, B. (1999). *NFS Illustrated*. Reading, MA: Addison-Wesley.
- Callaghan *et al.* 1995 Callaghan, B., Pawlowski, B. and Staubach, P. (1995). *NFS Version 3 Protocol Specification*. Internet RFC 1813, Sun Microsystems.
- Campbell 1997 Campbell, R. (1997). *Managing AFS: The Andrew File System*. Upper Saddle River, NJ: Prentice-Hall.
- Campbell *et al.* 1993 Campbell, R., Islam, N., Raila, D. and Madany, P. (1993). Designing and implementing Choices: An object-oriented system in C++. *Comms. ACM*, Vol. 36, No. 9, pp. 117–26.
- Canetti and Rabin 1993 Canetti, R. and Rabin, T. (1993). Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pp. 42–51.
- Cao and Singh 2005 Cao, F. and Singh, J. P. (2005). MEDYM: match-early with dynamic multicast for content-based publish-subscribe networks. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, Grenoble, France, November, pp. 292–313.
- Carriero *et al.* 1995 Carriero, N., Gelernter, D. and Zuck, L. (1995). Bauhaus Linda. In *LNCS 924: Object-based models and languages for concurrent systems*. Berlin, Heidelberg: Springer-Verlag, pp. 66–76.
- Carter *et al.* 1991 Carter, J.B., Bennett, J.K. and Zwaenepoel, W. (1991). Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pp. 152–64.
- Carter *et al.* 1998 Carter, J., Ranganathan, A. and Susarla, S. (1998). Khazana, an infrastructure for building distributed services. In *Proceedings of ICDCS '98*. Amsterdam, The Netherland, pp. 562–71.

- Carzaniga *et al.* 2001 Carzaniga, A., Rosenblum, D. S. and Wolf, A. L. (2001). Design and evaluation of a wide-area event notification service. *ACM Trans. on Computer Systems*, Vol. 19, No. 3, pp. 332–383.
- Castro and Liskov 2000 Castro, M. and Liskov, B. (2000). Proactive recovery in a Byzantine-fault-tolerant system. *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, CA, October, p. 19.
- Castro *et al.* 2002a Castro, M., Druschel, P., Hu, Y.C. and Rowstron, A. (2002). Topology-aware routing in structured peer-to-peer overlay networks. *Technical Report MSR-TR-2002-82*, Microsoft Research, 2002.
- Castro *et al.* 2002b Castro, M., Druschel, P., Kermarrec, and Rowstron, A. (2002). SCRIBE: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, Vol. 20, No. 8, pp. 1489–99.
- Castro *et al.* 2003 Castro, M., Costa, M. and Rowsron, A. (2003). *Performance and dependability of structured peer-to-peer overlays*. Technical Report MSR-TR-2003-94, Microsoft Research, 2003.
- CCITT 1988a CCTTT (1988). *Recommendation X.500: The Directory – Overview of Concepts, Models and Service*. International Telecommunications Union, Place des Nations, 1211 Geneva, Switzerland.
- CCITT 1988b CCITT (1988). *Recommendation X.509: The Directory – Authentication Framework*. International Telecommunications Union, Place des Nations, 1211 Geneva, Switzerland.
- CCITT 1990 CCITT (1990). *Recommendation I.150: B-ISDN ATM Functional Characteristics*. International Telecommunications Union, Place des Nations, 1211 Geneva, Switzerland.
- Ceri and Owicki 1982 Ceri, S. and Owicki, S. (1982). On the use of optimistic methods for concurrency control in distributed databases. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, Berkeley, CA, pp. 117–30.
- Ceri and Pelagatti 1985 Ceri, S. and Pelagatti, G. (1985). *Distributed Databases – Principles and Systems*. Maidenhead, England: McGraw-Hill.
- Chalmers *et al.* 2004 Chalmers, D., Dulay, N. and Sloman, M. (2004). Meta data to support context aware mobile applications. In *Proceedings of the IEEE Intl. Conference on Mobile Data Management (MDM 2004)*, Berkeley, CA, January, pp. 199–210.
- Chandra and Toueg 1996 Chandra, T. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, Vol 43, No. 2, pp. 225–67.

- Chandra *et al.* 2007 Chandra, T. D., Griesemer, R. and Redstone, J. (2007). Paxos made live: An engineering perspective. In *Proc. of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, (PODC'07), Portland, Oregon, pp. 398–407.
- Chandy and Lamport 1985 Chandy, K. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, Vol. 3, No. 1, pp. 63–75.
- Chang and Maxemchuk 1984 Chang, J. and Maxemchuk, N. (1984). Reliable broadcast protocols. *ACM Transactions on Computer Systems*, Vol. 2, No. 3, pp. 251–75.
- Chang and Roberts 1979 Chang, E.G. and Roberts, R. (1979). An improved algorithm for decentralized extrema-finding in circular configurations of processors. *Comms. ACM*, Vol. 22, No. 5, pp. 281–3.
- Chang *et al.* 2008 Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A. and Gruber, R. (2008). Bigtable: A distributed storage system for structured data. *ACM Trans. on Computer Systems* Vol. 26, No. 2, pp. 1–26.
- Charron-Bost 1991 Charron-Bost, B. (1991). Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, Vol. 39, No. 1, pp. 11–16.
- Chaum 1981 Chaum, D. (1981). Untraceable electronic mail, return addresses and digital pseudonyms. *Comms. ACM*, Vol. 24, No. 2, pp. 84–88.
- Chen *et al.* 1994 Chen, P., Lee, E., Gibson, G., Katz, R. and Patterson, D. (1994). RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, Vol. 26, No. 2, pp. 145–188.
- Cheng 1998 Cheng, C.K. (1998). *A survey of media servers*. Hong Kong University CSIS, November.
- Cheng *et al.* 2005 Cheng, Y-C., Chawathe, Y., LaMarca, A. and Krumm J. (2005) Accuracy characterization for metropolitan-scale Wi-Fi localization, *Third International Conference on Mobile Systems, Applications, and Services (MobiSys 2005)*, June.
- Cheriton 1984 Cheriton, D.R. (1984). The V kernel: A software base for distributed systems. *IEEE Software*, Vol. 1, No. 2, pp. 19–42.
- Cheriton 1986 Cheriton, D.R. (1986). VMTP: A protocol for the next generation of communication systems. In *Proceedings of the SIGCOMM '86 Symposium on Communication Architectures and Protocols*, pp. 406–15.
- Cheriton and Mann 1989 Cheriton, D. and Mann, T. (1989). Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems*, Vol. 7, No. 2, pp. 147–83.

- Cheriton and Skeen 1993
Cheriton, D. and Skeen, D. (1993). Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, Dec., pp. 44–57.
- Cheriton and Zwaenepoel 1985
Cheriton, D.R. and Zwaenepoel, W. (1985). Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, Vol. 3, No. 2, pp. 77–107.
- Cheswick and Bellovin 1994
Cheswick, E.R. and Bellovin, S.M. (1994). *Firewalls and Internet Security*. Reading, MA: Addison-Wesley.
- Chien 2004
Chien, A. (2004). Massively distributed computing: Virtual screening on a desktop Grid. In Foster, I. and Kesselman, C. (eds.), *The Grid 2*. San Francisco, CA: Morgan Kauffman.
- Chisnall 2007
Chisnall, D. (2007). *The Definitive Guide to the Xen Hypervisor*. Upper Saddle River, NJ: Prentice-Hall.
- Choudhary *et al.* 1989
Choudhary, A., Kohler, W., Stankovic, J. and Towsley, D. (1989). A modified priority based probe algorithm for distributed deadlock detection and resolution. *IEEE Transactions Software Engineering*, Vol. 15, No. 1, pp. 10–17.
- Chu *et al.* 2000
Chu, Y.-H., Rao, S.G. and Zhang, H. (2000). A case for end system multicast. In *Proc. of ACM Sigmetrics*, June, pp. 1–12.
- Cilia *et al.* 2004
Cilia, M., Antollini, M., Bornhövd, C. and Buchmann, A. (2004). Dealing with heterogeneous data in pub/sub systems: The concept-based approach. In *Proceedings of the International Workshop on Distributed Event-Based Systems*, Edinburgh, Scotland, May, pp. 26–31.
- Clark 1982
Clark, D.D. (1982). *Window and Acknowledgement Strategy in TCP*. Internet RFC 813.
- Clark 1988
Clark, D.D. (1988). The design philosophy of the DARPA Internet protocols. *ACM SIGCOMM Computer Communication Review*, Vol. 18, No. 4, pp. 106–114.
- Clarke *et al.* 2000
Clarke, I., Sandberg, O., Wiley, B. and Hong, T. (2000). Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, pp. 46–66.
- code.google.com I
Protocol buffers. *Home page*.
- code.google.com II
Protocol buffers. *Developer guide: techniques*.
- code.google.com III
Protocol buffers. *Third-party add-ons (RFC implementations)*.
- code.google.com IV
Google App Engine. *Home page*.
- Cohen 2003
Cohen, B. (2003). Incentives build robustness in BitTorrent. May 2003, publicação na Internet.

- Comer 2006 Comer, D.E. (2006). *Internetworking with TCP/IP, Volume 1: Principles, Protocols and Architecture*, 5th edn. Upper Saddle River, NJ: Prentice-Hall.
- Comer 2007 Comer, D.E. (2007). *The Internet Book*, 4th edn. Upper Saddle River, NJ: Prentice-Hall.
- Condict *et al.* 1994 Condict, M., Bolinger, D., McManus, E., Mitchell, D. and Lewontin, S. (1994). *Microkernel modularity with integrated kernel performance*. Technical report, OSF Research Institute, Cambridge, MA, April.
- Coulouris *et al.* 1998 Coulouris, G.F., Dollimore, J. and Roberts, M. (1998). Role and task-based access control in the PerDiS groupware platform. *Third ACM Workshop on Role-Based Access Control*, George Mason University, Washington, DC, October 22–23.
- Coulson *et al.* 2008 Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J. and Sivaharan, T. (2008). A generic component model for building systems software. *ACM Trans. on Computer Systems*, Vol. 26, No. 1, pp. 1–42.
- Cristian 1989 Cristian, F. (1989). Probabilistic clock synchronization. *Distributed Computing*, Vol. 3, pp. 146–58.
- Cristian 1991 Cristian, F. (1991). Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, Vol. 4, pp. 175–87.
- Cristian and Fetzer 1994 Cristian, F. and Fetzer, C. (1994). Probabilistic internal clock synchronization. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, October, pp. 22–31.
- Crow *et al.* 1997 Crow, B., Widjaja, I., Kim, J. and Sakai, P. (1997). IEEE 802.11 wireless local area networks. *IEEE Communications Magazine*, Vol. 35, No. 9, pp. 116–26.
- cryptography.org *North American Cryptography Archives*.
- Culler *et al.* 2001 Culler, D.E., Hill, J., Buonadonna, P., Szewczyk, R. and Woo, A. (2001). A network-centric approach to embedded software for tiny devices. *Proceedings of the First International Workshop on Embedded Software*, Tahoe City, CA, October, pp. 114–130.
- Culler *et al.* 2004 Culler, D., Estrin, D. and Srivastava, M. (2004). Overview of sensor networks. *IEEE Computer*, Vol. 37, No. 8, pp. 41–49.
- Curtin and Dolske 1998 Kurtin, M. and Dolski, J. (1998). A brute force search of DES Keyspace. *;login: – the Newsletter of the USENIX Association*, May.
- Custer 1998 Custer, H. (1998). *Inside Windows NT*, 2nd edn.. Redmond, WA: Microsoft Press.
- Czerwinski *et al.* 1999 Czerwinski, S., Zhao, B., Hodes, T., Joseph, A. and Katz, R. (1999). An architecture for a secure discovery service. In *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networks*. Seattle, WA, pp. 24–53.

- Dabek *et al.* 2001 Dabek, F., Kaashoek, M.F., Karger, D., Morris, R. and Stoica, I. (2001). Wide-area cooperative storage with CFS. In *Proc. of the ACM Symposium on Operating System Principles*, October, pp. 202–15.
- Dabek *et al.* 2003 Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J. and Stoica, I. (2003). Ion Stoica, Towards a common API for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, February, pp. 33–44.
- Daemen and Rijmen 2000 Daemen, J. and Rijmen, V. (2000). The block cipher Rijndael. Quisquater, J.-J. and Schneier, B.(eds.). Smart Card Research and Applications, LNCS 1820. Springer-Verlag, pp. 288–296.
- Daemen and Rijmen 2002 Daemen, J. and Rijmen, V. (2002). *The Design of Rijndael: AES – The Advanced Encryption Standard*, New York: Springer-Verlag.
- Dasgupta *et al.* 1991 Dasgupta, P., LeBlanc Jr., R.J., Ahamad, M. and Ramachandran, U. (1991). The Clouds distributed operating system. *IEEE Computer*, Vol. 24, No. 11, pp. 34–44.
- Davidson 1984 Davidson, S.B. (1984). Optimism and consistency in partitioned database systems. *ACM Transactions on Database Systems*, Vol. 9, No. 3, pp. 456–81.
- Davidson *et al.* 1985 Davidson, S.B., Garcia-Molina, H. and Skeen, D. (1985). Consistency in partitioned networks. *Computing Surveys*, Vol. 17, No. 3, pp. 341–70.
- Davies *et al.* 1998 Davies, N., Friday, A., Wade, S. and Blair, G. (1998). L²imbo: a distributed systems platform for mobile computing. *Mobile Networks and Applications*, Vol. 3, No. 2, pp. 143–156.
- de Ipiña *et al.* 2002 de Ipiña, D.L., Mendonça, P. and Hopper, A. (2002). TRIP: A low-cost vision-based location system for ubiquitous computing. *Personal and Ubiquitous Computing*, Vol. 6, No. 3, pp. 206–219.
- Dean 2006 Dean, J. (2006) Experiences with MapReduce, an abstraction for large-scale computation. In *Proc. of the 15th International Conference on Parallel Architectures and Compilation Techniques*, (PACT'06), Seattle, WA, p. 1.
- Dean and Ghemawat 2004 Dean, J. and Ghemawat, S. (2004). MapReduce: simplified data processing on large clusters. In *Proc. of Operating System Design and Implementation*, (OSDI'04), San Francisco, CA, pp. 137–150.
- Dean and Ghemawat 2008 Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Comms. ACM*, Vol. 51, No. 1, pp. 107–113.
- Dean and Ghemawat 2010 Dean, J. and Ghemawat, S. (2010). MapReduce: a flexible data processing tool. *Comms. ACM*, Vol. 53, No. 1, pp. 72–77.

- Debaty and Caswell 2001 Debaty, P. and Caswell, D. (2001). Uniform web presence architecture for people, places, and things. *IEEE Personal Communications*, Vol. 8, No. 4, pp. 6–11.
- DEC 1990 Digital Equipment Corporation (1990). *In Memoriam: J. C. R. Licklider 1915–1990*. Technical Report 61, DEC Systems Research Center.
- DeCandia *et al.* 2007 DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.* Vol. 41, No. 6, pp. 205–220.
- Delgrossi *et al.* 1993 Delgrossi, L., Halstrick, C., Hehmann, D., Herrtwich, R.G., Krone, O., Sandvoss, J. and Vogt, C. (1993). Media scaling for audiovisual communication with the Heidelberg transport system. *ACM Multimedia '93*, Anaheim, CA.
- Demers *et al.* 1989 Demers, A., Keshav, S. and Shenker, S. (1989). Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM '89*.
- Denning and Denning 1977 Denning, D. and Denning, P. (1977). Certification of programs for secure information flow. *Comms. ACM*, Vol. 20, No. 7, pp. 504–13.
- Dertouzos 1974 Dertouzos, M.L. (1974). Control robotics – the procedural control of physical processes. *IFIP Congress*.
- Dierks and Allen 1999 Dierks, T. and Allen, C. (1999). *The TLS Protocol Version 1.0*. Internet RFC 2246.
- Diffie 1988 Diffie, W. (1988). The first ten years of public-key cryptography. *Proceedings of the IEEE*, Vol. 76, No. 5, pp. 560–77.
- Diffie and Hellman 1976 Diffie, W. and Hellman, M.E. (1976). New directions in cryptography. *IEEE Transactions Information Theory*, Vol. IT-22, pp. 644–54.
- Diffie and Landau 1998 Diffie, W. and Landau, S. (1998). *Privacy on the Line*. Cambridge, MA: MIT Press.
- Dijkstra 1959 Dijkstra, E.W. (1959). A note on two problems in connection with graphs. *Numerische Mathematic*, Vol. 1, pp. 269–71.
- Dilley *et al.* 2002 Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R. and Weihs, B. (2002). Globally distributed content delivery. *IEEE Internet Computing*, pp. 50–58.
- Dingledine *et al.* 2000 Dingledine, R., Freedman, M.J. and Molnar, D. (2000). The Free Haven project: Distributed anonymous storage service. In *Proc. Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July, pp. 67–95.
- Dolev and Malki 1996 Dolev, D. and Malki, D. (1996). The Transis approach to high availability cluster communication. *Comms. ACM.*, Vol. 39, No. 4, pp. 64–70.

- Dolev and Strong 1983 Dolev, D. and Strong, H. (1983). Authenticated algorithms for Byzantine agreement. *SIAM Journal of Computing*, Vol. 12, No. 4, pp. 656–66.
- Dolev *et al.* 1986 Dolev, D., Halpern, J., and Strong, H. (1986). On the possibility and impossibility of achieving clock synchronization. *Journal of Computing Systems Science*, Vol. 32, No. 2, pp. 230–50.
- Dorcey 1995 Dorcey, T. (1995). CU-SeeMe desktop video conferencing software. *Connexions*, Vol. 9, No. 3, pp. 42–45.
- Douceur and Bolosky 1999 Douceur, J.R. and Bolosky, W. (1999). Improving responsiveness of a stripe-scheduled media server. In Proc. IS &T/SPIE Conf. on *Multimedia Computing and Networking*, pp. 192–203.
- Douglis and Ousterhout 1991 Douglis, F. and Ousterhout, J. (1991). Transparent process migration: Design alternatives and the Sprite implementation, *Software – Practice and Experience*, Vol. 21, No. 8, pp. 757–89.
- Draves 1990 Draves, R. (1990). A revised IPC interface. In *Proceedings of the USENIX Mach Workshop*, pp. 101–21, October.
- Draves *et al.* 1989 Draves, R.P., Jones, M.B. and Thompson, M.R. (1989). *MIG – the Mach Interface Generator*. Technical Report, Dept. of Computer Science, Carnegie-Mellon University.
- Druschel and Peterson 1993 Druschel, P. and Peterson, L. (1993). Fbufs: a high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pp. 189–202.
- Druschel and Rowstron 2001 Druschel, P. and Rowstron, A. (2001). PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Schloss Elmau, Germany, May, pp. 75–80.
- Dubois *et al.* 1988 Dubois, M., Scheurich, C. and Briggs, F.A. (1988). Synchronization, coherence and event ordering in multiprocessors. *IEEE Computer*, Vol. 21, No. 2, pp. 9–21.
- Dwork *et al.* 1988 Dwork, C., Lynch, N. and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, Vol. 35, No. 2, pp. 288–323.
- Eager *et al.* 1986 Eager, D., Lazowska, E. and Zahorjan, J. (1986). Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 5, pp. 662–675.
- earth.google.com Google Earth. *Home page*.
- Edney and Arbaugh 2003 Edney, J. and Arbaugh, W. (2003). *Real 802.11 Security: Wi-Fi Protected*. Boston MA: Pearson Education.
- Edwards and Grinter 2001 Edwards, W.K. and Grinter, R. (2001). At home with ubiquitous computing: Seven challenges. In *Proceedings of the Third International Conference on Ubiquitous Computing (Ubicomp 2001)*, Atlanta, GA, Sep.–Oct, pp. 256–272.

- Edwards *et al.* 2002 Edwards, W.K., Newman, M.W., Sedivy, J.Z., Smith, T.F. and Izadi, S. (2002). Challenge: Recombinant computing and the speakeasy approach. In *Proceedings of the Eighth ACM International Conference on Mobile Computing and Networking (MobiCom 2002)*, Atlanta, GA, September, pp. 279–286.
- EFF 1998 Electronic Frontier Foundation (1998). *Cracking DES, Secrets of Encryption Research, Wiretap Politics & Chip Design*. Sebastopol, CA: O'Reilly & Associates.
- Egevang and Francis 1994 Egevang, K. and Francis, P. (1994). *The IP Network Address Translator (NAT)*. Internet RFC 1631.
- Eisler *et al.* 1997 Eisler, M., Chiu, A. and Ling, L. (1997). *RPCSEC_GSS Protocol Specification*. Sun Microsystems, Internet RFC 2203.
- El Abbadi *et al.* 1985 El Abbadi, A., Skeen, D. and Cristian, C. (1985). An efficient fault-tolerant protocol for replicated data management. In *Proc. of the 4th Annual ACM SIGACT/SIGMOD Symposium on Principles of Data Base Systems*, Portland, OR, pp. 215–29.
- Ellis *et al.* 1991 Ellis, C., Gibbs, S. and Rein, G. (1991). Groupware – some issues and experiences. *Comms. ACM*, Vol. 34, No. 1, pp. 38–58.
- Ellison 1996 Ellison, C. (1996). Establishing identity without certification authorities. In *Proc. of the 6th USENIX Security Symposium*, San Jose, CA, July, p.7.
- Ellison *et al.* 1999 Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B. and Ylonen, T. (1999). *SPKI Certificate Theory*. Internet RFC 2693, September.
- Elrad *et al.* 2001 Elrad, T., Filman, R. and Bader A. (eds.) (2001). Theme section on aspect-oriented programming, *Comms. ACM*, Vol. 44, No. 10, pp. 29–32.
- Emmerich 2000 Emmerich, W. (2000). *Engineering Distributed Objects*. New York: John Wiley & Sons.
- esm.cs.cmu.edu ESM project at CMU. *Home page*
- Eugster *et al.* 2003 Eugster, P.T., Felber, P.A., Guerraoui, R. and Kermarrec A-M. (2003). The many faces of publish-subscribe, *ACM Computing Surveys*, Vol. 35, No. 2, pp. 114–131.
- Evans *et al.* 2003 Evans, C. and 15 other authors (2003). *Web Services Reliability (WS-Reliability)*, Fujitsu, Hitachi, NEC, Oracle Corporation, Sonic Software, and Sun Microsystems.
- Fall 2003 Fall, K. (2003). A delay-tolerant network architecture for challenged internets. In *Proceedings of the ACM 2003 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM 2003)*, Karlsruhe, Germany, August, pp. 27–34.
- Farley 1998 Farley, J. (1998). *Java Distributed Computing*. Cambridge, MA: O'Reilly.

- Farrow 2000 Farrow, R. (2000). Distributed denial of service attacks – how Amazon, Yahoo, eBay and others were brought down. *Network Magazine*, March.
- Ferguson and Schneier 2003 Ferguson, N. and Schneier, B. (2003). *Practical Cryptography*. New York: John Wiley & Sons.
- Ferrari and Verma 1990 Ferrari, D. and Verma, D. (1990). A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 4, pp. 368–79.
- Ferreira *et al.* 2000 Ferreira, P., Shapiro, M., Blondel, X., Fambon, O., Garcia, J., Kloostermann, S., Richer, N., Roberts, M., Sandakly, F., Coulouris, G., Dollimore, J., Guedes, P., Hagimont, D. and Krakowiak, S. (2000). PerDiS: Design, implementation, and use of a PERsistent DIstributed Store. In *LNCS 1752: Advances in Distributed Systems*. Berlin, Heidelberg, New York: Springer-Verlag. pp. 427–53.
- Ferris and Langworthy 2004 Ferris, C. and Langworthy, D. (eds.), Bilorusets, R. and 22 other authors (2004). *Web Services Reliable Messaging Protocol (WS-Reliable Messaging)*. BEA, IBM, Microsoft and TibCo.
- Fidge 1991 Fidge, C. (1991). Logical time in distributed computing systems. *IEEE Computer*, Vol. 24, No. 8, pp. 28–33.
- Fielding 2000 Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine
- Fielding *et al.* 1999 Fielding, R., Gettys, J., Mogul, J.C., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T. (1999). *Hypertext Transfer Protocol – HTTP/1.1*. Internet RFC 2616.
- Filman *et al.* 2004 Filman, R., Elrad, T., Clarke, S. and Aksit, M. (2004) *Aspect-Oriented Software Development*. Addison-Wesley.
- Fischer 1983 Fischer, M. (1983). The consensus problem in unreliable distributed systems (a brief survey). In Karpinsky, M. (ed.), *Foundations of Computation Theory*, Vol. 158 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 127–140. Yale University Technical Report YALEU/DCS/TR-273.
- Fischer and Lynch 1982 Fischer, M. and Lynch, N. (1982). A lower bound for the time to assure interactive consistency. *Inf. Process. Letters*, Vol. 14, No. 4, pp. 183–6.
- Fischer and Michael 1982 Fischer, M.J. and Michael, A. (1982). Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the Symposium on Principles of Database Systems*, pp. 70–5.
- Fischer *et al.* 1985 Fischer, M., Lynch, N. and Paterson, M. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, Vol. 32, No. 2, pp. 374–82.

- Fitzgerald and Rashid 1986 Fitzgerald, R. and Rashid, R.F. (1986). The integration of virtual memory management and interprocess communication in Accent. *ACM Transactions on Computer Systems*, Vol. 4, No. 2, pp. 147–77.
- Flanagan 2002 Flanagan, D. (2002). *Java in a Nutshell*, 4th edn. Cambridge, MA: O'Reilly.
- Floyd 1986 Floyd, R. (1986). *Short term file reference patterns in a UNIX environment*. Technical Rep. TR 177, Rochester, NY: Dept of Computer Science, University of Rochester.
- Floyd and Jacobson 1993 Floyd, S. and Jacobson, V. (1993). The synchronization of periodic routing messages. *ACM Sigcomm '93 Symposium*.
- Floyd *et al.* 1997 Floyd, S., Jacobson, V., Liu, C., McCanne, S. and Zhang, L. (1997). A reliable multicast framework for lightweight sessions and application level framing. *IEEE/ACM Transactions on Networking*, Vol. 5, No. 6, pp. 784–803.
- Fluhrer *et al.* 2001 Fluhrer, S., Mantin, I. and Shamir, A. (2001). Weaknesses in the key scheduling algorithm of RC4. In *Proceedings of the 8th annual workshop on Selected Areas of Cryptography (SAC)*, Toronto, Canada, pp. 1–24.
- Ford and Fulkerson 1962 Ford, L.R. and Fulkerson, D.R. (1962). *Flows in Networks*. Princeton, NJ: Princeton University Press.
- Foster and Kesselman 2004 Foster, I. and Kesselman, C. (eds.) (2004). *The Grid 2*. San Francisco, CA: Morgan Kauffman.
- Foster *et al.* 2001 Foster, I., Kesselman, C. and Tuecke, S. (2001). The anatomy of the Grid: Enabling scalable virtual organisations. *Intl. J. Supercomputer Applications.*, Vol. 15, No. 3, pp. 200–222.
- Foster *et al.* 2002 Foster, I., Kesselman, C., Nick, J. and Tuecke, S. (2002). Grid services distributed systems integration. *IEEE Computer*, Vol. 35, No. 6, pp. 37–46.
- Foster *et al.* 2004 Foster, I., Kesselman, C. and Tuecke, S. (2004). *The Open Grid Services Architecture*. In Foster, I. and Kesselman, C. (eds.), *The Grid 2*. San Francisco, CA: Morgan Kauffman.
- Fox *et al.* 1997 Fox, A., Gribble, S., Chawathe, Y., Brewer, E. and Gauthier, P. (1997). Cluster-based scalable network services. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 78–91.
- Fox *et al.* 1998 Fox, A., Gribble, S.D., Chawathe, Y. and Brewer, E.A. (1998). Adapting to network and client variation using active proxies: Lessons and perspectives. *IEEE Personal Communications*, Vol. 5, No. 4, pp. 10–19.
- Fox *et al.* 2003 Fox, D., Hightower, J., Liao, L., Schulz, D. and Borriello, G. (2003). Bayesian filtering for location estimation. *IEEE Pervasive Computing*, Vol. 2, No. 3, pp. 24–33.
- fractal.ow2.org I Projeto Fractal. *Home page*.
- fractal.ow2.org II Projeto Fractal. *Tutorial*.

- France and Rumpe 2007
- France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. *International Conference on Software Engineering (Future of Software Engineering session)*. IEEE Computer Society, Washington, DC, pp. 37–54.
- Fraser *et al.* 2003
- Fraser, K.A., Hand, S.M., Harris, T.L., Leslie, I.M. and Pratt, I.A. (2003). *The Xenoserver computing infrastructure*. Technical Report UCAM-CL-TR-552, Computer Laboratory, University of Cambridge.
- Freed and Borenstein 1996
- Freed, N. and Borenstein, N. (1996). *MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*. September. Internet RFC 1521.
- freenet.project.org
- O projeto Free Network.
- freepastry.org
- O projeto FreePastry.
- Frey and Roman 2007
- Frey, D. and G-C Roman, G-C. (2007). Context-aware publish subscribe in mobile ad hoc networks. In *Proceedings of the 9th International Conference on Coordination Models and Languages*, pp. 37–55.
- Ganesh *et al.* 2003
- Ganesh, A. J., Kermarrec, A. and Massoulié, L. (2003). Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, Vol. 52, No. 2, pp. 139–149.
- Garay and Moses 1993
- Garay, J. and Moses, Y. (1993). Fully polynomial Byzantine agreement in $t+1$ rounds. In *Proceedings of the 25th ACM symposium on theory of computing*, pp. 31–41.
- Garcia-Molina 1982
- Garcia-Molina, H. (1982). Elections in distributed computer systems. *IEEE Transactions on Computers*, Vol. C-31, No. 1, pp. 48–59.
- Garcia-Molina and Spauster 1991
- Garcia-Molina, H. and Spauster, A. (1991). Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, Vol. 9, No. 3, pp. 242–71.
- Garfinkel 1994
- Garfinkel, S. (1994). *PGP: Pretty Good Privacy*. Cambridge, MA: O'Reilly.
- Gehrke and Madden 2004
- Gehrke, J. and Madden, S. (2004). Query processing in sensor networks. *IEEE Pervasive Computing*, Vol. 3, No. 1, pp. 46–55.
- Gelernter 1985
- Gelernter, D. (1985). Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80–112.
- Ghemawat *et al.*, 2003
- Ghemawat, S., Gobioff, H. and Leung, S. (2003). The Google file system. *SIGOPS Oper. Syst. Rev.*, Vol. 37, No. 5, pp. 29–43.
- Gibbs and Tsichritzis 1994
- Gibbs, S.J. and Tsichritzis, D.C. (1994). *Multimedia Programming*. Addison-Wesley.

- Gibson *et al.* 2004 Gibson, G., Courtial, J., Padgett, M.J., Vasnetsov, M., Pas'ko, V., Barnett, S.M. and Franke-Arnold, S. (2004). Free-space information transfer using light beams carrying orbital angular momentum. *Optics Express*, Vol. 12, No. 22, pp. 5448–5456.
- Gifford 1979 Gifford, D.K. (1979). Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pp. 150–62.
- glassfish.dev.java.net GlassFish Application Server. *Home page*.
- Gokhale and Schmidt 1996 Gokhale, A. and Schmidt, D. (1996). Measuring the performance of communication middleware on high-speed networks. In *Proceedings of SIGCOMM '96*, pp. 306–17.
- Golding and Long 1993 Golding, R. and Long, D. (1993). *Modeling replica divergence in a weak-consistency protocol for global-scale distributed databases*. Technical report UCSC-CRL-93-09, Computer and Information Sciences Board, University of California, Santa Cruz.
- Goldschlag *et al.* 1999 Goldschlag, D., Reed, M. and Syverson, P. (1999). Onion routing for anonymous and private Internet connections. *Communications of the ACM*, Vol. 42, No. 2, pp. 39–41.
- Golub *et al.* 1990 Golub, D., Dean, R., Forin, A. and Rashid, R. (1990). *UNIX as an application program*. In *Proc. USENIX Summer Conference*, pp. 87–96.
- Gong 1989 Gong, L. (1989). A secure identity-based capability system. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May, pp. 56–63.
- googleblog.blogspot.com I The Official Google Blog. *Powering a Google search*.
- googleblog.blogspot.com II The Official Google Blog. *New Search Engine: Caffeine*.
- googletesting.blogspot.com The Google Testing Blog. *Home page*.
- Gordon 1984 Gordon, J. (1984). *The Story of Alice and Bob*. After dinner speech, see also: en.wikipedia.org/wiki/Alice_and_Bob.
- Govindan and Anderson 1991 Govindan, R. and Anderson, D.P. (1991). Scheduling and IPC mechanisms for continuous media. *ACM Operating Systems Review*, Vol. 25, No. 5, pp. 68–80.
- Goyal and Carter 2004 Goyal, S. and Carter, J. (2004). A lightweight secure cyber foraging infrastructure for resource-constrained devices. In *Proceedings of the Sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2004)*, December, pp. 186–195.
- Graumann *et al.* 2003 Graumann, D., Lara, W., Hightower, J. and Borriello, G. (2003). Real-world implementation of the Location Stack: The Universal Location Framework. In *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems & Applications (WMCSA 2003)*, Monterey, CA, October, pp. 122–128.

- Grace *et al.* 2003 Grace, P., Blair, G.S. and Samuel, S. (2003). ReMMoC: A reflective middleware to support mobile client interoperability. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA '03)*, Catania, Sicily, November, pp. 1170–1187.
- Grace *et al.* 2008 Grace, P., Hughes, D., Porter, B., Blair, G.S., Coulson, G. and Taiani, F. (2008). Experiences with open overlays: A middleware approach to network heterogeneity. In *Proceedings of the 3rd ACM Sigops/Eurosys European Conference on Computer Systems (Eurosyst '08)*, Glasgow, Scotland, pp. 123–136.
- Gray 1978 Gray, J. (1978). Notes on database operating systems. In *Operating Systems: an Advanced Course. Lecture Notes in Computer Science*, Vol. 60, Springer-Verlag, pp. 394–481.
- Gray and Szalay 2002 Gray, J. and Szalay, A. (2002). *The World-Wide Telescope, an Archetype for Online Science*. Technical Report. MSR-TR-2002-75. Microsoft Research.
- Greenfield and Dornan 2004 Greenfield, D. and Dornan, A. (2004). Amazon: Web site to web services, *Network Magazine*, October.
- Grimm 2004 Grimm, R. (2004). One.world: Experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, Vol. 3, No. 3, pp. 22–30.
- Gruteser and Grunwald 2003 Gruteser, M. and Grunwald, D. (2003). Enhancing location privacy in wireless LAN through disposable interface identifiers: a quantitative analysis. In *Proceedings of the 1st ACM international workshop on Wireless mobile applications and services on WLAN hotspots (WMASH '03)*, San Diego, CA, September, pp. 46–55.
- Guerraoui *et al.* 1998 Guerraoui, R., Felber, P., Garbinato, B. and Mazouni, K. (1998). System support for object groups. In *Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'98)*.
- Gummadi *et al.* 2003 Gummadi, K.P., Gummadi, R., Gribble, S.D., Ratnasamy, S., Shenker, S. and Stoica, I. (2003). The impact of DHT routing geometry on resilience and proximity. In Proc. ACM SIGCOMM 2003, pp. 381–94.
- Gupta *et al.* 2004 Gupta, A., Sahin, O. D., Agrawal, D. and Abbadi, A. E. (2004). Meghdoot: Content-based publish/subscribe over P2P networks. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, Toronto, Canada, October, pp. 254–273.
- Gusella and Zatti 1989 Gusella, R. and Zatti, S. (1989). The accuracy of clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE Transactions on Software Engineering*, Vol. 15, No. 7, pp. 847–53.
- Guttman 1999 Guttman, E. (1999). Service location protocol: Automatic discovery of IP network services. *IEEE Internet Computing*, Vol. 3, No. 4, pp. 71–80.

- Haartsen *et al.* 1998
- hadoop.apache.org
- Hadzilacos and Toueg 1994
- Hand *et al.* 2003
- Handley *et al.* 1999
- Harbison 1992
- Härder 1984
- Härder and Reuter 1983
- Harrenstien *et al.* 1985
- Harter and Hopper 1994
- Harter *et al.* 2002
- Härtig *et al.* 1997
- Hartman and Ousterhout 1995
- Hauch and Reiser 2000
- Haylon *et al.* 1998
- Haartsen, J., Naghshineh, M., Inouye, J., Joeressen, O.J. and Allen, W. (1998). Bluetooth: Vision, goals and architecture. *ACM Mobile Computing and Communications Review*, Vol. 2, No. 4, pp. 38–45.
- Hadoop. *Home page*.
- Hadzilacos, V. and Toueg, S. (1994). *A Modular Approach to Fault-tolerant Broadcasts and Related Problems*, Technical report, Dept of Computer Science, University of Toronto.
- Hand, S., Harris, T., Kotsovinos, E. and Pratt, I. (2003). Controlling the XenoServer open platform. In *Proceedings of the 6th IEEE Conference on Open Architectures and Network Programming (OPEN ARCH 2003)*, pp. 3–11.
- Handley, M., Schulzrinne, H., Schooler, E. and Rosenberg, J. (1999). *SIP: Session Initiation Protocol*. Internet RFC 2543.
- Harbison, S. P. (1992). *Modula-3*, Englewood Cliffs, NJ: Prentice-Hall.
- Härder, T. (1984). Observations on optimistic concurrency control schemes. *Information Systems*, Vol. 9, No. 2, pp. 111–20.
- Härder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys*, Vol. 15, No. 4, pp. 287–317.
- Harrenstien, K., Stahl, M. and Feinler, E. (1985). *DOD Internet Host Table Specification*. Internet RFC 952.
- Harter, A. and Hopper, A. (1994). A distributed location system for the active office. *IEEE Network*, Vol. 8, No. 1, pp. 62–70.
- Harter, A., Hopper, A., Steggles, P., Ward, A. and Webster, P. (2002). The anatomy of a context-aware application. *Wireless Networks*, Vol. 8, No. 2–3, pp. 187–197.
- Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S. and Wolter, J. (1997). The performance of kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pp. 66–77.
- Hartman, J. and Ousterhout, J. (1995). The Zebra striped network file system. *ACM Trans. on Computer Systems*, Vol. 13, No. 3, pp. 274–310.
- Hauch, R. and Reiser, H. (2000). Monitoring quality of service across organisational boundaries. In *Trends in Distributed Systems: Towards a Universal Service Market*, Proc. Third Intl. IFIP/HGI Working conference, USM, September.
- Hayton, R., Bacon, J. and Moody, K. (1998). OASIS: Access control in an open, distributed environment. In *Proceedings of the IEEE Symposium on Security and Privacy*, May, Oakland, CA, pp. 3–14.

- Hedrick 1988 Hedrick, R. (1988). *Routing Information Protocol*. Internet RFC 1058.
- Heidemann *et al.* 2001 Heidemann, J., Silva, F., Intanagonwiwat, C., Govindan, R., Estrin, D. and Ganesan, D. (2001). Building efficient wireless sensor networks with low-level naming. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Alberta, Canada, October, pp. 146–159.
- Heineman and Councill 2001 Heineman, G.T. and Councill, W.T. (2001). *Component-Based Software Engineering: Putting the Pieces Together*. Reading, MA: Addison-Wesley.
- Hennessy and Patterson 2006 Hennessy, J.L. and Patterson, D.A. (2006). *Computer Architecture: A Quantitative Approach*, 4th edn. San Francisco:CA: Morgan Kaufmann.
- Henning 1998 Henning, M. (1998). Binding, migration and scalability in CORBA. *Comms. ACM*, Vol. 41, No. 10, pp. 62–71.
- Henning and Vinoski 1999 Henning, M. and Vinoski, S. (1999). *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley.
- Herlihy 1986 Herlihy, M. (1986). A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, Vol. 4, No. 1, pp. 32–53.
- Herlihy and Wing 1990 Herlihy, M. and Wing, J. (1990). On linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, pp. 463–92.
- Herrtwich 1995 Herrtwich, R.G. (1995). Achieving quality of service for multimedia applications. *ERSADS '95, European Research Seminar on Advanced Distributed Systems*, l'Alpe d'Huez, France, April.
- Hightower and Bordello 2001 Hightower, J. and Bordello, G. (2001). Location systems for ubiquitous computing. *IEEE Computer*, Vol. 34, No. 8, pp. 57–66.
- Hightower *et al.* 2002 Hightower, J., Brumitt, B. and Borriello, G. (2002). The Location Stack: A layered model for location in ubiquitous computing. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems & Applications (WMCSA 2002)*, Callicoon, NY, June, pp. 22–28.
- Hill *et al.* 2000 Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. and Pister, K. (2000). System architecture directions for networked sensors. In *Proceedings of the Ninth ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, MA, November, pp. 93–104.
- Hirsch 1997 Hirsch, F.J. (1997). Introducing SSL and Certificates using SSLeay. *World Wide Web Journal*, Vol. 2, No. 3, pp. 141–173.

- Holmquist *et al.* 2001 Holmquist, L.E., Mattern, F., Schiele, B., Alahuhta, P., Beigl, M. and Gellersen, H.-W. (2001). Smart-Its Friends: A technique for users to easily establish connections between smart artefacts. In *Proceedings of the Third International Conference on Ubiquitous Computing (Ubicomp 2001)*, Atlanta, GA, September–October, pp. 116–122.
- Housley 2002 Housley, R. (2002). *Cryptographic Message Syntax (CMS) Algorithms*. Internet RFC 3370.
- Howard *et al.* 1988 Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N. and West, M.J. (1988). Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pp. 51–81.
- Huang *et al.* 2000 Huang, A., Ling, B., Barton, J. and Fox, A. (2000). Running the Web backwards: Appliance data services. In *Proceedings of the 9th international World Wide Web Conference*, pp. 619–31.
- Huitema 1998 Huitema, C. (1998). *IPv6 – the New Internet Protocol*. Upper Saddle River, NJ: Prentice-Hall.
- Huitema 2000 Huitema, C. (2000). *Routing in the Internet*, 2nd edn. Englewood Cliffs, NJ: Prentice-Hall.
- Hull *et al.* 2004 Hull, R., Clayton, B. and Melamad, T. (2004). Rapid authoring of mediascapes. In *Proceedings of the Sixth International Conference on Ubiquitous Computing (Ubicomp 2004)*, Nottingham, England, September, pp. 125–142.
- hulu.com Hulu. *Home page*.
- Hunt *et al.* 2007 Hunt, G.C. and Larus, J. R., Singularity: Rethinking the Software Stack, In *ACM SIGOPS Operating Systems Review*, Vol. 41, No. 2, pp. 37–49.
- Hunter and Crawford 1998 Hunter, J. and Crawford, W. (1998). *Java Servlet Programming*. Sebastopol, CA: O'Reilly.
- Hutchinson and Peterson 1991 Hutchinson, N. and Peterson, L. (1991). The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, Vol. 17, No. 1, pp. 64–76.
- Hutchinson *et al.* 1989 Hutchinson, N.C., Peterson, L.L., Abbott, M.B. and O'Malley, S.W. (1989). RPC in the x-Kernel: Evaluating new design techniques. In *Proc. of the 12th ACM Symposium on Operating System Principles*, pp. 91–101.
- Hyman *et al.* 1991 Hyman, J., Lazar, A.A. and Pacifici, G. (1991). MARS – The MAGNET-II Real-Time Scheduling Algorithm. *ACM SIGCOM '91*, Zurich.
- IEEE 1985a Institute of Electrical and Electronic Engineers (1985). *Local Area Network – CSMA/CD Access Method and Physical Layer Specifications*. American National Standard ANSI/IEEE 802.3, IEEE Computer Society.

- IEEE 1985b Institute of Electrical and Electronic Engineers (1985). *Local Area Network – Token Bus Access Method and Physical Layer Specifications*. American National Standard ANSI/IEEE 802.4, IEEE Computer Society.
- IEEE 1985c Institute of Electrical and Electronic Engineers (1985). *Local Area Network – Token Ring Access Method and Physical Layer Specifications*. American National Standard ANSI/IEEE 802.5, IEEE Computer Society.
- IEEE 1990 Institute of Electrical and Electronic Engineers (1990). *IEEE Standard 802: Overview and Architecture*. American National Standard ANSI/IEEE 802, IEEE Computer Society.
- IEEE 1994 Institute of Electrical and Electronic Engineers (1994). *Local and Metropolitan Area Networks – Part 6: Distributed Queue Dual Bus (DQDB) Access Method and Physical Layer Specifications*. American National Standard ANSI/IEEE 802.6, IEEE Computer Society.
- IEEE 1999 Institute of Electrical and Electronic Engineers (1999). *Local and Metropolitan Area Networks – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. American National Standard ANSI/IEEE 802.11, IEEE Computer Society.
- IEEE 2002 Institute of Electrical and Electronic Engineers (2002). *Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs)*. American National Standard ANSI/IEEE 802.15.1, IEEE Computer Society.
- IEEE 2003 Institute of Electrical and Electronic Engineers (2003). *Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*. American National Standard ANSI/IEEE 802.15.4, IEEE Computer Society.
- IEEE2004a Institute of Electrical and Electronic Engineers (2004). *IEEE Standard for Local and Metropolitan Area Networks – Part 16: Air Interface for Fixed Broadband Wireless Access Systems*. American National Standard ANSI/IEEE 802.16, IEEE Computer Society
- IEEE 2004b Institute of Electrical and Electronic Engineers (2004). *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Medium Access Control (MAC) Security Enhancement*. American National Standard ANSI/IEEE 802.11i, IEEE Computer Society.
- Imielinski and Navas 1999 Imielinski, T. and Navas, J.C. (1999). GPS-based geographic addressing, routing, and resource discovery. *Comms. ACM*, Vol. 42, No. 4, pp. 86–92.
- International PGP *The International PGP Home Page*.
- Internet World Stats 2004 Internet World Stats. www.internetworldstats.com

- Ishii and Ullmer 1997 Ishii, H. and Ullmer, B.,(1997). Tangible bits: Towards seamless interfaces between people, bits and atoms. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '97)*, Atlanta, GA, March, pp. 234–241.
- ISO 1992 International Organization for Standardization (1992). *Basic Reference Model of Open Distributed Processing, Part 1: Overview and Guide to Use*. ISO/IEC JTC1/SC212/WG7 CD 10746-1, International Organization for Standardization.
- ISO 8879 International Organization for Standardization (1986). *Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*.
- ITU/ISO 1997 International Telecommunication Union / International Organization for Standardization (1997). Recommendation X.500 (08/97): *Open Systems Interconnection – The Directory: Overview of Concepts, Models and Services*.
- Iyer *et al.* 2002 Iyer, S., Rowstron, A. and Druschel, P. (2002). Squirrel: A decentralized peer-to-peer web cache. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pp. 213–22.
- jakarta.apache.org The Apache foundation. *Apache Tomcat*.
- [java.sun.com I](http://java.sun.com/I) Sun Microsystems. *Java Remote Method Invocation*.
- [java.sun.com II](http://java.sun.com/II) Sun Microsystems. *Java Object Serialization Specification*.
- [java.sun.com III](http://java.sun.com/III) Sun Microsystems. *Servlet Tutorial*.
- [java.sun.com IV](http://java.sun.com/IV) Jordan, M. and Atkinson, M. (1999). *Orthogonal Persistence for the Java Platform – Draft Specification*. Sun Microsystems Laboratories, Palo Alto, CA.
- [java.sun.com V](http://java.sun.com/V) Sun Microsystems, *Java Security API*.
- [java.sun.com VI](http://java.sun.com/VI) Sun Microsystems (1999). *JavaSpaces technology*.
- [java.sun.com VII](http://java.sun.com/VII) Sun Microsystems. *The Java Web Services Tutorial*.
- [java.sun.com VIII](http://java.sun.com/VIII) Sun Microsystems (2003). *Java Data Objects (JDO)*.
- [java.sun.com IX](http://java.sun.com/IX) Sun Microsystems. *Java Remote Object Activation Tutorial*.
- [java.sun.com X](http://java.sun.com/X) Sun Microsystems. *JavaSpaces Service Specification*.
- [java.sun.com XI](http://java.sun.com/XI) Sun Microsystems. *Java Messaging Service (JMS) home page*.
- [java.sun.com XII](http://java.sun.com/XII) Sun Microsystems. *Enterprise JavaBeans Specification*.
- [java.sun.com XIII](http://java.sun.com/XIII) Sun Microsystems. *Java Persistence API Specification*.
- Johanson and Fox 2004 Johanson, B. and Fox, A. (2004). Extending tuplespaces for coordination in interactive workspaces. *Journal of Systems and Software*, Vol. 69, No. 3, pp. 243–266.
- Johnson and Zwaenepoel 1993 Johnson, D. and Zwaenepoel, W. (1993). The peregrine high-performance RPC system. *Software–Practice and Experience*, Vol. 23, No. 2, pp. 201–21.

- jonas.ow2.org OW2 Consortium. *JOnAS Application Server*.
- Jordan 1996 Jordan, M. (1996). Early experiences with persistent Java. In *Proceedings of the First International Workshop on Persistence and Java*. Glasgow, Scotland, September, pp. 1–9..
- Joseph *et al.* 1997 Joseph, A., Tauber, J. and Kaashoek, M. (1997). Mobile Computing with the Rover Toolkit. *IEEE Transactions on Computers: Special Issue on Mobile Computing*, Vol. 46, No. 3, pp. 337–52.
- Jul *et al.* 1988 Jul, E., Levy, H., Hutchinson, N. and Black, A. (1988). Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pp. 109–33.
- jxta.dev.javajet Jxta Community. *Home page*.
- Kaashoek and Tanenbaum 1991 Kaashoek, F. and Tanenbaum, A. (1991). Group communication in the Amoeba distributed operating system. In Proceedings of the 11th International Conference on Distributed Computer Systems, pp. 222–30.
- Kaashoek *et al.* 1989 Kaashoek, F., Tanenbaum, A., Flynn Hummel, S. and Bal, H. (1989). An efficient reliable broadcast protocol. *Operating Systems Review*, Vol. 23, No. 4, pp. 5–20.
- Kaashoek *et al.* 1997 Kaashoek, M., Engler, D., Ganzer, G., Briceño, H., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Jannotti, J. and Mackenzie, K. (1997). Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 52–65.
- Kahn 1967 Kahn, D. (1967). *The Codebreakers: The Story of Secret Writing*. New York: Macmillan.
- Kahn 1983 Kahn, D. (1983). *Kahn on Codes*. New York: Macmillan.
- Kahn 1991 Kahn, D. (1991). *Seizing the Enigma*. Boston: Houghton Mifflin.
- Kaler 2002 Kaler, C. (ed.) (2002). *Specification: Web Services Security (WS-Security)*.
- Kaliski and Staddon 1998 Kaliski, B. and Staddon, J. (1998). *RSA Cryptography Specifications*, Version 2.0. Internet RFC 2437.
- Kantor and Lapsley 1986 Kantor, B. and Lapsley, P. (1986). *Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News*. Internet RFC 977.
- Kehne *et al.* 1992 Kehne, A., Schonwalder, J. and Langendorfer, H. (1992). A nonce-based protocol for multiple authentications. *ACM Operating Systems Review*, Vol. 26, No. 4, pp. 84–9.
- Keith and Wittle 1993 Keith, B.E. and Wittle, M. (1993). LADDIS: The next generation in NFS file server benchmarking. *USENIX Association Conference Proceedings*, Berkeley, CA, June, pp. 261–78.

- Kiciman and Fox 2000 Kiciman, E. and Fox, A. (2000). Using dynamic mediation to integrate COTS entities in a ubiquitous computing environment. In *Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing (HUC2K)*, Bristol, England, September, pp. 211–226.
- Kille 1992 Kille, S. (1992). *Implementing X.400 and X.500: The PP and QUILP Systems*. Boston, MA: Artech House.
- Kindberg 1995 Kindberg, T. (1995). A sequencing service for group communication (abstract). In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, p. 260. Technical Report No. 698. Queen Mary and Westfield College Dept. of CS, 1995.
- Kindberg 2002 Kindberg, T. (2002). Implementing physical hyperlinks using ubiquitous identifier resolution. In *Proceedings of the Eleventh International World Wide Web Conference (WWW2002)*, Honolulu, HI, May pp. 191–199.
- Kindberg and Barton 2001 Kindberg, T. and Barton, J. (2001). A web-based nomadic computing system. *Computer Networks*, Vol. 35, No. 4, pp. 443–456.
- Kindberg and Fox 2001 Kindberg, T. and Fox, A. (2001). System software for ubiquitous computing. *IEEE Pervasive Computing*, Vol. 1, No. 1, pp. 70–81.
- Kindberg and Zhang 2003a Kindberg, T. and Zhang, K. (2003). Secure spontaneous device association. In *Proceedings of the Fifth International Conference on Ubiquitous Computing (Ubicomp 2003)*, Seattle, WA, October, pp. 124–131.
- Kindberg and Zhang 2003b Kindberg, T. and Zhang, K. (2003). Validating and securing spontaneous associations between wireless devices. In *Proceedings of the 6th Information Security Conference (ISC'03)*, Bristol, England, October, pp. 44–53.
- Kindberg *et al.* 1996 Kindberg, T., Coulouris, G., Dollimore, J. and Heikkinen, J. (1996). Sharing objects over the Internet: The Mushroom approach. In *Proceedings of the IEEE Global Internet 1996*, London, England, Nov., pp. 67–71.
- Kindberg *et al.* 2002a Kindberg, T., Barton, J., Morgan, J., Becker, G., Bedner, I., Caswell, D., Debaty, P., Gopal, G., Frid, M., Krishnan, V., Morris, H., Pering, C., Schettino, J. and Serra, B. (2002). People, places, things: Web presence for the real world. *Mobile Networks and Applications (MONET)*, Vol. 7, No. 5, pp. 365–376.
- Kindberg *et al.* 2002b Kindberg, T., Zhang, K. and Shanka, N. (2002). Context authentication using constrained channels. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems & Applications (WMCSA 2002)*, Callicoon, NY, June, pp. 14–21.
- Kirsch and Amir 2008 Kirsch, J. and Amir, Y. (2008). Paxos for system builders: An overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS '08)*, Vol. 341, Yorktown Heights, NY, pp. 1–6.

- Kistler and Satyanarayanan 1992 Kistler, J.J. and Satyanarayanan, M. (1992). Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 3–25.
- Kleinrock 1961 Kleinrock, L. (1961). *Information Flow in Large Communication Networks*. MIT, RLE Quarterly Progress Report, July.
- Kleinrock 1997 Kleinrock, L. (1997). Nomadicity: Anytime, anywhere in a disconnected world. *Mobile Networks and Applications*, Vol. 1, No. 4, pp. 351–7.
- Kohl and Neuman 1993 Kohl, J. and Neuman, C. (1993). *The Kerberos Network Authentication Service (V5)*. Internet RFC 1510.
- Kon *et al.* 2002 Kon, F., Costa, F., Blair, G. and Campbell, R. (2002), The case for reflective middleware. *Comms. ACM*, Vol. 45, No. 6, pp. 33–38.
- Konstantas *et al.*, 1997 Konstantas, D., Orlarey, Y., Gibbs, S. and Carbonel, O. (1997). Distributed music rehearsal. In *Proceedings of the International Computer Music Conference 97*, pp. 54–64.
- Kopetz and Verissimo 1993 Kopetz, H. and Verissimo, P. (1993). Real time and dependability concepts. In Mullender, (eds.), *Distributed Systems*, 2nd edn. Reading, MA: Addison-Wesley.
- Kopetz *et al.* 1989 Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C. and Zainlinger, R. (1989). Distributed fault-tolerant real-time systems – The MARS Approach. *IEEE Micro*, Vol. 9, No. 1, pp. 112–26.
- Krawczyk *et al.* 1997 Krawczyk, H., Bellare, M. and Canetti, R. (1997). *HMAC: Keyed-Hashing for Message Authentication*. Internet RFC 2104.
- Krumm *et al.* 2000 Krumm, J., Harris, S., Meyers, B., Brumitt, B., Hale, M. and Shafer, S. (2000). Multi-camera multi-person tracking for EasyLiving. In *Proceedings of the Third IEEE International Workshop on Visual Surveillance (VS'2000)*, Dublin, Ireland, July, pp. 3–10.
- Kshemkalyani and Singhal 1991 Kshemkalyani, A. and Singhal, M. (1991). Invariant-based verification of a distributed deadlock detection algorithm. *IEEE Transactions on Software Engineering*, Vol. 17, No. 8, pp. 789–99.
- Kshemkalyani and Singhal 1994 Kshemkalyani, A. and Singhal, M. (1994). On characterisation and correctness of distributed deadlock detection. *Journal of Parallel and Distributed Computing*, Vol. 22, No. 1, pp. 44–59.
- Kubiatowicz 2003 Kubiatowicz, J. (2003). Extracting guarantees from chaos, *Communications of the ACM*, vol. 46, No. 2, pp. 33–38.
- Kubiatowicz *et al.*, 2000 Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C. and Zhao, B. (2000). OceanStore: an architecture for global-scale persistent storage. In *Proc. ASPLOS 2000*, November, pp. 190–201.

- Kung and Robinson 1981 Kung, H.T. and Robinson, J.T. (1981). Optimistic methods for concurrency control. *ACM Transactions on Database Systems*, Vol. 6, No. 2, pp. 213–26.
- Kurose and Ross 2007 Kurose, J.F. and Ross, K.W. (2007). *Computer Networking: A Top-Down Approach Featuring the Internet*. Boston, MA: Addison-Wesley Longman.
- Ladin *et al.* 1992 Ladin, R., Liskov, B., Shrira, L. and Ghemawat, S. (1992). Providing availability using lazy replication. *ACM Transactions on Computer Systems*, Vol. 10, No. 4, pp. 360–91.
- Lai 1992 Lai, X. (1992). On the design and security of block ciphers. *ETH Series in Information Processing*, Vol. 1. Konstanz, Germany: Hartung-Gorre Verlag.
- Lai and Massey 1990 Lai, X. and Massey, J. (1990). A proposal for a new block encryption standard. In *Proceedings Advances in Cryptology-Eurocrypt '90*, pp. 389–404.
- Lamport 1978 Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. *Comms. ACM*, Vol. 21, No. 7, pp. 558–65.
- Lamport 1979 Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, Vol. C-28, No. 9, pp. 690–1.
- Lamport 1986 Lamport, L. (1986). On interprocess communication, parts I and II. *Distributed Computing*, Vol. 1, No. 2, pp. 77–101.
- Lamport 1989 Lamport, L. (1989). *The Part-Time Parliament*. Technical Report 49, DEC SRC, Palo Alto, CA.
- Lamport 1998 Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, Vol. 16, No. 2, pp. 133–69.
- Lamport *et al.* 1982 Lamport, L., Shostak, R. and Pease, M. (1982). Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 382–401.
- Lampson 1971 Lampson, B. (1971). Protection. In *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, p. 437. Reprinted in *ACM Operating Systems Review*. Vol. 8, No. 1, p. 18.
- Lampson 1981 Lampson, B.W. (1981). Atomic transactions. In *Distributed systems: Architecture and Implementation*. Vol 105 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 254–9.
- Lampson 1986 Lampson, B.W. (1986). Designing a global name service. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pp. 1–10.
- Lampson *et al.* 1992 Lampson, B.W., Abadi, M., Burrows, M. and Wobber, E. (1992). Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, Vol. 10, No. 4, pp. 265–310.

- Langheinrich 2001 Langheinrich, M. (2001). Privacy by design – principles of privacy-aware ubiquitous systems. In *Proceedings of the Third International Conference on Ubiquitous Computing (Ubicomp 2001)*, Atlanta, GA, Sep.–Oct, pp. 273–291.
- Langville and Meyer 2006 Lanville, A.M. and Meyer, C.D. (2006). *Pagerank and Beyond: The Science of Search Engine Rankings*. Princeton, NJ: Princeton University Press.
- Langworthy 2004 Langworthy, D. (ed.) (2004) *Web Services Coordination. (WS-Coordination)*, IBM, Microsoft, BEA.
- Leach *et al.* 1983 Leach, P.J., Levine, P.H., Douros, B.P., Hamilton, J.A., Nelson, D.L. and Stumpf, B.L. (1983). The architecture of an integrated local network. *IEEEJ. Selected Areas in Communications*, Vol. SAC-1, No. 5, pp. 842–56.
- Lee and Thekkath 1996 Lee, E.K. and Thekkath, C.A. (1996). Petal: Distributed virtual disks, In *Proc. of the 7th Intl. Conf. on Architectural Support for Prog. Langs, and Operating Systems*, October, pp. 84–96.
- Lee *et al.* 1996 Lee, C., Rajkumar, R. and Mercer, C. (1996). Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach. In *Proceedings Multimedia Japan '96*.
- Leffler *et al.* 1989 Leffler, S., McKusick, M., Karels, M. and Quartermain, J. (1989). *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Reading, MA: Addison-Wesley.
- Leibowitz *et al.* 2003 Leibowitz, N., Ripeanu, M. and Wierzbicki, A. (2003). Deconstructing the Kazaa network. In *Proc. of the 3rd IEEE Workshop on Internet Applications (WIAPP'03)*, Santa Clara, CA, p. 112.
- Leiner *et al.* 1997 Leiner, B.M., Cerf, V.G., Clark, D.D., Kahn, R.E., Kleinrock, L., Lynch, D.C., Postel, J., Roberts, L.G. and Wolff, S. (1997). A brief history of the Internet. *Comms. ACM*, Vol. 40, No. 1, pp. 102–108.
- Leland *et al.* 1993 Leland, W.E., Taqqu, M.S., Willinger, W. and Wilson, D.V. (1993). On the self-similar nature of Ethernet traffic. *ACM SIGCOMM '93*, San Francisco.
- Leslie *et al.* 1996 Leslie, I., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R. and Hyden, E. (1996). The design and implementation of an operating system to support distributed multimedia applications. *ACM Journal of Selected Areas in Communication*, Vol. 14, No. 7, pp. 1280–97.
- Liedtke 1996 Liedtke, J. (1996). Towards real microkernels. *Comms. ACM*, Vol. 39, No. 9, pp. 70–7.
- Linux AFS *The Linux AFS FAQ*.
- Liskov 1988 Liskov, B. (1988). Distributed programming in Argus. *Comms. ACM*, Vol. 31, No. 3, pp. 300–12.

- Liskov 1993 Liskov, B. (1993). Practical uses of synchronized clocks in distributed systems. *Distributed Computing*, Vol. 6, No. 4, pp. 211–19.
- Liskov and Scheifler 1982 Liskov, B. and Scheifler, R.W. (1982). Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, pp. 381–404.
- Liskov and Shrira 1988 Liskov, B. and Shrira, L. (1988). Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the SIGPLAN '88 Conference Programming Language Design and Implementation*. Atlanta, GA, pp. 260–7.
- Liskov *et al.* 1991 Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L. and Williams, M. (1991). Replication in the Harp file system. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pp. 226–38.
- Liu and Albitz 1998 Liu, C. and Albitz, P. (1998). *DNS and BIND*, third edition. O'Reilly.
- Liu and Layland 1973 Liu, C.L. and Layland, J.W. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, Vol. 20, No. 1, pp. 46–61.
- Liu *et al.* 2005 Liu, J., Sacchetti, D., Sailhan, F. and Issarny, V. (2005). Group management for mobile ad hoc networks: Design, implementation and experiment. In *Proceedings of the 6th international Conference on Mobile Data Management*, New York, pp. 192–199.
- Liu *et al.* 2008 Liu, J., Rao, S.G., Li, B. and Zhang, H. (2008). Opportunities and challenges of peer-to-peer Internet video broadcast. In *Proceedings of the IEEE, Special Issue on Recent Advances in Distributed Multimedia Communications*, Vol. 96, No. 1, pp. 11–24.
- Loepere 1991 Loepere, K. (1991). *Mack 3 Kernel Principles*. Open Software Foundation and Carnegie-Mellon University.
- Lundelius and Lynch 1984 Lundelius, J. and Lynch, N. (1984). An upper and lower bound for clock synchronization. *Information and Control*, Vol. 62, No. 2/3, pp. 190–204.
- Lv *et al.* 2002 Lv, Q., Cao, P., Cohen, E., Li, K., and Shenker, S. (2002). Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Marina Del Rey, CA, pp. 258–259.
- Lynch 1996 Lynch, N. (1996). *Distributed Algorithms*. San Francisco, CA: Morgan Kaufmann.
- Ma 1992 Ma, C. (1992). *Designing a Universal Name Service*. Technical Report 270, University of Cambridge.

- Macklem 1994 Macklem, R. (1994). Not quite NFS: Soft cache consistency for NFS. In *Proceedings of the Winter '94 USENIX Conference*, San Francisco, CA, January, pp. 261–278.
- Madhavapeddy *et al.* 2003 Madhavapeddy, A., Scott, D. and Sharp, R. (2003), Context-aware computing with sound. In *Proceedings of the Fifth International Conference on Ubiquitous Computing (Ubicomp 2003)*, Seattle, WA, October, pp. 315–332.
- Maekawa 1985 Maekawa, M. (1985). A \sqrt{N} algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, Vol. 3, No. 2, pp. 145–159.
- Maffeis 1995 Maffeis, S. (1995). Adding group communication and fault tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies*. Monterey, CA. pp. 135–146.
- Magee and Sloman 1989 Magee, J. and Sloman, M. (1989). Constructing distributed systems in Conic. *IEEE Trans. Software Engineering* Vol. 15, No. 6, pp. 663–675.
- Malkin 1993 Malkin, G. (1993). *RIP Version 2 – Carrying Additional Information*, Internet RFC 1388.
- Mamei and Zarnbonelli 2009 Mamei, M. and Zambonelli, F. (2009). Programming pervasive and mobile computing applications: The TOTA approach. *ACM Transactions on Software Engineering and Methodology*, Vol. 19, No. 4, p. 263.
- maps.google.com Google Maps. *Home page*.
- Marsh *et al.* 1991 Marsh, B., Scott, M., LeBlanc, T. and Markatos, E. (1991). First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pp. 110–21.
- Martin *et al.* 2004 Martin, T., Hsiao, M., Ha, D. and Krishnaswami, J. (2004). Denial-of-service attacks on battery-powered mobile computers. In *Proceedings of the 2nd IEEE Pervasive Computing Conference*, Orlando, FL, March, pp. 309–318.
- Marzullo and Neiger 1991 Marzullo, K. and Neiger, G. (1991). Detection of global state predicates. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, pp. 254–72.
- Mattern 1989 Mattern, F. (1989). Virtual time and global states in distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, Amsterdam, North-Holland, pp. 215–26.
- Maymounkov and Mazieres 2002 Maymounkov, P. and Mazieres, D. (2002). Kademia: A peer-to-peer information system based on the XOR metric. In *Proceedings of IPTPS02*, Cambridge, MA, pp. 53–65.
- mbone *BIBs: Introduction to the Multicast Backbone*.
- McGraw and Felden 1999 McGraw, G. and Felden, E. (1999). *Securing Java*. New York: John Wiley & Sons.

- McKusick and Quinlan 2010
- Meier and Cahill 2010
- Melliar-Smith *et al.* 1990
- Menezes 1993
- Menezes *et al.* 1997
- Metcalfe and Bogg 1976
- Milanovic *et al.* 2004
- Mills 1995
- Milojicic *et al.* 1999
- Mitchell and Dion 1982
- Mitchell *et al.* 1992
- Mockapetris 1987
- Mogul 1994
- Mok 1985
- Morin 1997
- Morris *et al.* 1986
- McKusick, K. and Quinlan, S. (2010). GFS: Evolution or Fast-Forward. *Comms. ACM*, Vol. 53, No. 3, pp. 42-49.
- Meier, R. and Cahill, V. (2010), On event-based middleware for location-aware mobile applications. *IEEE Transactions on Software Engineering*, Vol. 36, No. 3, pp.09-430.
- Melliar-Smith, P., Moser, L. and Agrawala, V. (1990). Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, pp. 17-25.
- Menezes, A. (1993). *Elliptic Curve Public Key Cryptosystems*. Dordrecht, The Netherlands: Kluwer Academic Publishers.
- Menezes, A., van Oorschot, O. and Vanstone, S. (1997). *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press.
- Metcalfe, R.M. and Boggs, D.R. (1976). Ethernet: distributed packet switching for local computer networks. *Comms. ACM*, Vol. 19, No., pp. 395-403.
- Milanovic, N., Malek, M., Davidson, A. and Milutinovic, V. (2004). Routing and security in mobile ad hoc networks. *IEEE Computer*, Vol. 37, No. 2, pp. 69–73.
- Mills, D. (1995). Improved algorithms for synchronizing computer network clocks. *IEEE Transactions Networks*, Vol. 3, No. 3, pp. 245-54.
- Milojicic, J., Douglis, F. and Wheeler, R. (1999). *Mobility, Processes, Computers and Agents*. Reading, MA: Addison-Wesley.
- Mitchell, J.G. and Dion, J. (1982). A comparison of two network-based file servers. *Comms. ACM*, Vol. 25, No. 4, pp. 233–45.
- Mitchell, C.J., Piper, F. and Wild, P. (1992). Digital signatures. In Simmons, G.J. (ed.), *Contemporary Cryptology*. New York: IEEE Press.
- Mockapetris, P. (1987). *Domain names – concepts and facilities*. Internet RFC 1034.
- Mogul, J.D. (1994). Recovery in Spritely NFS. *Computing Systems*, Vol. 7, No. 2, pp. 201–62.
- Mok, A.K. (1985). SARTOR – A design environment for real-time systems. In *Proc. Ninth IEEE COMP-SAC*, Chicago, IL, Octobep, pp. 174–81.
- Morin, R. (ed.) (1997). *MkLinux: Microkernel Linux for the Power Macintosh*. Prime Time Freeware.
- Morris, J., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S. and Smith, F.D. (1986). Andrew: A distributed personal computing environment. *Comms. ACM*, Vol. 29, No. 3, pp. 184–201.

- Moser *et al.* 1994 Moser, L., Amir, Y., Melliar-Smith, P. and Agarwal, D. (1994). Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pp. 56–65.
- Moser *et al.* 1996 Moser, L., Melliar-Smith, P., Agarwal, D., Budhia, R. and Lingley-Papadopoulos, C. (1996). Totem: A fault-tolerant multicast group communication system. *Comms. ACM*, Vol. 39, No. 4, pp. 54–63.
- Moser *et al.* 1998 Moser, L., Melliar-Smith, P. and Narasimhan, P. (1998). Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, Vol. 4, No. 2, pp. 81–92.
- Moss 1985 Moss, E. (1985). *Nested Transactions, An Approach to Reliable Distributed Computing*. Cambridge, MA: MIT Press.
- Multimedia Directory Multimedia Directory (2005). Scala Inc.
- Murphy *et al.* 2001 Murphy, A.L., Picco, G.P. and Roman, G.-C. (2001). Lime: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, Phoenix, AZ, April, pp. 234–233.
- Muthitacharoen *et al.* 2002 Muthitacharoen, A., Morris, R., Gil, T.M. and Chen, B. (2002). Ivy: A read/write peer-to-peer file system. In *Proc. Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December, pp. 31–44.
- Muhl *et al.* 2006 Muhl, G., Fiege, L. and Pietzuch, P.R. (2006). *Distributed Event-based Systems*. Berlin, Heidelberg: Springer-Verlag.
- Myers and Liskov 1997 Myers, A.C. and Liskov, B. (1997). A decentralized model for information flow control, *ACM Operating Systems Review*, Vol. 31, No. 5, pp. 129–42.
- Nagle 1984 Nagle, J. (1984). Congestion control in TCP/IP internetworks, *Computer Communications Review*, Vol. 14, No. 4, pp. 11–17.
- Nagle 1987 Nagle, J. (1987). On packet switches with infinite storage. *IEEE Transactions on Communications*, Vol. 35, No. 4, pp. 435–8.
- National Bureau of Standards 1977 National Bureau of Standards (1977). *Data Encryption Standard (DES)*. Federal Information Processing Standards No. 46, Washington, DC: US National Bureau of Standards.
- nocr.sdsc.edu National Biomedical Computation Resource, University of California, San Diego.
- Needham 1993 Needham, R. (1993). Names. In Mullender, S. (ed.), *Distributed Systems, an Advanced Course*, 2nd edn. Wokingham, England: ACM Press/Addison-Wesley, pp. 315–26.

- Needham and Schroeder 1978
- Nelson *et al.* 1988
- Neuman *et al.* 1999
- Neumann and Ts'o 1994
- Newcomer 2002
- Nielsen and Thatte 2001
- Nielsen *et al.* 1997
- NIST 1994
- NIST 2002
- NIST 2004
- nms.csail.mit.edu
- Noble and Satyanarayanan 1999
- now.cs.berkeley.edu
- Oaks and Wong 1999
- Ohkubo *et al.* 2003
- Oki *et al.* 1993
- Needham, R.M. and Schroeder, M.D. (1978). Using encryption for authentication in large networks of computers. *Commun. ACM*, Vol. 21, No. 12, pp. 993–9.
- Nelson, M.N., Welch, B.B. and Ousterhout, J.K. (1988). Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pp. 134–154.
- Neuman, B.C., Tung, B. and Wray, J. (1999). *Public Key Cryptography for Initial Authentication in Kerberos*. Internet Draft ietf-cat-kerberos-pk-init-09.
- Neuman, B.C. and Ts'o, T. (1994). Kerberos: An authentication service for computer networks. *IEEE Communications*, Vol. 32, No. 9, pp. 33–38.
- Newcomer, E. (2002). *Understanding Web Services XML, WSDL, SOAP and UDDI*. Boston, MA: Pearson.
- Nielsen, H.F. and Thatte, S. (2001). *Web Services Routing Protocol (WS-Routing)*. Microsoft Corporation.
- Nielsen, H., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H. and Lilley, C. (1997). Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of the SIGCOMM '97*, pp. 155–66.
- National Institute for Standards and Technology (1994). *Digital Signature Standard*, NIST FIPS 186. US Department of Commerce.
- National Institute for Standards and Technology (2002). *Secure Hash Standard* NIST FIPS 180-2 + Change Notice to include SHA-224. US Department of Commerce.
- National Institute for Standards and Technology (2004). *NIST Brief Comments on Recent Cryptanalytic Attacks on Secure Hashing Functions and the Continued Security Provided by SHA-1*. US Department of Commerce.
- The Berkeley RON project. *Home page*.
- Noble, B. and Satyanarayanan, M. (1999). Experience with adaptive mobile applications in Odyssey. *Mobile Networks and Applications*, Vol. 4, No. 4, pp. 245–254.
- The Berkeley NOW project. *Home page*.
- Oaks, S. and Wong, H. (1999). *Java Threads*, 2nd edn. Sebastopol, CA: O'Reilly.
- Ohkubo, M., Suzuki, K. and Kinoshita, S. (2003). Cryptographic approach to ‘privacy-friendly’ tags. In *Proceedings of the RFID Privacy Workshop*, MIT.
- Oki, B., Pfluegl, M., Siegel, A., and Skeen, D. (1993). The Information Bus: an architecture for extensible distributed systems. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, NC, December, NY, pp. 58–68.

- Olson and Ogbuji 2002 Olson, M. and Ogbuji, U. (2002). *The Python Web services developer: Messaging technologies compared – Choose the best tool for the task at hand*. IBM DeveloperWorks
- OMG 2000a Object Management Group (2000). *Trading Object Service Specification*, Vn. 1.0. Needham, MA: OMG.
- OMG 2000b Object Management Group (2000). *Concurrency Control Service Specification*. Needham, MA: OMG.
- OMG 2002a Object Management Group (2002). *The CORBA IDL Specification*. Needham, MA: OMG.
- OMG 2002b Object Management Group (2002). *CORBA Security Service Specification* Vn. 1.8. Needham, MA: OMG.
- OMG 2002c Object Management Group (2002). *Value Type Semantics*. Needham, MA: OMG.
- OMG 2002d Object Management Group (2002). *Life Cycle Service*, Vn. 1.2. Needham, MA: OMG.
- OMG 2002e Object Management Group (2002). *Persistent State Service*, Vn. 2.0. Needham, MA: OMG.
- OMG 2003 Object Management Group, (2003). *Object Transaction Service Specification*, Vn. 1.4. Needham, MA: OMG.
- OMG 2004a Object Management Group (2004). *CORBA/IOP 3.0.3 Specification*. Needham, MA: OMG.
- OMG 2004b Object Management Group (2004). *Naming Service Specification*. Needham, MA: OMG.
- OMG 2004c Object Management Group (2004). *Event Service Specification*, Vn. 1.2. Needham, MA: OMG.
- OMG 2004d Object Management Group (2004). *Notification Service Specification*, Vn. 1.1. Needham, MA: OMG. Technical report telecom/98-06-15.
- OMG 2004e Object Management Group (2004). *CORBA Messaging*. Needham, MA: OMG.
- Omidyar and Aldridge 1993 Omidyar, C.G. and Aldridge, A. (1993). Introduction to SDH/SONET. *IEEE Communications Magazine*, Vol. 31, pp. 30–3.
- open.eucalyptus.com Eucalyptus. *Home page*.
- OpenNap 2001 OpenNap: Open Source Napster Server, Beta release 0.44, September 2001.
- Oppen and Dalal 1983 Oppen, D.C. and Dalal Y.K. (1983). The Clearinghouse: a decentralized agent for locating named objects in a distributed environment. *ACM Trans. on Office Systems*, Vol. 1, No. 3, pp. 230–53.
- Oram 2001 Oram, A. (2001). *Peer-to-Peer: Harnessing the Benefits of Disruptive Technologies*, O'Reilly, Sebastopol, CA.

- Orfali *et al.* 1996 Orfali, R., Harkey, D. and Edwards, J. (1996). *The Essential Distributed Objects Survival Guide*. New York: Wiley.
- Orfali *et al.* 1997 Orfali, R., Harkey, D., and Edwards, J. (1997) *Instant CORBA*. New York: John Wiley & Sons.
- Organick 1972 Organick, E.I. (1972). *The MULTICS System: An Examination of its Structure*. Cambridge, MA: MIT Press.
- Orman *et al.* 1993 Orman, H., Menze, E., O'Malley, S. and Peterson, L. (1993). A fast and general implementation of Mach IPC in a Network. In *Proceedings of the Third USENIX Mach Conference*, April.
- OSF *Introduction to OSF DCE*. The Open Group.
- Ousterhout *et al.* 1985 Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M. and Thompson, J. (1985). A Trace-driven analysis of the UNIX 4.2 BSD file system. In *Proc. of the 10th ACM Symposium Operating System Principles*, p. 15–24.
- Ousterhout *et al.* 1988 Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M. and Welch, B. (1988). The Sprite network operating system. *IEEE Computer*, Vol. 21, No. 2, pp. 23–36.
- Parker 1992 Parker, B.(1992). *The PPP AppleTalk Control Protocol (ATCP)*. Internet RFC 1378.
- Parrington *et al.* 1995 Parrington, G.D., Shrivastava, S.K., Wheater, S.M. and Little, M.C. (1995). The design and implementation of Arjuna. *USENIX Computing Systems Journal*, Vol. 8, No. 3, pp. 255–308.
- Partridge 1992 Partridge, C. (1992). *A Proposed Flow Specification*. Internet RFC 1363.
- Patel and Abowd 2003 Patel, S.N. and Abowd, G.D. (2003). A 2-way laser-assisted selection scheme for handhelds in a physical environment. In *Proceedings of the Fifth International Conference on Ubiquitous Computing (Ubicomp 2003)*, Seattle, WA, October, pp. 200–207.
- Patterson *et al.* 1988 Patterson, D., Gibson, G. and Katz, R. (1988). A case for redundant arrays of interactive disks. *ACM International Conf. on Management of Data (SIGMOD)*, pp. 109–116.
- Pease *et al.* 1980 Pease, M., Shostak, R. and Lamport, L. (1980). Reaching agreement in the presence of faults. *Journal of the ACM*, Vol. 27, No. 2, pp. 228–34.
- Pedone and Schiper 1999 Pedone, F. and Schiper, A. (1999). Generic broadcast. In *Proceeding of the 13th International Symposium on Distributed Computing (DISC '99)*, September, pp. 94–108.
- Peng and Dabek 2010 Peng, D. and Dabek, F. (2010). Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the Ninth Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, October, pp. 1–15.

- Perrig *et al.* 2002 Perrig, A., Szewczyk, R., Wen, V., Culler, D. and Tygar, D. (2002). SPINS: Security protocols for sensor networks. *Wireless Networks*, Vol. 8, No. 5, pp. 521–534.
- Peterson 1988 Peterson, L. (1988). The Profile Naming Service. *ACM Transactions on Computer Systems*, Vol. 6, No. 4, pp. 341–64.
- Peterson *et al.* 1989 Peterson, L.L., Buchholz, N.C. and Schlichting, R.D. (1989). Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, Vol. 7, No. 3, pp. 217–46.
- Petersen *et al.* 1997 Petersen, K., Spreitzer, M., Terry, D., Theimer, M. and Demers, A. (1997). Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 288–301.
- Peterson *et al.* 2005 Peterson, L.L., Shenker, S. and Turner, J. (2005). Overcoming the Internet impasse through virtualization. *Computer*, Vol. 38, No. 4, pp. 34–41.
- Pietzuch and Bacon 2002 Pietzuch, P. R. and Bacon, J. (2002). Hermes: A distributed event-based middleware architecture. In *Proceedings of the First International Workshop on Distributed Event-Based Systems*, Vienna, Austria, pp. 611–618.
- Pike *et al.* 1993 Pike, R., Presotto, D., Thompson, K., Trickey, H. and Winterbottora, P. (1993). The use of name spaces in Plan 9. *Operating Systems Review*, Vol. 27, No. 2, pp. 72–76.
- Pike *et al.* 2005 Pike, R., Dorward, S., Griesemer, R. and Quinlan, S. (2005). Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, Vol. 13, No. 4, pp. 277–298.
- Pinheiro *et al.* 2007 Pinheiro, E., Weber, W.D. and Barroso, L.A. (2007). Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, pp. 17–28.
- Plaxton *et al.* 1997 Plaxton, C.G., Rajaraman, R. and Richa, A.W. (1997). Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of the ACM Symposium on Parallel Algorithms and Architectures*, pp. 311–320.
- Ponnekanti and Fox 2004 Ponnekanti, S. and Fox, A. (2004). Interoperability among independently evolving web services. In *Proceedings of the ACM/Usenix/IFIP 5th International Middleware Conference*, Toronto, Canada, pp. 331–57.
- Ponnekanti *et al.* 2001 Ponnekanti, S.R., Lee, B., Fox, A., Hanrahan, P. and Winograd, T. (2001). ICrafter: A service framework for ubiquitous computing environments. In *Proceedings of the Third International Conference on Ubiquitous Computing (Ubicomp 2001)*, Atlanta, GA, Sep.–Oct., pp. 56–75.
- Popek and Goldberg 1974 Popek, G.J. and Goldberg, R.P. (1974). Formal requirements for virtualizable third generation architectures. *Comms. ACM*, Vol. 17, No. 7, pp. 412–421.

- Popek and Walker 1985 Popek, G. and Walker, B. (eds.) (1985). *The LOCUS Distributed System Architecture*. Cambridge, MA: MIT Press.
- Postel 1981a Postel, J. (1981). *Internet Protocol*. Internet RFC 791.
- Postel 1981b Postel, J. (1981). *Transmission Control Protocol*. Internet RFC 793.
- Pottie and Kaiser 2000 Pottie, G.J. and Kaiser, W.J. (2000). Embedding the Internet: Wireless integrated network sensors. *Comms. ACM*, Vol. 43, No. 5, pp. 51–58.
- Powell 1991 Powell, D. (ed.) (1991). *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Berlin and New York: Springer-Verlag.
- Pradhan and Chiueh 1998 Pradhan, P. and Chiueh, T. (1998). Real-time performance guarantees over wired and wireless LANS. In *IEEE Conference on Real-Time Applications and Systems*, RTAS'98, June, p. 29.
- Prakash and Baldoni 1998 Prakash, R. and Baldoni, R. (1998). Architecture for group communication in mobile systems. In *Proceedings of the the 17th IEEE Symposium on Reliable Distributed Systems*, Washington, DC, pp. 235–242.
- Preneel *et al.* 1998 Preneel, B., Rijmen, V. and Bosselaers, A. (1998). Recent developments in the design of conventional cryptographic algorithms. In *Computer Security and Industrial Cryptography, State of the Art and Evolution*. Vol. 1528 of Lecture Notes in Computer Science, Springer-Verlag, pp. 106–131.
- privacy.nb.ca *International Cryptography Freedom.*
- Radia *et al.* 1993 Radia, S., Nelson, M. and Powell, M. (1993). *The Spring Naming Service*. Technical Report 93–16, Sun Microsystems Laboratories, Inc.
- Raman and McCanne 1999 Raman, S. and McCanne, S. (1999). A model, analysis, and protocol framework for soft state-based communication. In *Proceedings of the ACM SIGCOMM*, 1999, Cambridge, MA, pp. 15–25.
- Randall and Szydlo 2004 Randall, J. and Szydlo, M. (2004). Collisions for SHA0, MD5, HAVAL, MD4, and RIPEMD, but SHA1 still secure. *RSA Laboratories Technical Note*, August 31.
- Rashid 1985 Rashid, R.F. (1985). Network operating systems. In *Local Area Networks: An Advanced Course, Lecture Notes in Computer Science*, 184, Springer-Verlag, pp. 314–40.
- Rashid 1986 Rashid, R.F. (1986). From RIG to Accent to Mach: the evolution of a network operating system. In *Proceedings of the ACM/IEEE Computer Society Fall Joint Conference*, ACM, November.

- Rashid and Robertson 1981
- Rashid *et al.* 1988
- Ratnasamy *et al.* 2001
- Raynal 1988
- Raynal 1992
- Raynal and Singhal 1996
- Redmond 1997
- Reed 1983
- Rellermeyer *et al.* 2007
- Rescorla 1999
- research.microsoft.com
- Rether
- Rhea *et al.* 2001
- Rhea *et al.* 2003
- Ricart and Agrawala 1981
- Rashid, R. and Robertson, G. (1981). Accent: a communications oriented network operating system kernel. *ACM Operating Systems Review*, Vol. 15, No. 5, pp. 64–75.
- Rashid, R., Tevanian Jr, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W.J. and Chew, J. (1988). Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, Vol. 37, No. 8, pp. 896–907.
- Ratnasamy, S., Francis, P., Handley, M., Karp, R. and Shenker, S. (2001). A scalable content-addressable network. In *Proc. ACM SIGCOMM 2001*, August, pp. 161–72.
- Raynal, M. (1988). *Distributed Algorithms and Protocols*. New York: John Wiley & Sons.
- Raynal, M. (1992). About logical clocks for distributed systems. *ACM Operating Systems Review*, Vol. 26, No. 1, pp. 41–8.
- Raynal, M. and Singhal, M. (1996). Logical time: Capturing causality in distributed systems. *IEEE Computer*, Vol. 29, No. 2, pp. 49–56.
- Redmond, F.E. (1997). *DCOM: Microsoft Distributed Component Model*. IDG Books Worldwide.
- Reed, D.P. (1983). Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, Vol. 1, No. 1, pp. 3–23.
- Rellermeyer, J. S., Alonso, G., and Roscoe, T. (2007). R-OSGi: Distributed applications through software modularization. In *Proceedings of the ACM/IFIP/USENIX 2007 international Conference on Middleware*, Newport Beach, CA, November, pp. 1–20.
- Rescorla, E. (1999). *Diffie-Hellman Key Agreement Method*. Internet RFC 2631.
- Microsoft Research. *Writings of Leslie Lamport*.
- Rether: A Real-Time Ethernet Protocol.
- Rhea, S., Wells, C., Eaton, P., Geels, D., Zhao, B., Weatherspoon, H. and Kubiatowicz, J. (2001). Maintenance-free global data storage. *IEEE Internet Computing*, Vol. 5, No. 5, pp. 40–49.
- Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B. and Kubiatowicz, J. (2003). Pond: The OceanStore prototype, In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pp. 1–14.
- Ricart, G. and Agrawala, A.K. (1981). An optimal algorithm for mutual exclusion in computer networks. *Comms. ACM*, Vol. 24, No. 1, pp. 9–17.

- Richardson *et al.* 1998
- Ritchie 1984
- Rivest 1992a
- Rivest 1992b
- Rivest *et al.* 1978
- Rodrigues *et al.* 1998
- Roman *et al.* 2001
- Rose 1992
- Rosenblum and Ousterhout 1992
- Rosenblum and Wolf 1997
- Rowley 1998
- Rowstron and Druschel 2001
- Rowstron and Wood 1996
- royal.pingdom.com
- Richardson, T., Stafford-Fraser, Q., Wood, K.R. and Hopper, A. (1998). Virtual network computing, *IEEE Internet Computing*, Vol. 2, No. 1, pp. 33–8.
- Ritchie, D. (1984). A Stream Input Output System. *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, pt 2, pp. 1897–910.
- Rivest, R. (1992). *The MD5 Message-Digest Algorithm*. Internet RFC 1321.
- Rivest, R. (1992). *The RC4 Encryption Algorithm*. RSA Data Security Inc.
- Rivest, R.L., Shamir, A. and Adelman, L. (1978). A method of obtaining digital signatures and public key cryptosystems. *Comms. ACM*, Vol. 21, No. 2, pp. 120–6.
- Rodrigues, L., Guerraoui, R. and Schiper, A. (1998). Scalable atomic multicast. In *Proceedings IEEE IC3N '98*. Technical Report 98/257. École polytechnique fédérale de Lausanne.
- Roman, G., Huang, Q. and Hazemi, A. (2001). Consistent group membership in ad hoc networks. In *Proceedings of the 23rd International Conference on Software Engineering*, Washington, DC, pp. 381–388.
- Rose, M.T. (1992). *The Little Black Book: Mail Bonding with OSI Directory Services*. Englewood Cliffs, NJ: Prentice-Hall.
- Rosenblum, M. and Ousterhout, J. (1992). The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, Vol. 10, No. 1, pp. 26–52.
- Rosenblum, D.S. and Wolf, A.L. (1997). A design framework for Internet-scale event observation and notification. In *Proceedings of the sixth European Software Engineering Conference/ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, pp. 344–60.
- Rowley, A. (1998). *A Security Architecture for Groupware*. Doctoral Thesis, Queen Mary and Westfield College, University of London.
- Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001*, Heidelberg, Germany, Nov., pp. 329–50.
- Rowstron, A. and Wood, A. (1996). An efficient distributed tuple space implementation for networks of workstations. In *Proceedings of the Second International Euro-Par Conference*, Lyon, France, pp. 510–513.
- Royal Pingdom. *Map of all Google data centre locations as of 2008*.

- Rozier *et al.* 1988 Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Leonard, P. and Neuhauser, W. (1988). Chorus distributed operating systems. *Computing Systems Journal*, Vol. 1, No. 4, pp. 305–70.
- Rozier *et al.* 1990 Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrman, F., Kaiser, C., Langlois, S., Leonard, P. and Neuhauser, W. (1990). *Overview of the Chorus Distributed Operating System*. Technical Report CS/TR-90-25.1, Chorus Systèmes, France.
- RTnet RTnet: Hard Real-Time Networking for Linux/RTAI.
- Salber *et al.* 1999 Salber, D., Dey, A.K. and Abowd, G.D. (1999). The Context Toolkit: Aiding the development of context-enabled applications. In *Proceedings of the 1999 Conference on Human Factors in Computing Systems (CHI '99)*, Pittsburgh, PA, May, pp. 434–441.
- Saltzer *et al.* 1984 Saltzer, J.H., Reed, D.P. and Clarke, D. (1984). End-to-end arguments in system design. *ACM Transactions on Computer Systems*, Vol. 2, No. 4, pp. 277–88.
- Sandberg 1987 Sandberg, R. (1987). *The Sun Network File System: Design, Implementation and Experience*. Technical Report. Mountain View, CA: Sun Microsystems.
- Sandberg *et al.* 1985 Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D. and Lyon, B. (1985). The design and implementation of the Sun Network File System. In *Proceedings of the Usenix Conference*, Portland, OR, p. 119.
- Sanders 1987 Sanders, B. (1987). The information structure of distributed mutual exclusion algorithms. *ACM Transactions on Computer Systems*, Vol. 5, No. 3, pp. 284–99.
- Sandholm and Gawor 2003 Sandholm, T. and Gawor, J. (2003). *Globus Toolkit 3 Core – A Grid Service Container Framework*. July.
- Sandhu *et al.* 1996 Sandhu, R., Coyne, E., Felstein, H. and Youman, C. (1996). Role-based access control models. *IEEE Computer*, Vol. 29, No. 2, pp. 38–37.
- Sansom *et al.* 1986 Sansom, R.D., Julin, D.P. and Rashid, R.F. (1986). *Extending a capability based system into a network environment*. Technical Report CMU-CS-86-116, Carnegie-Mellon University.
- Santifaller 1991 Santifaller, M. (1991). *TCP/IP and NFS, Internetworking in a Unix Environment*. Reading, MA: Addison-Wesley.
- Saroiu *et al.* 2002 Saroiu, S., Gummadi, P. and Gribble, S. (2002). A measurement study of peer-to-peer file sharing systems. In *Proc. Multimedia Computing and Networking (MMCN)*, pp. 156–70.

- Sastry *et al.* 2003 Sastry, N., Shankar, U. and Wagner, D. (2003). Secure verification of location claims. In *Proceedings of the ACM Workshop on Wireless Security (WiSe 2003)*, September, pp. 1–10.
- Satyanarayanan 1981 Satyanarayanan, M. (1981). A study of file sizes and functional lifetimes. In *Proceedings of the 8th ACM Symposium on Operating System Principles*, Asilomar, CA, pp. 96–108.
- Satyanarayanan 1989a Satyanarayanan, M. (1989). Distributed File Systems. In Mullender, S. (ed.), *Distributed Systems, an Advanced Course*, 2nd edn. Wokingham, England: ACM Press/Addison-Wesley, pp. 353–83.
- Satyanarayanan 1989b Satyanarayanan, M. (1989). Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, Vol. 7, No. 3, pp. 247–80.
- Satyanarayanan 2001 Satyanarayanan, M. (2001). Pervasive computing; Vision and challenges. *IEEE Personal Communications*, Vol. 8, No. 4, pp. 10–17.
- Satyanarayanan *et al.* 1990 Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H. and Steere, D.C. (1990). Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, Vol. 39, No. 4, pp. 447–59.
- Schilit *et al.* 1994 Schilit, B.N., Adams, N.I. and Want, R. (1994). Context-aware computing applications. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December, pp. 85–90.
- Schiper and Raynal 1996 Schiper, A. and Raynal, M. (1996). From group communication to transactions in distributed systems. *Comms. ACM*, Vol. 39, No. 4, pp. 84–7.
- Schiper and Sandoz 1993 Schiper, A. and Sandoz, A. (1993). Uniform reliable multicast in a virtually synchronous environment. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pp. 561–8.
- Schlageter 1982 Schlageter, G. (1982). Problems of optimistic concurrency control in distributed database systems. *SigMOD Record*, Vol. 13, No. 3, pp. 62–6.
- Schmidt 1998 Schmidt, D. (1998). Evaluating architectures for multithreaded object request brokers. *Comms. ACM*, Vol. 44, No. 10, pp. 54–60.
- Schneider 1990 Schneider, F.B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, Vol. 22, No. 4, pp. 300–19.
- Schneider 1996 Schneider, S. (1996). Security properties and CSP. In *IEEE Symposium, on Security and Privacy*, pp. 174–187.

- Schneier 1996 Schneier, B. (1996). *Applied Cryptography*, 2nd edn. New York: John Wiley & Sons.
- Schroeder and Burrows 1990 Schroeder, M. and Burrows, M. (1990). The performance of Firefly RFC. *ACM Transactions on Computer Systems*, Vol. 8, No. ., pp. 1–17.
- Schroeder *et al.* 1984 Schroeder, M.D., Birrell, A.D. and Needham, R.M. (1984). Experience with Grapevine: The growth of a distributed system, *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 3–23.
- Schulzrinne *et al.* 1996 Schulzrinne, H., Casner, S., Frederick, D. and Jacobson, V. (1996). *RTP: A Transport Protocol for Real-Time Applications*. Internet RFC 1889.
- sector.sourceforge.net Sector/Sphere. *Home page*.
- Seetharamanan 1998 Seetharamanan, K. (ed.) (1998). Special Issue: The CORBA Connection. *Comms. ACM*, Vol. 41, No. 10.
- session directory *User Guide to sd (Session Directory)*.
- Shannon 1949 Shannon, C.E. (1949). Communication theory of secrecy systems. *Bell System Technical Journal*, Vol. 28, No. 4, pp. 656–715.
- Shepler 1999 Shepler, S. (1999). *NFS Version 4 Design Considerations*. Internet RFC 2624, Sun Microsystems.
- Shih *et al.* 2002 Shih, E., Bahl, P. and Sinclair, M. (2002). Wake on Wireless: An event driven energy saving strategy for battery operated devices. In *Proceedings of the Eighth Annual ACM Conference on Mobile Computing and Networking*, Atlanta, GA, September, pp. 160–171.
- Shirky 2000 Shirky, C. (2000). What's P2P and what's not, 11/24/2000. Internet publication.
- Shoch and Hupp 1980 Shoch, J.F. and Hupp, J.A. (1980). Measured performance of an Ethernet local network. *Comms. ACM*, Vol. 23, No. 12, pp. 711–21.
- Shoch and Hupp 1982 Shoch, J.F. and Hupp, J.A. (1982). The ‘Worm’ programs – early experience with a distributed computation. *Comms. ACM*, Vol. 25, No. 3, pp. 172–80.
- Shoch *et al.* 1982 Shoch, J.F., Dalal, Y.K. and Redell, D.D. (1982). The evolution of the Ethernet local area network. *IEEE Computer*, Vol. 15, No. 8, pp. 10–28.
- Shoch *et al.* 1985 Shoch, J.F., Dalal, Y.K., Redell, D.D. and Crane, R.C. (1985). The Ethernet. In *Local Area Networks: An Advanced Course. Vol 184 of Lecture Notes in Computer Science*. Springer-Verlag, pp. 1–33.
- Shrivastava *et al.* 1991 Shrivastava, S., Dixon, G.N. and Parrington, G.D. (1991). An overview of the Arjuna distributed programming system. *IEEE Software*, pp. 66–73.
- Singh 1999 Singh, S. (1999). *The Code Book*. London: Fourth Estate.

- Sinha and Natarajan 1985 Sinha, M. and Natarajan, N. (1985). A priority based distributed deadlock detection algorithm. *IEEE Transactions on Software Engineering*. Vol. 11, No. 1, pp. 67–80.
- Sirivianos *et al.* 2007 Sirivianos, M., Park, J.H., Chen R. and Yang, X. (2007). Free-riding in BitTorrent networks with the large view exploit. In *Proceedings of the 6th International Workshop on Peer-to-Peer Systems (IPTPS '07)*, Bellevue, WA.
- Spafford 1989 Spafford, E.H. (1989). The Internet worm: Crisis and aftermath. *Comms. ACM*, Vol. 32, No. 6, pp. 678–87.
- Spasojevic and Satyanarayanan 1996 Spasojevic, M. and Satyanarayanan, M. (1996). An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems*, Vol. 14, No. 2, pp. 200–222.
- Spector 1982 Spector, A.Z. (1982). Performing remote operations efficiently on a local computer network. *Comms. ACM*, Vol. 25, No. 4, pp. 246–60.
- Spurgeon 2000 Spurgeon, C.E. (2000). *Ethernet: The Definitive Guide*. Sebastopol, CA: O'Reilly.
- Srikanth and Toueg 1987 Srikanth, T. and Toueg, S. (1987). Optimal clock synchronization. *Journal ACM*. Vol. 34, No. 3, pp. 626–45.
- Srinivasan 1995a Srinivasan, R. (1995). *RPC: Remote Procedure Call Protocol Specification Version 2*. Sun Microsystems. Internet RFC 1831. August.
- Srinivasan 1995b Srinivasan, R. (1995). *XDR: External Data Representation Standard*. Sun Microsystems. Internet RFC 1832. Network Working Group.
- Srinivasan and Mogul 1989 Srinivasan, R. and Mogul, J.D. (1989). Spritely NFS: Experiments with cache-consistency protocols. In *Proc. of the 12th ACM Symposium on Operating System Principles*, Litchfield Park, AZ, December, pp. 45–57.
- Srisuresh and Holdrege 1999 Srisuresh, P. and Holdrege, M. (1999). *IP Network Address Translator (NAT) Terminology and Considerations*. Internet RFC 2663.
- Stajano 2002 Stajano, F. (2002). *Security for Ubiquitous Computing*. New York: John Wiley & Sons.
- Stajano and Anderson 1999 Stajano, F. and Anderson, R. (1999). The resurrecting duckling: Security issues for adhoc wireless networks. In *Proceedings of the 7th International Workshop on Security Protocols*, pp. 172–194.
- Stallings 2002 Stallings, W. (2002). *High Speed Networks – TCP/IP and ATM Design Principles*. 2nd edn. Upper Saddle River, NJ: Prentice-Hall.
- Stallings 2005 Stallings, W. (2005). *Cryptography and Network Security – Principles and Practice*, 4th edn. Upper Saddle River, NJ: Prentice-Hall.

- Stallings 2008 Stallings, W. (2008). *Operating Systems*, 6th edn. Upper Saddle River, NJ: Prentice-Hall International.
- Steiner *et al.* 1988 Steiner, J., Neuman, C. and Schiller, J. (1988). Kerberos: An authentication service for open network systems. In *Proceedings of the Usenix Winter Conference*, Berkeley, CA.
- Stelling *et al.* 1998 Stelling, P., Foster, I., Kesselman, C., Lee, C. and von Laszewski, G. (1998). A fault detection service for wide area distributed computations. In *Proceedings of the 7th IEEE Symposium on High Performance Distributed Computing*, pp. 268–78.
- Stoica *et al.* 2001 Stoica, I., Morris, R., Karger, D., Kaashoek, F. and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM '01*, August, pp. 149–60.
- Stojmenovic 2002 Stojmenovic, I. (ed.) (2002). *Handbook of Wireless Networks and Mobile Computing*. New York: John Wiley & Sons.
- Stoll 1989 Stoll, C. (1989). *The Cuckoo's Egg: Tracking a Spy Through a Maze of Computer Espionage*. New York: Doubleday.
- Stone 1993 Stone, H. (1993). *High-performance Computer Architecture*, 3rd edn. Reading, MA: Addison-Wesley.
- Stonebraker *et al.* 2010 Stonebraker, M., Abadi, D., DeWitt, D. J., Madden, S., Paulson, E., Pavlo, A. and Rasin, A. (2010). MapReduce and parallel DBMSs: Friends or foes? *Comms. ACM*, Vol. 53, No. 1, pp. 64–71.
- Sun 1989 Sun Microsystems Inc. (1989). *NFS: Network File System Protocol Specification*. Internet RFC 1094.
- Sun and Ellis 1998 Sun, C. and Ellis, C. (1998). Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the Conference on Computer Supported Cooperative Work Systems*, pp. 59–68.
- SWAP-CA 2002 Shared Wireless Access Protocol (Cordless Access) Specification (SWAP-CA), Revision 2.0.1. The HomeRF Technical Committee, July 2002.
- Szalay and Gray 2001 Szalay, A. and Gray, J. (2001) The World-Wide Telescope. *Science*, Vol. 293, No. 5537, pp. 2037–2040.
- Szalay and Gray 2004 Szalay, A. and Gray, J. (2004). *Scientific Data Federation: The World-Wide Telescope*. In Foster, I. and Kesselman, C. (eds.), *The Grid 2*. San Francisco, CA: Morgan Kauffman.
- Szyperski 2002 Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*, 2nd edn. Reading, MA: Addison-Wesley.
- Tanenbaum 2003 Tanenbaum, A.S. (2003). *Computer Networks*, 4th edn. Upper Saddle River, NJ: Prentice-Hall International.
- Tanenbaum 2007 Tanenbaum, A.S. (2007). *Modern Operating Systems*, 3rd edn. Englewood Cliffs, NJ: Prentice-Hall.

- Tanenbaum and van Renesse 1985 Tanenbaum, A. and van Renesse, R. (1985). Distributed operating systems. *Computing Surveys, ACM*, Vol. 17, No. 4, pp. 419–70.
- Tanenbaum *et al.* 1990 Tanenbaum, A.S., van Renesse, R., van Staveren, H., Sharp, G., Mullender, S., Jansen, J. and van Rossum, G. (1990). Experiences with the Amoeba distributed operating system. *Comms. ACM*, Vol. 33, No. 12, pp. 46–63.
- Taufer *et al.* 2003 Taufer, M., Crowley, M., Karanicolas, J., Cicotti, P., Chien, A. and Brooks, L. (2003). *Moving Towards Desktop Grid Solutions for Large Scale Modelling in Computational Chemistry*. University of California, San Diego.
- Terry *et al.* 1995 Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M. and Hauser, C. (1995). Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 172–183.
- TFCC IEEE Task Force on Cluster Computing.
- Thaler *et al.* 2000 Thaler, D., Handley, M. and Estrin, D. (2000). *The Internet Multicast Address Allocation Architecture*. Internet RFC 2908.
- Thekkath *et al.* 1997 Thekkath, C.A., Mann, T. and Lee, E.K. (1997). Frangipani: A scalable distributed file system, in *Proc. of the 16th ACM Symposium on Operating System Principles*, St. Malo, France, October, pp. 224–237.
- Tokuda *et al.* 1990 Tokuda, H., Nakajima, T. and Rao, P. (1990). Real-time Mach: Towards a predictable real-time system. In *Proceedings of the USENIX Mach Workshop*, pp. 73–82.
- Topolcic 1990 Topolcic, C. (ed.) (1990). *Experimental Internet Stream Protocol, Version 2*. Internet RFC 1190.
- Tsoumakos and Roussopoulos 2006 Tsoumakos, D. and Roussopoulos, N. (2006). Analysis and comparison of P2P search methods. In *Proceedings of the 1st international Conference on Scalable information Systems (InfoScale '06)*, Hong Kong, p.25.
- Tzou and Anderson 1991 Tzou, S.-Y. and Anderson, D. (1991). The performance of message-passing using restricted virtual memory remapping. *Software-Practice and Experience*, Vol. 21, pp. 251–67.
- van Renesse *et al.* 1989 van Renesse, R., van Staveren, H. and Tanenbaum, A. (1989). The performance of the Amoeba distributed operating system. *Software – Practice and Experience*, Vol. 19, No. 3, pp. 223–34.
- van Renesse *et al.* 1995 van Renesse, R., Birman, K., Friedman, R., Hayden, M. and Karr, D. (1995). A framework for protocol composition in Horus. In *Proceedings of the PODC 1995*, pp. 80–9.
- van Renesse *et al.* 1996 van Renesse, R., Birman, K. and Maffeis, S. (1996). Horus: A flexible group communication system. *Comms. ACM*, Vol. 39, No. 4, pp. 54–63.

- van Renesse *et al.* 1998 van Renesse, R., Birman, K., Hayden, M., Vaysburd, A. and Karr, D. (1998). Building adaptive systems using Ensemble. *Software—Practice and Experience*, Vol. 28, No. 9, pp. 963–979.
- van Steen *et al.* 1998 van Steen, M., Hauck, F., Homburg, P. and Tanenbaum, A. (1998). Locating objects in wide-area systems. *IEEE Communication*, Vol. 36, No. 1, pp. 104–109.
- Vinoski 1998 Vinoski, S. (1998). New features for CORBA 3.0. *Comms. ACM*, Vol. 41, No. 10, pp. 44–52.
- Vinoski 2002 Vinoski, S. (2002). Putting the ‘Web’ into Web Services. *IEEE Internet Computing*, Vol. 6, No. 4, pp. 90–92.
- Vogels 2003 Vogels, W. (2003). Web Services are not Distributed objects. *IEEE Internet Computing*, Vol. 7, No. 6, pp. 59–66.
- Vogt *et al.* 1993 Vogt, C., Herrtwich, R.G. and Nagarajan, R. (1993). HeiRAT – The Heidelberg Resource Administration Technique: Design Philosophy and Goals. *Kommunikation in verteilten Systemen*, Munich, Informatik aktuell, Springer.
- Volpano and Smith 1999 Volpano, D. and Smith, G. (1999). Language issues in mobile program security. In *Mobile Agents and Security*. Vol. 1419 in *Lecture Notes in Computer Science*, Springer-Verlag, pp. 25–43.
- von Eicken *et al.* 1995 von Eicken, T., Basu, A., Buch, V. and Vogels, V. (1995). U-Net: A user-level network interface for parallel and distributed programming. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 40–53.
- Wahl *et al.* 1997 Wahl, M., Howes, T. and Kille, S. (1997). *The Lightweight Directory Access Protocol (v3)*. Internet RFC 2251.
- Waldo 1999 Waldo, J. (1999). The Jini architecture for network-centric computing. *Comms. ACM*, Vol. 42 No. 7, pps. 76–82.
- Waldo *et al.* 1994 Waldo, J., Wyant, G., Wollrath, A. and Kendall, S. (1994). A note on distributed computing. In Arnold *et al.* 1999, pp. 307–26.
- Waldspurger *et al.* 1992 Waldspurger, C., Hogg, T., Huberman, B., Kephart, J. and Stornetta, W. (1992). Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, Vol. 18, No. 2, pp. 103–17.
- Wang *et al.* 2001 Wang, N., Schmidt, D. C. and O’Ryan, C. (2001). Overview of the CORBA component model. In Heineman, G. T. and Councill, W.T. (eds), *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Boston, MA: Addison-Wesley, pp. 557–571.
- Want 2004 Want, R. (2004). Enabling ubiquitous sensing with RFID. *IEEE Computer*, Vol. 37, No. 4, pp. 84–86.

- Want and Pering 2003 Want, R. and Pering, T. (2003). New horizons for mobile computing. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communication (PerCom '03)*, Dallas-Fort Worth, TX, March, pp. 3–8.
- Want *et al.* 1992 Want, R., Hopper, A., Falcao, V. and Gibbons, V. (1992). The Active Badge location system. *ACM Transactions on Information Systems*, Vol. 10, No. 1, pp. 91–102.
- Want *et al.* 2002 Want, R., Pering, T., Danneels, G., Kumar, M., Sundar, M. and Light, J. (2002). The personal server: Changing the way we think about ubiquitous computing. In *Proceedings of the Fourth International Conference on Ubiquitous Computing (Ubicomp 2002)*, Goteborg, Sweden, Sep.–Oct., pp. 194–209.
- Weatherspoon and Kubiatowicz 2002 Weatherspoon, H. and Kubiatowicz, J.D. (2002). Erasure coding vs. replication: A quantitative comparison. *1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, March, pp. 328–38.
- web.mit.edu I *Kerberos: The Network Authentication Protocol.*
- web.mit.edu II *The Three Myths of Firewalls.*
- Wegner 1987 Wegner, P. (1987). Dimensions of object-based language design. *SIGPLAN Notices*, Vol. 22, No. 12, pp. 168–182.
- Weikum 1991 Weikum, G. (1991). Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, Vol. 16, No. 1, pp. 132–40.
- Weiser 1991 Weiser, M. (1991). The computer for the 21st Century. *Scientific American*, Vol. 265, No. 3, pp. 94–104.
- Weiser 1993 Weiser, M. (1993). Some computer science issues in ubiquitous computing. *Comms, ACM*, Vol. 36, No. 7, pp. 74–84.
- Wellner 1991 Wellner, P.D. (1991). The DigitalDesk calculator – tangible manipulation on a desk-top display. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, Hilton Head, SC, November, pp. 27–33.
- Wheeler and Needham 1994 Wheeler, D.J. and Needham, R.M. (1994). TEA, a Tiny Encryption Algorithm. Technical Report 355, *Two Cryptographic Notes*, Computer Laboratory, University of Cambridge, December, pp. 1–3.
- Wheeler and Needham 1997 Wheeler, D.J. and Needham, R.M. (1997). *Tea Extensions*. October 1994, pp. 1–3.
- Whitaker *et al.* 2002 Whitaker, A., Shaw, M. and Gribble, D.G. (2002). Denali: Lightweight virtual machines for distributed and networked applications. *Technical Report 02-02-01*, University of Washington.
- Wiesmann *et al.* 2000 Wiesmann, M., Pedone, F., Schiper, A., Kemme, B. and Alonso, G. (2000). Understanding replication in databases and distributed systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS '2000)*, Taipei, Republic of China, p. 464.

- Williams 1998
- Winer 1999
- Wobber *et al.* 1994
- Wright *et al.* 2002
- wsdl4j.sourceforge.org
- Wulf *et al.* 1974
- Wuu and Bernstein 1984
- www.accessgrid.org
- www.adventiq.com
- www.akamai.com
- www.apple.com I
- www.apple.com II
- www.beowulf.org
- www.bittorrent.com
- www.bluetooth.com
- www.butterfly.net
- www.bxa.doc.gov
- www.cdk5.net
- www.citrix.com
- www.conviva.org
- www.coralcdn.org
- Williams, P. (1998). JetSend: An appliance communication protocol. In *Proceedings of the IEEE International Workshop on Networked Appliances, (IEEE IWNA '98)*, Kyoto, Japan, November, pp. 51–53.
- Winer, D. (1999). *The XML-RPC specification*.
- Wobber, E., Abadi, M., Burrows, M. and Lampson, B. (1994). Authentication in the Taos operating system. *ACM Transactions on Computer Systems*. Vol. 12, No. 1, pp. 3–32.
- Wright, M., Adler, M., Levine, B.N. and Shields, C. (2002). An analysis of the degradation of anonymous protocols. In *Proceedings of the Network and Distributed Security Symposium (NDSS '02)*, February.
- The Web Services Description Language for Java Toolkit (WSDL4J)*.
- Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C. and Pollack, F. (1974), HYDRA: The kernel of a multiprocessor operating system. *Comms. ACM*, Vol. 17, No. 6, pp. 337–345.
- Wuu, G.T. and Bernstein, A.J. (1984). Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the Third Annual Symposium on Principles of Distributed Computing*, pp. 233–42.
- O projeto Access Grid*.
- Adventiq Ltd. *Home page apresentando sua tecnologia KVM-over-IP*.
- Akamai. Home page*.
- Apple Computer. *Especificações do protocolo Bonjour*.
- Apple Computer. *iChat AV videoconferência para o público em geral*.
- O projeto Beowulf. Centro de recursos*.
- Site oficial do BitTorrent*.
- Site oficial do Bluetooth SIG*.
- Plataforma de jogos online escalável, confiável e de alto desempenho, GoGrid*.
- Bureau of Export Administration, US Department of Commerce, *Commercial Encryption Export Controls*.
- Coulouris, G., Dollimore, J. e Kindberg, T. (eds.), *Distributed Systems, Concepts and Design: material de apoio*.
- Citrix Corporation. *Citrix XenApgs*.
- Conviva. *Home page*.
- Coral Content Distribution Network. *Home page*.

- www.cren.net Corporation for Research and Educational Networking, *CRÉN Certificate Authority*.
- www.cryptopp.com Crypto++® Library 5.2.1.
- www.cs.cornell.edu O 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS'09). *Discurso programático por Jeff Dean sobre sistemas distribuídos de larga escala na Google: sistemas atuais e futuros rumos*.
- www.cs.york.ac.uk/dame *Distributed Aircraft Maintenance Environment (DAME)*.
- www.cuseeme.com CU-SeeMe Networks Inc. *Home page*.
- www.dancres.org Projeto de código-fonte aberto Blitz. *Home page*.
- www.doi.org International DOI Foundation. *Páginas sobre identificadores de objeto digitais*.
- www.dropbox.com Serviço de hospedagem de arquivos Dropbox. *Home page*.
- www.dtnrg.org Delay Tolerant Networking Research Group. *Home page*.
- www.gigaspaces.com GigaSpaces. *Home page*.
- www.globalcrossing.net Global Crossing. *IP Network Performance – histórico mensal*.
- www.globexplorer.com *Globexplorer, a maior biblioteca online do mundo de imagens aéreas e de satélite*.
- www.globus.org O projeto Globus. *Versão estável mais recente do Globus Toolkit*.
- www.google.com I Google. *Google Apps*.
- www.google.com II Google. *Google Maps*.
- www.google.com III Google. *Informações corporativas da Google*.
- www.google.com IV Google. *Informações corporativas da Google. Princípios de projeto*.
- www.gridmpi.org O projeto GridMPI. *Home page*.
- www.handle.net Sistema Handle. *Home page*.
- www.iana.org I Internet Assigned Numbers Authority. *Home Page da IANA*.
- www.iana.org II Internet Assigned Numbers Authority. *Registro no espaço de endereçamento multicast IPv4*.
- www.ibm.com IBM. *Home page do servidor de aplicativos WebSphere*.
- www.ietf.org Internet Engineering Task Force. *Página de índice do Internet RFC*.
- www.iona.com Iona Technologies, *Orbix*.
- www.ipnsig.org Projeto de Internet InterPlaNetary. *Home page*.
- www.ipoque.com Ipoque GmbH. *Internet Study 2008/2009*.
- www.isoc.org Robert Hobbes Zakon. *Timeline na Internet de Hobbes*.
- www.jbidwatcher.com Projeto JBidwatcher. *Home page*.

- www.jboss.org Comunidade de código-fonte aberto JBoss. *Home page*.
- www.jgroups.org Projeto JGroups. *Home page*.
- www.json.org Representação de dados externos JSON. *Home page*.
- www.kontiki.com Kontiki Delivery Management System. *Home page*.
- www.microsoft.com I Microsoft Corporation. *Serviços de Active Directory*.
- www.microsoft.com II Microsoft Corporation. *Autenticação Kerberos do Windows 2000*, White Paper.
- www.microsoft.com III Microsoft Corporation. *Home page do NetMeeting*.
- www.microsoft.com IV Microsoft Corporation. *Home page do Azure*.
- www.mozilla.org Netscape Corporation. *Especificação SSL 3.0*.
- www.mpi-forum.org Message Passing Interface (MPI) Forum. *Home page*.
- www.neesgrid.org NEES Grid, *Building the National Virtual Collaboratory for Earthquake Engineering*.
- www.netscape.com *Netscape*. Home page.
- www.nfc-forum.org Near Field Communication (NFC). *Home page do fórum*.
- www.oasis.org Troca de mensagens confiável de Web Services. WS-Reliablemessaging. Vn 1.1. *Padrão Oasis*.
- www.omg.org Object Management Group. *Índice para serviços CORBA*.
- www.opengroup.org Open Group. *Portal para o mundo do DCE*.
- www.openmobilealliance.org Open Mobile Alliance. *Home page*.
- www.openssl.org Projeto OpenSSL. *OpenSSL: o toolkit Open Source para SSL/TLS*.
- www.openstreetmap.org OpenStreetMap. *Home page*.
- www.osgi.org A OSGi Alliance. *Home page*.
- www.ow2.org O consórcio OW2. *Home page*.
- www.parc.com PARC Forum. *Apresentação para Marissa Mayer, vice-presidente da Google*.
- www.pgp.com PGP. *Home page*.
- www.progress.com Apama da Progress Software. *Home page*.
- www.prototypejs.org Prototype JavaScript Framework. *Home page*.
- www.realvnc.com RealVNC Ltd. *Home page*.
- www.redbooks.ibm.com IBM Redbooks. *Fundamentos do WebSphere MQ V6*.
- www.reed.com Read, D.P. (2000). *O argumento fim a fim*.
- www.research.ibm.com Projeto Gryphon. *Home page*.
- www.rsasecurity.com I RSA Security Inc. *Home page*.
- www.rsasecurity.com II RSA Corporation (1997). *DES Challenge*.
- www.rsasecurity.com III RSA Corporation (2004). *RSA Factoring Challenge*.

- www.rtj.org *Real-Time for Java TM Experts Group.*
- www.secinf.net *Network Security Library.*
- www.sei.cmu.edu Software Engineering Institute, Carnegie Mellon. *Home page da iniciativa Ultra Large Systems (ULS).*
- www.smart-its.org O projeto Smart-Its. *Home page.*
- www.spec.org *Benchmark SPEC SFS97.*
- www.springsource.org Comunidade SpringSource. *Spring Framework.*
- www.upnp.org Universal Plug and Play. *Home page.*
- www.us.cdnetworks.com CDNetworks Inc. *Home page.*
- www.uscms.org USCMS, o *Compact Muon Solenoid.*
- www.verisign.com Verisign Inc. *Home page.*
- www.w3.org I World Wide Web Consortium. *Home page.*
- www.w3.org II World Wide Web Consortium. *Páginas sobre a HyperText Markup Language.*
- www.w3.org III World Wide Web Consortium. *Páginas sobre atribuição de nomes e endereçamento.*
- www.w3.org IV World Wide Web Consortium. *Páginas sobre o HyperText Transfer Protocol.*
- www.w3.org V World Wide Web Consortium. *Páginas sobre a Resource Description Framework e outros esquemas de metadados.*
- www.w3.org VI World Wide Web Consortium. *Páginas sobre a Extensible Markup Language.*
- www.w3.org VII World Wide Web Consortium. *Páginas sobre a Extensible Stylesheet Language.*
- www.w3.org VIII *XML Schemas.* W3C Recommendation. (2001)
- www.w3.org IX World Wide Web Consortium. *Páginas sobre SOAP.*
- www.w3.org X World Wide Web Consortium. *Páginas sobre Canonical XML, Version 1.0.* W3C Recommendation.
- www.w3.org XI World Wide Web Consortium. *Páginas sobre Web Services Description Language (WSDL).*
- www.w3.org XII World Wide Web Consortium. *Páginas sobre XML Signature Syntax and Processing.*
- www.w3.org XIII World Wide Web Consortium. *Páginas sobre a especificação de gerenciamento de chaves XML, XKMS.*
- www.w3.org XIV World Wide Web Consortium. *Páginas sobre XML Encryption Syntax and Processing.*
- www.w3.org XV World Wide Web Consortium. *Páginas sobre Web Services Choreography Requirements.* W3C Working Draft.
- www.w3.org XVI Burdett, D., Kavantsas, N. *Introdução ao modelo WS Choreography.* W3C Working Draft.

- [www.w3.org XVII](http://www.w3.org/XVII) World Wide Web Consortium. *Páginas sobre Web Services Choreography Description Language Version 1.0*.
- [www.w3.org XVIII](http://www.w3.org/XVIII) World Wide Web Consortium. *Páginas sobre Web Services Choreography Interface (WSCI)*.
- [www.w3.org XIX](http://www.w3.org/XIX) World Wide Web Consortium. *Páginas sobre Device Independence*.
- [www.w3.org XX](http://www.w3.org/XX) World Wide Web Consortium. *Páginas a the Semantic Web*.
- [www.w3.org XXI](http://www.w3.org/XXI) World Wide Web Consortium. *Páginas sobre o XML Binary Charaterization Working Group*.
- [www.w3.org XXII](http://www.w3.org/XXII) World Wide Web Consortium. *Páginas sobre as recomendações do SOAP Message Transmission Optimization Protocol*.
- [www.w3.org XXIII](http://www.w3.org/XXIII) World Wide Web Consortium. *Páginas sobre o WS-Addressing Working Group*.
- [www.w3.org XXIV](http://www.w3.org/XXIV) World Wide Web Consortium. *Páginas sobre a Geolocation API Specification*.
- www.wapforum.org WAP Forum. *White Papers e especificações*.
- www.wlana.com *O IEEE 802.11 Wireless LAN Standard*.
- www.xbow.com Crossbow Technology Inc. *Páginas sobre redes sensoriais sem fio*.
- www.xen.org Comunidade de código-fonte aberto Xen. *Home page*.
- www.zeroconf.org IETF Zeroconf Working Group. *Home page*.
- Wyckoff *et al.* 1998 Wyckoff, P., McLaughry, S., Lehman, T. and Ford, D. (1998). T Spaces. *IBM Systems Journal*, Vol. 37, No. 3.
- Xu and Liskov 1989 Xu, A. and Liskov, B. (1989). The design for a fault-tolerant, distributed implementation of Linda. In *Proceedings of the 19th International Symposium on Fault-Tolerant Computing*, Chicago, IL, June, pp. 199–206.
- zakon.org Zakon, R.H. Hobbes' Internet Timeline v7.0.
- Zhang and Kindberg 2002 Zhang, K. and Kindberg, T. (2002). An authorization infrastructure for nomadic computing. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, Monterey, CA, June, pp. 107–113.
- Zhang *et al.* 1993 Zhang, L., Deering, S.E., Estrin,D., Shenker, S. and Zappala, D. (1993). RSVP – A new resource reservation protocol. *IEEE Network Magazine*, Vol. 9, No. 5, pp. 8–18.
- Zhang *et al.* 2005a Zhang, H., Goel, A., and Govindan, R. (2005), Improving lookup latency in distributed hash table systems using random sampling. *IEEE/ACM Trans. Netw.* 13, 5 (Oct. 2005), 1121–1134.

- Zhang *et al.* 2005b Zhang, X., Liu, J., Li, B. and Yum, T.-S. (2005). CoolStreaming/DONet: A data-driven overlay network for live media streaming. In *Proceedings of IEEE INFOCOM'05*, Miami, FL, USA, March, pp. 2102–2011.
- Zhao *et al.* 2004 Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D. and Kubiatowicz, J.D. (2004). Tapestry: A resilient global-scale overlay for service deployment, *IEEE Journal on Selected Areas in Communications*, Vol. 22, No. 1, pp. 41–53.
- Zimmermann 1995 Zimmermann, P.R. (1995). *The Official PGP User's Guide*. MIT Press.

Índice

A

- abertura 17, 18, 41, 528–529, 926
- acontece antes 236, 606–607
- acordo
 - de entrega *multicast* 648
 - no consenso e problemas relacionados 660–662
 - problemas de 659
 - uniforme 650
- acordo, de entrega *multicast* 236
- adaptabilidade 25
- adaptação
 - às variações de recurso 869–870
 - de conteúdo 866–869
 - reconhecimento de energia 870
- adaptação com reconhecimento de energia 870
- adaptador de objeto 348
- AES (Advance Encryption Standard) 490, 501
- agente de requisição de objeto (ORB) 340
- agente móvel 49–51
- AJAX 53–57
- aleatoriedade 670
- algoritmo de cifra de fluxo RC4 490
- algoritmo de criptografia de chave pública RSA 485, 491–493
- algoritmo de criptografia triple-DES 501
- algoritmo de hashing seguro SHA 499, 501
- algoritmo de hashing seguro SHA-1 434–437, 450–451, 457–458
- algoritmo de mensagem de resumo MD5 499, 501
- algoritmo de Nagle 124–125
- algoritmo de roteamento de vetor de distância 99
- algoritmo do balde furado 893
- algoritmos de roteamento de estado de enlace 101
- alias 571
- Amazon Elastic Compute Cloud (EC2) 418–419
- Amazon Elastic MapReduce 419–420
- Amazon Flexible Payments Service (FPS) 419–420
- Amazon Simple DB 419–420
- Amazon Simple Queue Service (SQS) 419–420
- Amazon Simple Storage Service (S3) 419–420
- Amazon Web Services (AWS) 418–419, 965
 - Amazon Elastic Compute Cloud (EC2) 418–419
 - Amazon Elastic MapReduce 419–420
 - Amazon Flexible Payments Service 419–420
 - Amazon Simple DB 419–420
 - Amazon Simple Queue Service (SQS) 419–420
 - Amazon Simple Storage Service (S3) 419–420
 - Dynamo 720, 801
 - REST no 418–419
- ambiente de execução 286
- Amoeba
 - protocolo de *multicast* 654
 - servidor de execução 289
- análise de tráfego 865
- Andrew File System (AFS) 530, 548–557
 - desempenho 556–557
 - no DCE/DFS 559–560
 - para Linux 530
 - suporte remoto 556–557
- anéis de E/S, no Xen 327–328
- anúncio
 - em publicar-assinar 245
 - no roteamento baseado em conteúdo 252
- API Java
 - DatagramPacket 151–152
 - DatagramSocket 152
 - InetAddress 149
 - MulticastSocket 172–173
 - ServerSocket 155–156
 - Socket 156
 - aplicação de compartilhamento de arquivos 425
- Apollo Domain 262–263
- applet 31–32, 49–51
 - threads dentro de 298
- argumento fim-a-fim 61–62, 174

- armazenamento de objeto persistente
comparação com serviço de arquivo 523
Java persistente 214–524
- ARP *veja* protocolo de resolução de endereços
- arquitetura aberta de serviços de grade (OGSA) 417–418
- arquitetura de duas camadas físicas 53–54
- arquitetura Gossip 783–792
mensagem de fofoca 785
processando 789–791
propagando 791–792
processamento de atualização 788–789
processamento de consulta 788
- arquitetura de *n* camadas físicas 53–54
- arquitetura de processamento simétrico 283
- arquitetura de *software* 360–361
- arquitetura de três camadas físicas 53–54, 362–363
- arquitetura orientada a serviço (SOA) 413–414
mashup 414
- arquivo
mapeado 288
replicado 795
- arquivo de recuperação 751–761
reorganização 755, 759
- arrendamentos 223
no JavaSpaces 271–272
no Jini 216
nos serviços de descoberta 831
para *callbacks* 223
- assinatura, em publicar-assinar 243
- assinatura de método 205
- assinatura digital 476–477, 493–500
na XML 409–410
- Assistente pessoal digital (PDA) 818
- associação 821, 825, 827–835
direta 834
espontânea 826
espontânea segura 860–863
física 834
indireta 842
problema 827
- associação de dispositivo espontânea segura 860–863
- associação espontânea 826
- associativo 261–262
- ataque da data de nascimento 498
- ataque de falsificação de mensagem 467
- ataque de força bruta 484
- ataque de intromissão 467
- ataque de mascaramento 467
- ataque de negação de serviço 19, 75–76
- ataque de reprodução 467
- ataque de texto puro escolhido 492
- ataque do homem no meio 467
- ativação 213
objeto distribuído 339
ativação do escalonador 302
ativador 213
- ATM (Asynchronous Transfer Mode) 88, 90, 92, 95, 102, 130
- atomicidade da falha 680, 751–752
- atualização perdida 683, 695
- autenticação 74–75, 474–476
baseada na localização 863
- autenticação Kerberos
para NFS 544–545
- autoridade certificadora 500
- Azure da Microsoft 965

B

- backbone* 113
- baixo acoplamento 385
- balanceamento de carga
Bigtable 954
- baliza 874
infravermelha 834
rádio 853
- base de computação confiável 472, 826
- Bayou 425, 792–794
- procedimento de integração 793
- verificação de dependência 793
- bean*
Enterprise JavaBeans (EJB) 366–367
- bean* de sessão
Enterprise JavaBeans (EJB) 366–367
- bean* orientado a mensagem
Enterprise JavaBeans (EJB) 366–367
- Bigtable 927, 948–954
arquitetura 950–954
balanceamento de carga 954
interface 948–950
separação entre controle e dados 951
tablets 950
- BitTorrent 425, 435–436, 447–448, 906–908
desafogamento 907
desafogamento otimista 908
- leecher* (parasita) 907
- rastreador 906
- seeder* (semeador) 906
- torrente 907
- trecho 906

- broadcast* 233
buffers de protocolo 927, 929–932
 serialização 929
- C**
- cache 49–50, 766
 coerência de arquivos colocados na cache 799
cache Web Squirrel 449–452
callback
 na invocação de método remoto CORBA 357–358
 na invocação de método remoto Java 223
camada de aplicação 93, 95
camada de apresentação 95
camada de enlace de dados 95
camada de sessão 95
camada de transporte 95
camada física 93, 95
camadas 175
camadas em protocolos 93
camadas físicas 52–53
 arquitetura de duas camadas físicas 53–54
 arquitetura de n camadas físicas 53–54
 arquitetura de três camadas físicas 53–54, 362–363
camadas lógicas 51–52
caminho mais curto primeiro (OSPF) 113
canais de comunicação
 ameaças aos 74–75
 desempenho 63–64
canal confiável 631
canal fisicamente restrito 834, 861–863
canal seguro 74–75
cancelamento em cascata 689
cancelar 682
capacidade 480
capacidade de linearização 776
capacidade de ser atingido 614
capacidade de serialização de uma cópia 802
captura de chamada do sistema 285
capturar exceção 206
carimbo de tempo
 Lamport 608
 vetorial 609
carimbo de tempo de Lamport 608
carimbo de tempo vetorial 609
 comparação 610
 operação de mistura 609
CDR *veja* CORBA Common Data Representation
centro de dados 13, 923
certificado 477–479
 formato padrão X.509 499
 certificado de chave pública 476, 478
 certificado X.509 499
CGI *veja* Common Gateway Interface
chamada de procedimento remoto 43–44, 186, 195–204
 buffers de protocolo 931
 desempenho da *veja* mecanismo de invocação, desempenho do
 enfileirada 313
 filtragem de duplicatas 198
 implementação 200–201
 leve 309–311
 nula 305
 parâmetros de entrada 196
 parâmetros de saída 196
 procedimento *stub* 200
 procedimento *stub* de servidor 200
 retentativa de mensagem de requisição 198
 retransmissão de respostas 198
 semântica 198
 transparência 198–199
chave de sessão 475
chave pública 473
chave secreta 473
Chorus 317
Chubby 927, 940–947
 arquitetura 943–944
 consistência de cache 943
 interface 941–943
 Paxos 943–946
 replicação 943
 travas 940
ciclo de vida 371
cifra de bloco 485–486
cifra de fluxo 486–487
circuito virtual 97
classes serventes 221, 348–349, 354–355
cliente magro 56–57
clientes 15
cliente-servidor 6
codec 887
código de autenticação de mensagem (MAC) 496
código móvel 17, 19, 49–51, 75–76
 ameaças à segurança 467
coleta de lixo 611
 em sistema de objeto distribuído 209
 local 206
coleta de lixo distribuída 209, 215–216
 em Java 209
comércio eletrônico
 necessidades de segurança 469

- Common Gateway Interface 31–32
compactação de dados 887
compactação de vídeo MPEG 886, 887, 892
compatilhamento de carga 289–290
componente 41–42, 59–61, 336, 358–365
 arquitetura de *software* 360–361
 composição 361–362
 contêiner 362–364
 contrato 360–361
 definição 360–361
 distribuição 364–365
 engenharia de *software* baseada em componentes (CBSE) 916
 estratégia leve 364–365
 estratégia no lado do servidor 365–366
 estratégia pesada 365–366
 interface exigida 360–361
 interface fornecida 360–361
 limitações dos objetos 336, 358–361
 middleware 336, 358–365
 OpenCOM 374
 OSGi 374
 servidor de aplicação 363–364
computação acoplada ao corpo 820
computação com reconhecimento de contexto 10, 820
computação com reconhecimento de localização 10
computação com serviço público 13–14
computação em Grade 14
computação em nuvem 13–14, 319, 320, 417–420, 921–922
 Amazon Web Services (AWS) 418–419, 965
 Dynamo 720, 801
 Eucalyptus 965
 Google App Engine 922, 965
 Hadoop 965
 Microsoft's Azure 965
 OpenStreetMaps 965
 Sector/Sphere 965
computação manual 818
computação móvel 10–11, 818–879
 origem da 818
computação nômade 871
computação ubíqua 10–11, 818–879
 origem da 819
computador de rede 57–58
comunicação
 assíncrona 148
 confiável 71–72, 148
 entidades em comunicação 41–43
 funções 45–48
 grupo 169–174
 paradigmas de comunicação 42–45
 produtor-consumidor 146
 síncrona 148
 suporte do sistema operacional para 303–311
comunicação assíncrona 148, 232
 em publicar-assinar 244
comunicação confiável 71–72
 integridade 71–72
 no SOAP 392–393
 validade 71–72
comunicação em grupo 43–44, 169–174, 232–239, 646–659, 771–775
 confiabilidade 646
 implementação 236–239
 JGroups 238–242
 modelo de programação 233–235
 modo de visualização síncrono 773–775
 ordenação 646
 para aplicações colaborativas 233
 para disseminação confiável de informações 233
 para gerenciar dados replicados 771
 para monitoramento e gerenciamento de sistema 233
 para tolerância a falhas 233
comunicação em grupo com modo de visualização síncrono 238–239
comunicação em grupo virtualmente síncrona 775
comunicação entre processos 41–43
 características 147
comunicação generativa, em espaços de tuplas 45, 265
comunicação indireta 43–44, 230
 comunicação em grupo 43–44, 232–239
 espaço de tuplas 45, 265–271
 filas de mensagem 43–44, 232–258
 memória compartilhada distribuída 45, 261–265
 publicar-assinar 45, 242–253
 uma comparação de estratégias 274–276
comunicação produtor-consumidor 146
comunicação síncrona 148
concorrência 2, 23
 de atualizações de arquivo 527
conexão persistente 192, 309
conexão persistente 309
confiança 826, 857, 863
confirmação 682
confirmação negativa 649

- confirmação provisória 735–736
 conjunto de cifras 512
 consenso 659–670, 940, 944
 em um sistema síncrono 663
 Paxos 944
 relacionado a outros problemas 662
 resultado de impossibilidade para um sistema
 assíncrono 667–668
 consenso de quorum 803, 809–811
 consistência
 de dados replicados 768
 Google File System (GFS) 939–940
 consistência interativa 662
 consistência sequencial 777
 consulta
 espaço-temporal 857
 processamento distribuído 850
 consulta espaço-temporal 857
 consumo de energia 823, 825
 da compactação 869
 da comunicação 849
 dos protocolos de descoberta 831
 e adaptação 870
 e negação de serviço 858
 contêiner 362–364
 contêiner de *servlets* 395–396
 Context Toolkit 847
 contexto 844
 contexto de atribuição de nomes 573
 contrato, em components 360–361
 controlador 823
 controle de acesso 479–481
 controle de admissão 890, 896–897
 controle de concorrência 683–726
 atualização perdida 683
 com travas *veja* travas
 comparação de métodos 718
 Dynamo 720
 em transação distribuída 740–743
 Google Apps 719
 na Wikipedia 719
 no Dropbox 719
 no Dynamo 720
 no serviço de controle de concorrência CORBA 696
 operações conflitantes 686
 otimista *veja* controle de concorrência otimista
 pela ordem de indicação de hora *veja* ordem de
 indicação de hora
 recuperação inconsistente 684
 regras de conflito de operação 686, 694
 controle de concorrência otimista 707–711
 comparação de validação para frente e para trás 711
 exemplos 719
 fase de atualização 708
 fase de trabalho 707
 inanição 711
 na transação distribuída 742
 validação 708
 validação para frente 710
 validação para trás 709
 controle de congestionamento *veja* controle de
 congestionamento de rede
 controle de fluxo 123–124
 cookie 538
 CoolStreaming 910
 Cooltown 871–878
 baliza 874
 cópia na escrita 290
 cópia primária 547–548
 CORBA
 agente de requisição de objeto (ORB) 340,
 348
 arquitetura 348–351
 adaptador de objeto 348
 agente de requisição de objeto (ORB) 348
 esqueleto 349–350
 proxy 349–350
 Common Data Representation 160–161
 comparado com serviços Web 398
 eficiência comparada com serviços Web 399
 empacotamento 161
 estudo de caso 340–358
 exemplo de cliente e servidor 353–358
 interface de esqueleto dinâmico 350–351
 interface de invocação dinâmica 350–351
 invocação de método remoto 341–358
 callback 357–358
 linguagem de definição de interface 197, 341–
 344
 atributo 345
 herança 345–346
 interface 342
 método 342
 módulo 342
 pseudo-objeto 350–351
 tipo 343–344
 mapeamento de linguagem 345–346
 modelo de objeto 341
 objeto 341
 protocolo Internet Inter-ORB (IIOP) 352

- referência de objeto interoperável (IOR) 350–351
comparada com URL 398
persistente 352
transiente 352
referência de objeto remoto *veja* referência de objeto interoperável CORBA
repositório de implementações 349–350
repositório de interfaces 349–350
RMI assíncrona 347
serviços 352
 Event Service 253
 serviço de controle de concorrência 696
 serviço de estado persistente 523
corretagem 57–58
corte 613
 consistente 613
 fronteira 613
crachá ativo 820, 844, 846, 855
 eventos 838
credenciais 482–483
criptografia 74–75, 473–477, 484–493
 desempenho de algoritmos 501
 e política 502
 na segurança da XML 410–411
criptografia assimétrica 484, 491–493
criptografia de chave pública 491–493
criptografia de curva elíptica 493
criptografia simétrica 484
CSMA/CA 136–137
CSMA/CD 131
cyber foraging 869
- D**
- dados críticos em relação ao tempo 25
datagrama 97, 110–111
delegação (de direitos) 482
Denali 320
depurando programas distribuídos 612, 619–625
DES (Data Encryption Standard) 489, 501
desacoplamento em relação ao espaço 43–44, 230
 em espaços de tuplas 267
desacoplamento temporal 43–44, 230
 em espaços de tuplas 267
desafio, para autenticação 475
desafogamento, no BitTorrent 907
desafogamento otimista, no BitTorrent 908
desativação
 objeto distribuído 339
descoberta de dispositivo 828
descoberta de serviço 11
 veja serviço de descoberta
desempacotamento 158
despachante 210–211
 em serviços Web 398
genérico na RMI Java 224
no CORBA 348–349
detecção de colisão 133
detecção de término 611
detector de falha 632–633
 para resolver consenso 668–669
DHCP – Dynamic Host Configuration Protocol 117, 121–122, 827
Diffie–Hellman, protocolo 862
difusão (na criptografia) 487
direitos de acesso 72–73
disponibilidade 22–23, 766, 782–801
distribuição, de componentes 364–365
DNS *veja* Domain Name System
Domain Name System (DNS) 124–125, 576–583
 consulta 577–578
 implementação BSD 582
 navegação 580–581
 nome de domínio 577–578
 registro de recursos 581
 servidor de nomes 578–583
 zona 578–579
domínio, no Xen 322
 domínio0 322
 domínioU 322
domínio de atribuição de nomes 572
domínio de proteção 479
download de código 31–32
 na invocação de método remoto Java 219
driver de dispositivo dividido, no Xen 326–327
Dropbox
 controle de concorrência 719
DSL (linha de assinante digital) 88, 103
Dynamo
 consenso de quorum 801
 controle de concorrência 720
 replicação 801
- E**
- e-Commerce* 4
eleição 641–646
 algoritmo valentão 644
 para processos em um anel 642
empacotamento 158
empresa para empresa (B2B) 414
 integração 414
encadeamento de blocos de cifra (CBC) 485
encadeamento de certificados 478, 500

- encapsulamento 93, 107
- End System Multicast (ESM) 908–912
 - adaptação com reconhecimento de desempenho 910, 912
 - auto-organização 910
 - construção em árvore 910–912
 - gerenciamento de participação de membros 911
 - ingressando em uma árvore 911
 - lidando com nós que saem 912
 - seleção do pai 911
 - endereçamento IP 108–111
 - endereço de Internet *veja endereço IP*
 - endereço de transporte 96
 - endereço IP, API Java 149
 - engenharia de *software* baseada em componentes (CBSE) 916
 - Ensemble 238–239
 - Enterprise Application Integration (EAI) 254
 - Enterprise JavaBeans (EJB)
 - bean* 366–367
 - bean* de sessão 366–367
 - bean* orientado a mensagem 366–367
 - configuração por exceção 367–368
 - estudo de caso 364–372
 - funções 365–366
 - gerenciado por *beans* 365–366
 - gerenciado por contêiner 365–366
 - gerenciamento de ciclo de vida 367–368
 - interface comercial 366–367
 - entidades em comunicação 41–43
 - componente 41–42
 - nó 41–42
 - objeto 41–42
 - processo 41–42
 - serviços Web 42–43
 - equivalência serial 681, 685
 - erro de acesso inválido a página 291
 - escalabilidade 19–22, 527, 924
 - escalonamento
 - no Xen 322–325
 - escalonamento de recursos na base do melhor esforço 888
 - escalonamento em tempo real 898
 - escalonamento não preemptivo 300–301
 - escalonamento preemptivo 399
 - eScience 4
 - escrita prematura 689
 - espaço de endereçamento 285, 287–288
 - alias 296
 - herança 290
 - região 287
 - região compartilhada 288, 308
 - espaço de nomes 570
 - espaço de tuplas 45, 265–271
 - atribuição de nomes livre 267
 - comparado com sistema de eventos 840
 - compatilhamento distribuído 267
 - comunicação generativa 45, 265
 - desacoplamento espacial 267
 - desacoplamento temporal 267
 - em sistema volátil 838–840
 - implementação 268–271
 - implementação *peer-to-peer* 271
 - L²imbo 840
 - Linda 265
 - modelo de programação 265–267
 - replicação 268–270
 - TOTA 840
 - espaço inteligente 822
 - especificação de fluxo 894
 - esqueleto 210–211
 - desnecessário com despachante genérico 224
 - dinâmico 212
 - dinâmico no CORBA 350–351
 - em serviços Web 398
 - no CORBA 349–350
 - esqueleto dinâmico 212, 350–351
 - no CORBA 350–351
 - eSquirt 875–877
 - estabelecimento de ponto de verificação 755
 - estaçao de base, sem fio 134–135, 819
 - estado global 610–625
 - consistente 614
 - estável 614
 - instantâneo 615–619
 - predicado 614
 - estranho conhecido 854
 - estudos de caso
 - Andrew File System (AFS) 548–557
 - arquitetura Gossip 783–792
 - Bayou 792–794
 - BitTorrent 906–908
 - cache Web Squirrel 449–452
 - Cooltown 871–878
 - CORBA 340–358
 - Domain Name System (DNS) 576–583
 - End System *Multicast* (ESM) 908–912
 - Enterprise JavaBeans (EJB) 364–372
 - Fractal 372–378
 - Global Name Service 585–588
 - Gnutella 447–449

- infraestrutura do Google 917–965
 invocação de método remoto Java 217–225
 Java Messaging Service (JMS) 258–262
 JavaSpaces 271–274
 JGroups 238–242
 Kerberos 505–510
 Message Passing Interface (MPI) 178–180
 Network File System (NFS) 536–548
 Network Time Protocol 603–606
 OceanStore 451–456
 PAN sem fio Bluetooth IEEE 802.15.1 138
 protocolo Needham–Schroeder 504–505
 protocolos de Internet 106
 rede Ethernet 130
 rede local sem fio IEEE 802.11 (WiFi) 134–135
 roteamento de IP 113
 RPC Sun 201–204
 segurança do WiFi 515–517
 serviço de diretório X.500 588–592
 servidor de arquivos de vídeo Tiger 901–906
 sistema de arquivos Coda 795–801
 sistema de arquivos Ivy 455–459
 sobreposição de roteamento Pastry 436–445
 sobreposição de roteamento Tapestry 444–446
 Transport Layer Security (TLS) 511–515
 Websphere MQ 256–258
 World Wide Web 26–34
 Xen 320–331
 Ethernet para aplicações em tempo real 134–135
 Eucalyptus 965
 evento 6
 composto 838
 concorrência 608
 crachá ativo 838
 em sistema volátil 837–838
 heap 839
 no Xen 322–323
 notificação 243
 ordem 66–67
 sistema, comparado com espaço de tuplas 840
 exatamente uma vez 198
 exceção 205
 capturar 206
 lançar 206
 na invocação remota CORBA 200
 exclusão mútua 633–641
 algoritmo de Maekawa 639
 entre processos em um anel 636
 pelo servidor central 635
 token 636
 usando *multicast* 637–639
 execuções restritas 690
 Exokernel 318
 explosão de confirmações 647
- F**
- falha
 arbitrária 68–70
 bizantina 68–70
 mascaramento 70–71
 temporização 70–71
 falha bizantina 68–70
 falha de detecção 21–22
 falha de temporização 70–71
 falha independente 2
 falha por colapso 68–70, 632
 falha por omissão 67–68
 comunicação 68–70
 processo 68–70
 falhas de entrega 189
 falhas por omissão de envio 68–70, 148
 falhas por omissão de recepção 68–70
 fidelidade 869
 fila de espera 649
 filas de mensagem 45, 59–61, 254–258
 entrega confiável 255
 implementação 256–258
 integridade 255
 intermediário de mensagem 256
 modelo de programação 254–256
 segurança 256
 transação 255
 transformação de mensagem 255
 validade 255
 filtragem, no roteamento baseado em conteúdo 251
 filtro, em publicar-assinar 245
 firewall 9, 125–129, 392–393, 483
 fluxo de dados *veja rede, fluxo de dados*
 fluxos de dados baseados no tempo 886
 fluxos de dados isocrônicos 886
 fofoca (*gossip*) 447–448
 em End System Multicast (ESM) 911
 em publicar-assinar 253
 fofoca informada 253
- Fractal
 Architectural Description Language (ADL)
 375–376
 composição hierárquica 375–376
 controlador 375–376
 estudo de caso 372–378
 gerenciamento de ciclo de vida 376
 interceptação 377

- interface de cliente 373
 interface de servidor 373
 membrana 375–376
 programação com interfaces 372
 reflexão 376
 vínculo 373
 vínculo composto 373
 vínculo primitivo 373
- frame relay* *veja rede, frame relay*
- Freenet 425, 429–430
- front-end* 770
- FTP 95, 96, 106, 127–128
- função de alçação 484
- função de *hashing* segura 426, 495
- função de *hashing* unilateral 498
- função de resumo 495
- função de resumo segura 476
- função unilateral 484
- funções 45–48
- G**
- Galileo, sistema de navegação por satélite 852
- games *online* para vários jogadores 5–6
- gateway* 89
- generais bizantinos 662, 664–668
- gerador de fluxo de chaves 486
- gerenciador de recuperação 751–752
- gerenciador de réplicas 769
- gerenciador de travas 697
- gerenciamento de ciclo de vida 339
- Enterprise JavaBeans (EJB) 367–368
 - Fractal 376
 - gerenciamento de dispositivos
 - no Xen 326–329
 - gerenciamento de memória virtual
 - no Xen 325–326
 - gerenciamento de recursos (para multimídia) 897–899
 - glifo 834
 - Global Name Service 585–588
 - identificador de diretório 585
 - raiz de trabalho 586
 - Globus Toolkit 417–418
 - GLONASS 852
 - GNS *veja Global Name Service* 585
 - Gnutella 425, 447–449
 - Google 5
 - aplicações 921
 - Caffeine 918
 - centro de dados 923
 - computação em nuvem 921–922
 - esquadriamento 918
 - Google App Engine 921
 - Google Apps 921
 - índice invertido 918
 - mecanismo de busca 918–921
 - modelo físico 922–924
 - PageRank 919
 - pesquisa profunda 918
 - plataforma como serviço 922
 - software* como serviço 921
 - Google App Engine 921, 922, 965
 - Google Apps 921
 - controle de concorrência 719
 - Google Earth 4, 948, 950, 961
 - Google File System (GFS) 927, 935–940
 - arquitetura 937–939
 - consistência 939–940
 - interface 936–937
 - replicação 937
 - requisitos 935–936
 - separação entre controle e dados 938
 - trechos 937
 - uso de cache 938
 - Google Maps 4, 948, 961
 - GPS *veja Sistema de Posicionamento Global (Global Positioning System – GPS)*
 - Grade 414–418
 - aplicações de uso intensivo de dados 416
 - aplicações de uso intensivo de poder computacional 417–418
 - arquitetura aberta de serviços de grade (OGSA) 417–418
 - eScience 4
 - Globus Toolkit 417–418
 - middleware* 417–418
 - o World-Wide Telescope 415
 - requisitos da 416
 - grafo de precedência 809–810
 - grupo, de computadores 13, 49–50, 923
 - grupo 233
 - aberto 235
 - fechado 235
 - gerenciamento de membros 237, 771
 - grupo de objetos 234
 - grupo de processos 234
 - grupo não sobreposto 235
 - grupo sobreposto 235
 - membros 233
 - modo de visualização 237, 771–775
 - sobreposto 658
 - grupo de *multicast* 170
 - grupo de objetos 234

grupo de processos 234, 238–239
Gryphon 253

H

Hadoop 965
Handle System 570
Hermes 253
heterogeneidade 16, 17, 41, 528–529, 530
 em publicar-assinar 244
 serviço de nomes 573
hiperchamada 322–323
hipervisor 319, 321
histórico 597
 de operações do servidor no protocolo
 requisição-resposta 190
 global 613
Horus 238–239
HTML 27
HTTP 30–31, 95, 96, 106, 107, 125–126, 192–195
 desempenho da 309
 no SOAP 390–391
 sobre conexão persistente 313
hub 105
hub Ethernet 103, 105
hyperlink 26
 físico 871, 873–875
hyperlink físico 871, 873–875

I

IANA (Internet Assigned Numbers Authority) 96, 577–578
IDEA (International Data Encryption Algorithm) 490, 501
identificação automática 855
identificador 566, 568
identificador de grupo de arquivos 535
identificador exclusivo de arquivo (UFID) 530, 551–552
imparcialidade 634
impasse 700–704
 com travas de leitura-escrita 700
 definição 700
 detecção 611, 702
 distribuído *veja* impasse distribuído
 grafo de espera por 700
 prevenção 701, 703
 tempos limites 703
impasse distribuído 743–752
 caminhamento pela arestas 746–752
 prioridades de transação 749
 impasse fantasma 745
inanição 634, 711
indireção 230
infraestrutura como serviço 319
infraestrutura de chave pública 499
infraestrutura do Google 927
 abertura 926
 arquitetura 924–928
 Bigtable 927, 948–954
 buffers de protocolo 927, 929–932
 Chubby 927, 940–947
 confiabilidade 925
 desempenho 926, 928
 escalabilidade 924
 Google File System (GFS) 927, 935–940
 MapReduce 927, 956–960
 publicar-assinar 927, 932–934
 Sawzall 928, 962–963
instrução sensível
 privilegiada 322
 sensível ao comportamento 322
 sensível ao controle 322
 virtualização 322
integridade
 da comunicação confiável 71–72
 de entrega *multicast* 236, 647
 de filas de mensagem 255
 no consenso e problemas relacionados 660–662
interação espontânea 11, 825–826
interação espontânea 825–826
interceptação
 Fractal 377
interface 195–198, 204, 205
 interface de serviço 196
 linguagem de definição de interface (IDL) 197
 interface de invocação dinâmica 212, 350–351, 397
 interface de serviço 196, 395
interface exigida
 componente 360–361
interface fornecida
 componente 360–361
interface remota 207, 208
 na invocação de método remoto Java 217
interligação de sistemas abertos *veja* OSI Reference Model
intermediário de mensagem, em filas de mensagem 256
International Atomic Time 599
Internet 8, 96, 106–129
 protocolos de roteamento 113–116

- inter-redes 88, 94, 103–105
 interrupção de *software* 295
 intranet 8, 14
 inundação, no roteamento baseado em conteúdo 250
 invasor, ameaças à segurança por um 75–76
 invasor 73–74
 invoca uma operação 15
 invocação assíncrona 313
 no CORBA 347
 persistente 313
 invocação de método remoto 43–44, 186
 coleta de lixo distribuída 215–216
 CORBA 341–358
 desempenho da *veja* mecanismo de invocação, desempenho do
 despachante 210–211
 download de classes 212
 esqueleto 210–211
 estudo de caso Java 217–225
 implementação 209–215
 interface de invocação dinâmica 212
 invocação dinâmica 212
 método de fábrica 213
 módulo de comunicação 209
 módulo de referência remota 210
 nula 305
 objeto de fábrica 213
 passagem de parâmetro e resultado em Java 218
 proxy 210–211
 servente 210–211
 vinculador 213
 invocação de método remoto Java 217–225
 callback 223
 classes serventes 221
 download de classes 219
 interface remota 217
 passagem de parâmetro e resultado 218
 programa cliente 222
 programa servidor 221
 projeto e implementação 224–225
 RMIClientRegistry 220
 servente 221
 uso de reflexão 224
 invocação dinâmica 212
 em serviços Web 387
 no CORBA 350–351
 invocação remota 15, 42–43
 chamada de procedimento remoto 43–44, 186, 195–204
 infraestrutura do Google 929
 invocação de método remoto 43–44, 186, 204–216
 protocolo de requisição-resposta 42–43, 186–195
 IOR *veja* referência de objeto interoperável CORBA
 IP 95, 110–123
 API 147–158
 IPC *veja* comunicação entre processos
 IPv4 108
 IPv6 90, 105, 118–121
 ISDN 95
 ISIS 775
 isolamento
 no Xen 321
- J**
- Java
 reflexão 163–164
 serialização de objetos 162–165
 thread *veja* *thread*, Java
 Java Messaging Service (JMS)
 cliente JMS 258
 conexão 259
 consumidor de mensagem 260
 destino JMS 259
 fábrica de conexão 259
 mensagem JMS 258
 produtor de mensagem 260
 provedor de JMS 258
 transação 259
 JavaSpaces 271–274
 arrendamentos 271–272
 modelo de programação 271–272
 objetos no 271–272
 transação 271–272
 JetSend 841
 JGroups 238–242
 bloco de construção 238–239, 241
 canal 238–240
 grupo de processos 238–239
 pilha de protocolos 239, 241–242
 Jini 272–273
 arrendamentos 216
 especificação de evento distribuído 247
 serviço de descoberta 832–833
 jitter 63–64
- K**
- Kazaa 425
 Kerberos 505–510

L

L²imbo 840
LAN *veja rede, local*
lançar exceção 206
largura de banda 63–64
latência 63–64
LDAP *veja Lightweight Directory Access Protocol*
leecher (parasita), no BitTorrent 907
leitura suja 688
Lightweight Directory Access Protocol 591–592
Linda 265
 Bauhaus Linda 267
 vários espaços de tuplas 267
Linear-Bounded Arrival Processes (LBAP) 892
linearização 614
linguagem de definição de interface (IDL) 197
linguagem de definição de interface 208
 CORBA 341–344
 em serviços Web 400–404
 exemplo da IDL CORBA 197
 exemplo da RPC Sun 201
links 26
lista de controle de acesso 480, 528–529
lista de intenções 751–752, 757
localização
 absoluta 854
 autenticação com base na 863
 física 855
 percepção 852–857
 pilha 856
 relativa 854
 semântica 855
log 753–755
Log-Structured File Storage (LFS) 560–561

M

Mach 317
MAN *veja rede, metropolitana*
mapeador de porta 203
MapReduce 927, 956–960
 arquitetura 958–960
 interface 956–958
máquina de estado 268, 769
máquina virtual 17
mascarando falhas 21–22, 70–71
mashup (na arquitetura orientada a serviços) 414
mechanismo de invocação 282
 assíncrona 313
dentro de um computador 309–311
desempenho de saída 307

desempenho do 305–311
escalonamento e comunicação como parte do 282
latência 306
suporte do sistema operacional para 303–311
mechanismo de segurança 464
média tolerante a falhas 603
MEDYM 253
Meghdoot 253
memória compartilhada distribuída 45, 261–265, 523
 Apollo Domain 262–263
 comparação com passagem de mensagens 264
memória pseudo-física, no Xen 325
mensagem
 destino 148
 requisição 188
 resposta 188
mensagem de pulsação 442–443
mensagem de requisição 188
mensagem de resposta 188
mensagem de resposta perdida 190
Message Passing Interface (MPI) 178–180
Message-Oriented Middleware (MOM) 254
metadados 526
método de fábrica 213, 355–356
micronúcleo 315–318
 comparação com núcleo monolítico 316
micronúcleo L4 318
Microsoft Virtual Server 320
middleware 17, 52–53, 58–62
 categorias 59–60
 componente 59–61, 336, 358–365
 filas de mensagem 59–61
 limitações 59–61
 objeto distribuído 59–61, 336–340
 peer-to-peer 59–61
 publicar-assinar 59–61
 serviços Web 59–61
 servidor de aplicação 59–61, 363–364
 suporte do sistema operacional para 281
middleware de Grade 417–418
mídia contínua 12, 886
mistura (na criptografia) 487
mistura 865
mobileIP 120–123
mobilidade
 física *versus* lógica 822
 transparência 24, 527
modelo
 de arquitetura 38, 40–59
 falha 67–72

- física 38–40
 fundamental 38, 61–77
 interação 62–68
 segurança 71–77
 modelo cliente-servidor 46
 modelo de falha 67–72
multicast IP 171
 protocolo de confirmação atômica 731–732
 protocolo de requisição-resposta 189
 TCP 155–156
 transação 679
 UDP 151–152
 modelo de interação 38, 62–68
 modelo de objeto distribuído 207
 comparado com serviços Web 393–394
 modelo de proteção *sandbox* 468
 modelo de segurança 38, 71–77
 modelos de arquitetura 38, 40–59
 modelos de consistência 261–262
 modelos de falha 38
 modelos físicos 38–40
 modelos fundamentais 38, 61–77
 modo de visualização, de grupo *veja* grupo, modo de visualização 773–775
 modo supervisor 285
 modo usuário 285
 módulo de referência remota 210
 moldagem de tráfego (para dados multimídia) 893
 monitor de máquina virtual 319
 montagem de pacotes 95
 morcego ativo 854
 MTU *veja* unidade de transferência máxima
multicast 233, 646–659
 atômico 651
 básico 647
 com ordem total 174
 confiável 173–174, 236, 647–651
 direcionado 850
 em nível de aplicativo 908
 entrega com ordem causal 651, 657
 entrega com ordem FIFO 651, 653
 entrega com ordem total 651, 654
 não confiável 171
 operação 169
 ordem causal 236
 ordem FIFO 236
 ordem total 236
 ordenado 173–174, 236, 651–659
 para dados altamente disponíveis 170
 para dados replicados 174
 para grupo sobrepostos 658
 para notificações de evento 170
 para serviços de descoberta 170
 para tolerância a falhas 169, 779, 780
multicast IP 106, 170–174, 236, 649
 alocação de endereço 171
 API Java 172–173
 modelo de falha 171
 roteador 170
 multimídia 882–913
 adaptação de fluxo 899
 baseada na Web 884
 compactação de dados 887
 controle de admissão 896–897
 estratégia de sistema final 909
 fluxo 882–886
 largura de banda de recurso 883
 larguras de banda típicas 886
 mídia contínua 12, 886
 taxa de rajada de fluxo 892
 tempo de reprodução 90
 multiprocessador
 memória compartilhada 283
 Non-Uniform Memory Access (NUMA)
 262–263
 memória distribuída 264
- ## N
- não repúdio 470, 494
Napster 425, 428–431, 435–436
NAT *veja* Network Address Translation
 navegação
 controlada pelo servidor 575
 multicast 575
 navegação por satélite 852
veja Sistema de Posicionamento Global (Global Positioning System – GPS)
Near Field Communication (NFC) 834, 855
 negação de serviço
 ataque de tortura de privação do sono 858
 negócios financeiros 6–8
Nemesis 317
Network Address Translation (NAT) 116
Network File System (NFS) 430–431, 454–456, 529–530, 536–548
 aprimoramentos 557
 autenticação Kerberos 544–545
 Automounter 541
benchmarks 545–546
 desempenho 545–546
 montagem incondicional e condicional 540

- serviço de montagem 538–539
sistema de arquivos virtual (VFS) 536
Spritley NFS 557
v-node 537
WebNFS 558
- Network Information Service (NIS) 780
Network Time Protocol 603–606
NFS *veja* Network File System
NIS *veja* Network Information Service
NNTP 106
nó 41–42
nome 566, 568
 componente 571
 desvinculado 570
 prefixo 571
 puro 566
nome *host* 568
nonce 504
NQNFS (Not Quite NFS) 558
NTP *veja* Network Time Protocol
núcleo 285
 monolítico 315
número do *i-node* 537
- 0**
- objeto 41–42
 assinatura de método 205
 ativação 213
 ativo 213
 distribuído 206
 modelo 207
 exceção 205
 fábrica 213
 instanciação 205
 interface 205
 localização 215
 modelo 205
 CORBA 341
nenhuma instanciação em serviços Web 393–394
passivo 213
persistente 214
proteção 72–73
referência 204, 205
referência remota 207
remoto 207
- objeto ativo 213
objeto de fábrica 213
objeto distribuído 41–42, 59–61, 206, 336, 337
 ativação 339
 comunicação entre objetos 339
 desativação 339
 função da classe 338
 gerenciamento de ciclo de vida 339
 herança de interface 339
 middleware 336–340
 persistência 339
 objeto passivo 213
 objeto persistente 214
 objeto remoto 207
 ativador 213
 instanciação 208
OceanStore 451–456
olho por olho no BitTorrent, BitTorrent
 olho por olho 907
- OMG (Object Management Group) 340
Open Mobile Alliance (OMA) 868
OpenCOM 374
OpenStreetMap 965
operação assíncrona 311–314
operação atômica 676
operação desconectada 767, 792, 800–801, 824
operação idempotente 190, 198–199, 532
operações comutativas 782
operações conflitantes 686
operações de arquivo
 no modelo de serviço de arquivo plano 532
 no modelo de serviço de diretório 534
 no servidor NFS 4, 538
 no UNIX 526
operações que causam travamento 148
ordem *big-endian* 158
ordem causal 236, 606–607
 de entrega *multicast* 651
 de tratamento requisição 770
ordem de carimbo de tempo 711–718
 conflitos de operação 712
 em transação distribuída 741
 regra de escrita 713
 regra de leitura 714
 versão múltipla 715–718
ordem de emissão 148
ordem *little-endian* 158
ordem total 236, 597, 609
 de entrega *multicast* 651
 de tratamento de requisição 770
ordenação FIFO 236
 de entrega *multicast* 651
 de tratamento de requisição 770
OSGi 374
OSI Reference Model 94
overlay de despachante 933–934

P

- pacotes *veja rede, pacotes*
 padrões 51–59, 916
 padrões arquitetônicos 51–59
 - camadas físicas 52–53
 - camadas lógicas 51–52
 - corretagem 57–58
 - proxy* 57–58
 - reflexão 58–59
 padrões IEEE 802 128–129
 PageRank 919
 páginas Web dinâmicas 30–32
 parada por falha 68–70
 paradigmas de comunicação 42–45
 - comunicação entre processos 42–43
 - comunicação indireta 43–44
 - invocação remota 42–43
 Parallels 320
 parâmetros de entrada 196
 parâmetros de saída 196
 paravirtualização 320, 322–323
 participação de rede 631, 808–810
 - primária 772
 - virtual 810–814
 partícula 824
 passagem de mensagens 146, 178
 - Message Passing Interface (MPI) 178–180
 Paxos 843, 943–946
 - Multi-Paxos 946*peer-to-peer* 46–48, 59–61, 424–460
 - BitTorrent 906
 - CoolStreaming 910
 - e posse de direitos de cópia 429–430
 - espaço de tuplas 271
 - middleware* 425, 430–434
 - peer-to-peer* estruturado 445–446
 - peer-to-peer* não estruturado 445–448
 - publicar-assinar 249
 - sobreposição de roteamento 433–437
 - ultrapares 447–448
 - peer-to-peer* não estruturado 445–448
 - fofoca 447–448
 - Gnutella 447–449
 - pesquisa em anel expandida 447–448
 - protocolo epidêmico 447–448
 - random walk* 447–448
 - perfil Composite Capabilities / Preferences (CC/PP) 868
 - persistência
 - objeto distribuído 339
 - pesquisa em anel expandida 447–448
 - pesquisa na Web 3–5
 - PGP *veja Pretty Good Privacy*
 - Plan 9 573
 - plataforma 52–53, 281
 - plataforma como serviço 319, 922
 - pó inteligente *veja partícula*
 - política de segurança 464
 - ponte 104
 - POP 106
 - porta 96
 - porta de servidor 150
 - posicionamento 48–52
 - POTS (sistema telefônico antigo) 91
 - PPP 95, 106, 108
 - presença na Web 872
 - Pretty Good Privacy (PGP) 502
 - principal 72–73
 - princípio do limite 828
 - princípio fim-a-fim 908
 - privacidade 826, 856, 857, 864–866
 - proxy* 865
 - procedimento *stub* 200
 - no CORBA 349–350
 - procedimento *stub* de servidor 200
 - processador virtual 301–302
 - processamento de eventos complexos 7
 - em publicar-assinar 248
 - processo 41–42, 286–303
 - ameaças ao 73–74
 - correto 632
 - criação 289–291
 - custo da criação 295
 - em nível de usuário 285
 - multithread* 286, 287
 - programação orientada a dados 386, 837–844
 - projeto SETI@home 427–428
 - promessa 313
 - promessa de *callback* 552–553
 - propagação de atualização ávida 804
 - propagação de atualização preguiçosa 803
 - propriedade da segurança 615
 - propriedade da subsistência 615
 - propriedade uniforme 650
 - propriedades ACID 681, 720
 - proteção 284–285
 - e linguagem fortemente tipada 285
 - pelo núcleo 285
 - protocolo 92–98
 - ARP 111–112
 - camada de aplicação 93, 95
 - camada de apresentação 95

- camada de enlace de dados 95
camada de rede 95
camada de sessão 95
camada de transporte 95
camada física 93, 95
camada inter-redes 94
camadas 93
composição dinâmica 304
conjunto 94
FTP 95, 96, 106, 127–128
HTTP 95, 96, 106, 107, 125–126
IP 95, 110–123
IPv4 108
IPv6 90, 105, 118–121
mobileIP 120–123
NNTP 106
pilha 94, 304
POP 106
PPP 95, 106, 108
SMTP 95, 106
suporte do sistema operacional para 304
TCP 95, 122–125
TCP/IP 106
transporte 92
UDP 95, 107, 122–123
- protocolo antientropia 791, 792
protocolo de confirmação atômica 728–740
 modelo de falha 731–732
 protocolo de confirmação de duas fases 731–732
- protocolo de confirmação de duas fases 730–735
 ações de tempo limite 733–734
 desempenho 734–735
 recuperação 758–761
 transação aninhada 735–740
 confirmação hierárquica 738
 confirmação plana 739
- protocolo de criptografia misto 476, 493
protocolo de *handshake*, no TSL 512
protocolo de informações de roteamento (RIP) 100
protocolo de Internet *veja IP*
- protocolo de requisição-resposta 42–43, 186–195
 doOperation 187, 189
 getRequest 187
 histórico de operações do servidor 190
 identificadores de mensagem 189
 mensagem de resposta perdidas 190
 modelo de falha 189
 protocolos de troca 190
 sendReply 187
 tempo limite 189
 uso de TCP 191
- protocolo de resolução de endereços (ARP)
 111–112
- protocolo de transporte 92
protocolo epidêmico 447–448
protocolo Internet Inter-ORB (IIOP) 352
protocolo Medium Access Control (MAC) 95,
 131, 134, 136
- protocolo Needham–Schroeder 504–505
protocolo resurreição do patinho 863
protocolos de roteamento Bellman–Ford 99
 proxy 57–58, 210–211
 dinâmico 397
 em serviços Web 387, 395–397
 no CORBA 349–350
 pseudônimo 865
publicar-assinar 45, 59–61, 242–253
 anúncio 245
 aplicações 243
 assinante 243
 assinatura 243
 baseado em assunto 246
 baseada em canal 246
 baseada em conceito 248
 baseada em conteúdo 247
 baseado em tipo 247
 baseado em tópico 246, 933–934
 características 244
 exemplo de sistemas 253
 filtro 245
 função da fofoca 253
 garantias de entrega 245
 implementação 248–253
 implementação centralizada 248
 implementação decentralizada 248
 implementação *peer-to-peer* 249
 infraestrutura do Google 927, 932–934
 modelo de programação 245–248
 objetos de interesse 247
 processamento de eventos complexos 248
 publicador 243
 roteamento baseado em conteúdo 250
 publicador, em publicar-assinar 243
- Q**
- QoS *veja* qualidade do serviço
quadro negro compartilhado
 implementação no CORBA 353–354
 implementado em serviços Web 394
 implementado na RMI Java 217–224
 interface IDL CORBA 343
 invocação dinâmica 212

- qualidade do serviço 25, 41
 controle de admissão 890
 gerenciamento 882, 887–897
 negociação 889–896
 parâmetros 890
- R**
- Radio Frequency IDentification *veja* RFID
random walk 447–448
 rastreador, no BitTorrent 906
 rastreamento 852
 e privacidade 856, 864
 Real Time Transport Protocol (RTP) 90
 reconhecimento de contexto 844–857
 em publicar-assinar 248
 recuperação 688–690, 761
 cancelamento em cascata 689
 de cancelamento 688
 de protocolo de confirmação de duas fases 758–761
 escrita prematura 689
 execuções restritas 690
 leitura suja 688
 lista de intenções 751–752, 757
 log 753–755
 status de transação 752, 757
 transações aninhadas 759–761
 versões de sombra 756–757
 recuperação de falha 22–23
 recuperação inconsistente 684, 695
 recurso 2
 compatilhamento 14–16, 424
 invocação em 282
 rede
 ad hoc 134–135, 848
 análise de tráfego 865
 ATM (Asynchronous Transfer Mode) 88, 90, 92, 95, 102, 130
 camada 95
 comutação de circuitos 91
 controle de congestionamento 102
 CSMA/CA 136–137
 CSMA/CD 131
 endereço de transporte 96
 fluxo de dados 90
 frame relay 91
 gateway 89
 IEEE 802.11 (WiFi) 129, 134–138, 515–517
 IEEE 802.15.1 (Bluetooth) 129, 138–141
 IEEE 802.15.4 (ZigBee) 130
 IEEE 802.16 (WiMAX) 130
 IEEE 802.3 (Ethernet) 104, 129–135
 IEEE 802.4 (Token Bus) 129
 IEEE 802.5 (Token Ring) 129
 Internet 96, 106–129
 interplanetária 850
 largura de banda total do sistema 83
 latência 83
 local 86
 longa distância 87
 metropolitana 87
 montagem de pacotes 95
multicast IP 106
 pacotes 89
 padrões IEEE 802 128–129
 parâmetros de desempenho 83
 ponte 104
 porta 96
 protocolo *veja* protocolo
 requisitos 83–85
 requisitos de confiabilidade 84
 requisitos de escalabilidade 84
 requisitos de segurança 85
 roteador 104
 roteamento 87, 98–102, 113–116
 roteamento de IP 113
 taxa de transferência de dados 83
 TCP/IP 106
 tolerante a atraso 850
 tolerante a rompimento 850
 troca de pacotes 91
 tunelamento 105
 Ultra Wide Band 855
 rede *ad hoc* 134–135, 848
 móvel 849
 rede Bluetooth 129, 138–141
 rede de distribuição de conteúdo 176, 909
 rede de intermediários 249
 rede de sobreposição 175
 para multimedia 908
 rede de telefonia móvel GSM 88
 rede de tempo real 134–135
 rede espontânea 821
 serviço de descoberta 584
 rede Ethernet 104, 129–135
 rede IEEE 802.11 (WiFi) 129, 134–138
 segurança 515–517
 rede IEEE 802.15.1 (Bluetooth) 129, 138–141
 rede IEEE 802.15.4 (ZigBee) 130
 rede IEEE 802.16 (WiMAX) 130
 rede IEEE 802.3 (Ethernet) 104, 129–135

- rede IEEE 802.4 (Token Bus) 129
rede IEEE 802.5 (Token Ring) 129
rede local *veja* rede, local
rede privada virtual (VPN) 85, 128–129
rede Token Bus 129
rede Token Ring 129
rede tolerante a atraso *veja* rede tolerante a rompimento 850
rede tolerante a rompimento 176, 850
rede WiFi 129, 134–138, 515–517
rede WiMAX 130
rede ZigBee 130
redundância 22–23
Redundant Arrays of Inexpensive Disks (RAID)
560–561
referência de objeto remoto 168, 207, 208
comparada com URI 393–394
no CORBA 350–351
reflexão 58–59
em Java 163–164
Fractal 376
na invocação de método remoto Java 224
OpenCOM 374
região *veja* espaço de endereçamento
registros históricos
sistemas de arquivo distribuídos 522
surgimento da criptografia moderna 465
relação representa (na segurança) 482
relógio
acordo 599
computador 64–65, 597
defeituoso 600
desvio 64–65, 598–599
distorção 604
exatidão 600
global 2
lógico 608
matriz 610
monotonicidade 598–599
precisão 599
resolução 598–599
sincronização *veja* sincronização de relógios
vetorial 609
rende-zvous, no roteamento baseado em conteúdo
252
replicação 49–50, 447–448, 765–817
ativa 780–782
backup primário 778–780
Chubby 943
consenso de quorum 803, 809–811
cópias disponíveis 803, 805–809
com validação 809–810
de arquivos 528–529
em espaços de tuplas 238–270
Google File System (GFS) 937
partição virtual 803, 810–814
transacional 802–814
transparência 24, 767
replicação passiva *veja* replicação, *backup*
primário
repositório de implementações, no CORBA
349–350
repositório de interfaces, no CORBA 349–350
representação externa de dados 158–164
resolução de nomes 566, 569, 573–574, 874
resolvedor 874
Resource Reservation Protocol (RSVP) 90, 896
REST (REpresentational State Transfer) 384, 386,
418–419
resumo de mensagem 476
RFC 18
RFID (Radio Frequency IDentification) 845, 855,
864, 874
RIP–1 101, 113
RIP–2 113
RMI *veja* invocação de método remoto
RMIRegistry 220
roteador 103, 104
roteamento *ad hoc* 824
roteamento baseado em conteúdo 250
anúncio 252
filtragem 251
inundação 250
rende-zvous 252
roteamento entre domínios sem classes (CIDR)
115, 425
roteamento *veja* rede
roteamento
RPC Firefly 308
RPC leve 309–311
RPC Sun 201–204, 536
linguagem de definição de interface 201
portmapper 203
representação externa de dados 203
rpcgen 201
RPC *veja* chamada de procedimento remoto
RR 190
RRA 190
RSVP *veja* Resource Reservation Protocol

S

Sawzall 928, 962–963
 Scribe 253
 seção crítica 633
 Sector/Sphere 965
 Secure Sockets Layer *veja* Transport Layer Security
seeder (semeador), no BitTorrent 906
 segurança 18, 19
 “Alice e Bob” nomes de protagonistas 466
 ameaças: vazamento, falsificação, vandalismo 466
 ameaças de código móvel 467
 ataque de falsificação de mensagem 467
 ataque de intromissão 467
 ataque de mascaramento 467
 ataque de negação de serviço 471
 ataque de reprodução 467
 ataque do homem no meio 467
 base de computação confiável 472
 diretrizes de projeto 467
 em filas de mensagem 256
 em Java 468
 modelos de vazamento de informações 468
 segurança da XML 406–411
 Transport Layer Security (TLS) 511–515
 segurança Java 468
 segurança WEP WiFi 515–517
 Semantic Web 844
 semântica de atualização 528–529, 554–555
 semântica de invocação
 exatamente uma vez 198
 no máximo uma vez 198–199
 pelo menos uma vez 198–199
 talvez 198–199
send não bloqueante 148
 sensor 823, 844–857
 fusão 846
 localização 852–857
 modo de erro 845
 rede, sem fio 848–852
 sequenciador 654
 serialização 162
 buffers de protocolo 929
 serialização de objetos, em Java 162–165
 série 614
 servente 210–212, 221, 348–349, 354–355, 394, 400

serviço 15
 altamente disponível 782–801
 tolerante a falhas 767, 775–782
 serviço de arquivo plano 530–533
 serviço de autenticação 506
 serviço de descoberta 584, 828–833
 Jini 832–833
 sem servidor 831
 serviço de descoberta de rede 830
 serviço de diretório 533–535, 584
 atributo 584
 serviço de descoberta como 828
 UDDI 404–406
 serviço de diretório X.500 588–592
 árvore de informações de diretório 589
 LDAP 591–592
 serviço de localização 215
 serviço de nomes 566–593
 heterogeneidade 573
 navegação 574–575
 replicação 579–580
 uso de cache 576, 579–580
 uso de Chubby 944
 serviço de nomes Spring 573
 serviço de vídeo sob demanda 884
 serviço tolerante a falhas 767, 775–782
 serviços Web 32–33, 42–43, 59–61, 384–399
 baixo acoplamento 385
 comparados com CORBA 398, 399
 comparados com modelo de objeto distribuído 393–394
 contêiner de servlets 399
 coordenação 411–414
 coreografia 412–413
 descrição de serviço 400–404
 despachante 398
 esqueleto 398
 infraestrutura 383
 interface de invocação dinâmica 397
 interface de serviço in Java 395
 invocação dinâmica 387
 modelo 393–394
 padrões de comunicação 384
 programa cliente Java 395–396
 programa servidor Java 395
 proxy 387, 395–396
 proxy dinâmico 397
 REST (REpresentational State Transfer) 384–386
 sem serventes 394

- serviço de diretório 404–406
SOAP 384, 387–394
SOAP com Java 394–398
Uniform Resource Identifier 382
servidor 15
arquitetura *multithread* 293
desempenho de saída 292
multithread 292
pessoal 825
servidor blade 14
servidor de aplicação 59–61, 360–361, 363–364
servidor de arquivos de vídeo Tiger 901–906
servidor de autenticação 504
servidor pessoal 825
servidor sem estado 533
servlet 298
Session Initiation Protocol 885
Siena 253
Simple Public-Key Infrastructure (SPKI) 500
simulação
de sistema operacional 316
sincronização, de operações do servidor 678
sincronização de mídia 885
sincronização de relógios 599–606
algoritmo de Berkeley 603
algoritmo de Cristian 601–603
em um sistema síncrono 601
externa 599
interna 599
Network Time Protocol 603–606
SIP *veja* Session Initiation Protocol
sistema aberto 314
sistema baseado em eventos distribuídos 6, 45, 242
evento 6
processamento de eventos complexos 7
sistema de arquivos Coda 458–459, 795–801
grupo de armazenamento de volume (VSG) 796
grupo de armazenamento de volume disponível (AVSG) 796
vetor de versão Coda (CVV) 796
sistema de arquivos distribuído Frangipani 562
sistema de arquivos Ivy 455–459
sistema de arquivos sem servidor (xFS) 561–562
sistema de arquivos sem servidor xFS 561–562
sistema de arquivos virtual (VFS) *veja* Network File System (NFS), sistema de arquivos virtual
sistema de disco virtual distribuído Petal 562
sistema de informações geográficas 855
Sistema de Posicionamento Global (Global Positioning System – GPS) 599, 852
sistema de reconhecimento de localização *veja* localização, percepção
sistema distribuído aberto 18
sistema distribuído assíncrono 65–66, 235, 601, 659, 667–668, 944
sistema distribuído síncrono 64–65, 235, 601, 624–625, 630, 633, 659, 663
sistema multimídia distribuído 12
sistema operacional 279–332
arquitetura 314–318
política e mecanismo 314
suporte para comunicação e invocação 303–314
suporte para processos e *threads* 286–303
sistema operacional de rede 280
sistema operacional distribuído 281
sistema V
execução remota 289
suporte para grupos 775
sistema volátil 821–826
sistemas de sistemas 40
sistemas distribuídos Ultra Large Scale (ULS) 40, 924
Skype 177
smartphones 818, 824
SMTP 95, 106
SOAP 384, 387–394
cabeçalho 390–391
com Java 394–398
comunicação confiável 392–393
e *firewalls* 392–393
endereçamento e roteamento 391–392
envelope 389–390
especificação 388–389
implementação Java 397
mensagens 388–389
transporte de mensagem 390–391
uso de HTTP 390–391
sobreposição de roteamento CAN 425, 435–436
sobreposição de roteamento Chord 425
sobreposição de roteamento Kademia 425, 435–436
sobreposição de roteamento Pastry 423, 435–445
sobreposição de roteamento Tapestry 425, 435–436, 444–446
soft state 843
software como serviço 319, 414, 921

- Solaris
 processo leve 301–302
 soquete 149
 conexão 156
 Speakeasy 842
 SPIN 317
 SPKI *veja* Simple Public-Key Infrastructure
spoofing de IP 111–112
 Spritely NFS 557
 SSL *veja* Transport Layer Security
 status de transação 752, 757
 Structure-less CBR 253
 subsistema 316
 Sun Network File System (NFS) *veja* Network File System
 supervisor 284
switch Ethernet 103, 134–135
 Switched Ethernet 134–135
- T**
- tabela de concessões, no Xen 327–328
 tabela de hashing distribuída 176
 no roteamento baseado em conteúdo 252
 tabela de objetos remotos 210
 tag *veja* identificação automática
 TCP 95, 122–125, 153–158
 API 154
 API Java 155–158
 e protocolos de requisição-resposta 191, 308
 modelo de falha 155–156
 TCP/IP *veja* protocolo, TCP/IP
 TEA (Tiny Encryption Algorithm) 488–489, 501
 telefone celular *veja* telefone móvel
 telefone com câmera 834, 861, 874
 telefone móvel 818, 820, 821, 823
 com câmera *veja* telefone com câmera
 e associação segura 859
 e proximidade 854
 perfil de agente de usuário 868
 telefonia na Internet 885
 tempo 595–610
 lógico 606–610
 tempo de reprodução, para elementos de dados multimídia 90
 tempo lógico 67–68, 606–610
 Tempo Universal Coordenado (UTC) 598–599
 tempos limites 68–70
 TERA 253
 terceiro participante confiável 857, 859, 860
- término
 de consenso e problemas relacionados 660–662
 texto cifrado 484
 texto puro 484
thread 41–42, 148, 286, 292–303
 arquitetura *multithread* 293
 C 297
 comparação com processo 294
 custo da criação 295
 em multiprocessador 294
 em nível de núcleo 300–301
 escalonamento 299–300
 implementação 300–303
 Java 297–301
 no cliente 294
 no servidor 290
 POSIX 297
 programação 297–301
 recepção bloqueante 148
 sincronização 298
 trabalhador 293
 troca 296
 TIB Rendezvous 253
 Ticket Granting Service (TGS) 506
 tipo MIME
 uso em HTTP 867
 tiquete, de autenticação 474
 TLS *veja* Transport Layer Security
 tolerância a falhas 22–23
 torrente, no BitTorrent 907
 TOTA (Tuples On The Air) 840
 transação 679–692
 cancelar 682
 closeTransaction 682
 com dados replicados 802–814
 em filas de mensagem 255
 em JavaSpaces 274–275
 em serviços Web 411–412
 equivalência serial 685
 modelo de falha 679
 no Java Messagng Service (JMS) 259
 openTransaction 682
 propriedades ACID 681, 720
 recuperação *veja* recuperação
 transação aninhada 690–690, 759
 ações de tempo limite 740
 confirmação provisória 735–736
 protocolo de confirmação de duas fases 735–740
 recuperação 759–761
 travamento 699

- transação atômica *veja* transação
transação distribuída
 anhinda 728
 controle de concorrência 740–743
 ordem de carimbo de tempo 741
 otimista 742
 travamento 740
 coordenador 730–731
 plana 728
 protocolo de confirmação atômica 728–740
 protocolo de confirmação de duas fases 731–732
 protocolo de confirmação de uma fase 731
transcodificação 867
transferência de estado 774
transformação operacional 793
transição de domínio 296
transparência 23–25, 204
 acesso 23, 527, 546–547
 concorrência 23
 desempenho 24, 527
 em chamada de procedimento remoto 198–199
 escalabilidade 24, 527
 falha 24
 localização 23, 527, 546–547
 mobilidade 24, 527
 mudança de escala 24, 527
 rede 24
 replicação 24, 767
Transport Layer Security (TLS) 107, 511–515
tratamento de falha 21–23
travamento
 em transação distribuída 740
travas 692–706
 aumentando a concorrência 704
 causando impasse *veja* impasse
 Chubby 940
 compartilhadas 694
 duas fases restrito 693
 duas versões 704
 em transação aninhada 699
 exclusivas 692
 gerenciador de travas 696
 granularidade 693
 hierárquicas 705, 706
 implementação 696
 leitura-escrita 694, 695
 leitura-escrita-efetivação 704
 promoção 695
 regras de conflito de operação 694
 tempos limites 703
 travamento de duas fases 693
trecho, no BitTorrent 906
triangulação 853
troca de contexto 296
tunelamento 105
- U**
- UDDI *veja* Universal Description, Discovery and Integration
UDP 95, 107, 122–123, 150–153
 API Java 151–153
 modelo de falha 151–152
 para comunicação de requisição-resposta 308, 312
 uso de 151–152
UFID *veja* identificador exclusivo de arquivo
Ultra Wide Band (UWB) 855
ultrapar, na computação *peer-to-peer* 447–448
UMTS 88
unicast 233
unidade de transferência máxima (MTU) 95, 110–111, 131
Uniform Resource Identifier 568
 comparado com referência de objeto remoto 393–394
 em serviços Web 382
Uniform Resource Locator 28–31, 568
Uniform Resource Name 569
Universal Description, Discovery and Integration (UDDI) 404–406
Universal Plug and Play (UPnP) 829
Universal Transfer Format 163–164
UNIX
 chamada de sistema
 exec 289
 fork 290, 289
 i-node 537
 sinal 295
 upcall 302
URI *veja* Uniform Resource Identifier
URL *veja* Uniform Resource Locator
URN *veja* Uniform Resource Name
uso de cache
 arquivo, escrita direta 542
 arquivos no cliente 542
 arquivos no servidor 541
 de arquivos inteiros 548–549
 Google File System (GFS) 938
 procedimento de validação 543–544, 552–553

UTC *veja* Tempo Universal Coordenado
 UTF *veja* Universal Transfer Format

V

validade
 de comunicação confiável 71–72
 de entrega *multicast* 236, 648
 de filas de mensagens 255
 vazamento de informações como ameaça à
 segurança 468
 Verisign Corporation 500
 versões de sombra 756–757
 vetor de inicialização (para uma cifra) 486
 videoconferência 884, 887
 aplicativo CU-SeeMe 884
 aplicativo iChat AV 884
 aplicativo NetMeeting 884
 vinculador 213
 portmapper 203
 rmiregistry 220
 virtualização
 condição para virtualização (Popek e Goldberg)
 322
 Denali 320
 em nível de sistema 318–320
 hipervisor 319
 instrução sensível 322
 Microsoft Virtual Server 320
 monitor de máquina virtual 319
 Parallels 320
 paravirtualização 320, 322–323
 virtualização total 319
 VMWare 320
 Xen 320–331
 virtualização de sistema 318–320
 virtualização total 319
 VMWare 320
 Voice over IP 177, 885
 aplicativo Skype 885
 aplicativo Vonage 885
 VoIP *veja* Voice over IP
 votação 639
 VPN *veja* rede privada virtual

W

WAN *veja* rede, longa distância
 Web 26–34
 semântica 844
 uso de cache 449–452

Web Services Description Language (WSDL)
 400–404
 elementos principais 400
 interface 402
 operações ou mensagens 401
 parte concreta 403
 serviço 404
 vínculo 403
Webcasting 12
 WebNFS 558
 Websphere MQ 256
 agente de canal de mensagem 257
 canal de mensagem 257
 gerenciador de fila 256
 Message Queue Interface 256
 topologia rede em estrela 258
 Wikipedia
 controle de concorrência 719
 WLAN (rede local sem fio) 88
 WMAN (rede metropolitana sem fio) 88
 World Wide Web *veja* Web
 WSDL *veja* Web Services Description
 Language
 WWAN (rede de longa distância sem fio) 88

X

XDR *veja* RPC Sun
 representação externa de dados
 Xen 320–331, 419–420
 anéis de E/S 327–328
 domínio 322
driver de dispositivo dividido 326–327
 escalonamento 322–325
 gerenciamento de dispositivos 326–329
 gerenciamento de memória virtual 325–326
 hiperchamada 322–323
 hipervisor 321
 isolamento 321
 memória pseudofísica 325
 monitor de máquina virtual 320–330
 tabela de concessões 327–328
 XenoServer Open Platform 329–331
 XML (Extensible Markup Language) 32–33,
 164–168
 analisando 166–167
 elementos e atributos 165–166
 espaços de nome 166–167
 esquemas 167–168
 para interação 844

segurança 406–411
assinatura digital 409–410
criptografia 410–411
requisitos 407
XML canônica 409–410
XSLT (eXtensible Style Language Transformations)
867

Z

Zero Configuration Networking 827
zona de mistura 865