SISTEMAS DISTRIBUÍDOS

Conceitos e Projeto

5ª Edição

George Coulouris
Jean Dollimore
Tim Kindberg
Gordon Blair

bookman

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 Hewllet-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

Catalogação na publicação: Ana Paula M. Magnus – CRB 10/2052

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

5ª 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

bookman

2013

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

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*

Prefácio

Esta 5ª 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 5ª edição

Novos capítulos:

Comunicação indireta: comunicação de grupo, sistemas de publicar-assinar e estudos de caso sobre JavaSpaces, JMS, WebSphere e Messa 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, L^{2}imbo, BitTorrent, End System Multicast.

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

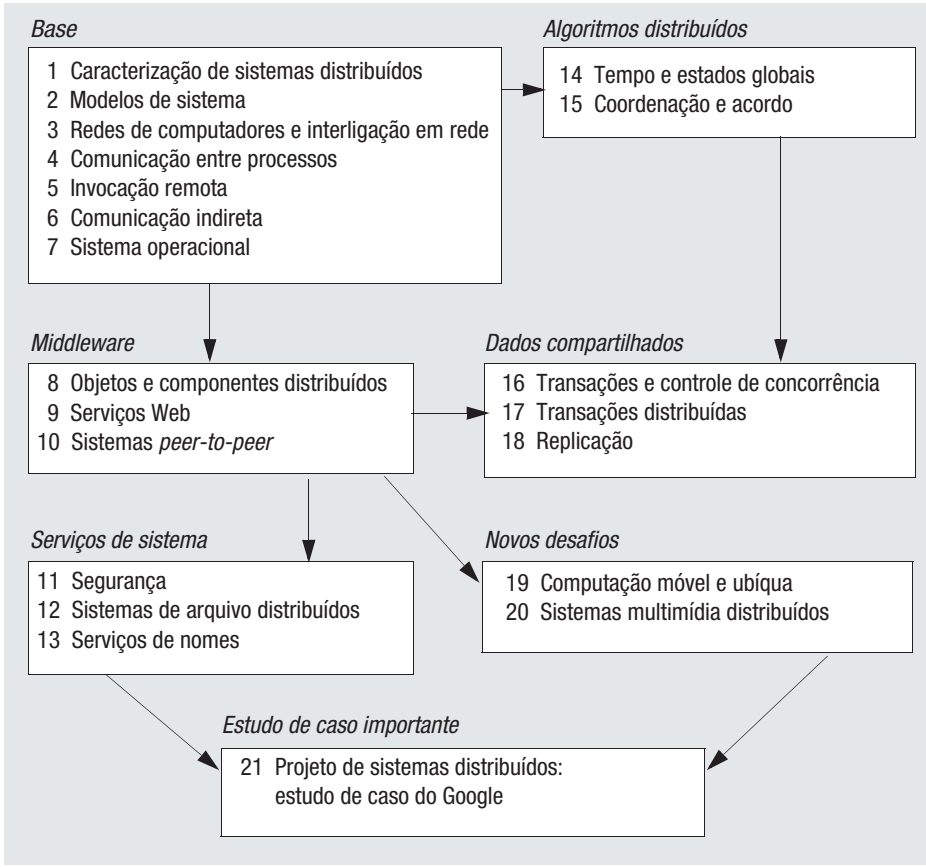
O grupo de capítulos seguinte aborda o importante tópico do *middleware*, examinando diferentes estratégias para suportar aplicativos distribuídos, incluindo objetos e componentes distribuídos, *Web services* e soluções *peer-to-peer* alternativas. Abordamos, em seguida, os tópicos bem-estabelecidos 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 4ª edição

Antes de começarmos a escrever esta nova edição, fizemos um levantamento junto aos professores que utilizavam a 4ª 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 4ª 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	*Reestruturação de material* Nova Seção 1.2: Exemplos de sistemas disttribuídos Seção 1.2.2: Introdução da computação em nuvem MMOGs
2	Modelos de sistema	*Reestruturação de material* 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	*Várias atualizações* 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	*Reestruturação de material* 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	*Várias atualizações* 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: *peer-to-peer* 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 (TOTA e L^2imbo)
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

Finanças e comércio	O crescimento do *e-Commerce*, exemplificado por empresas como Amazon e eBay, e as tecnologias de pagamento subjacentes, como PayPal; o surgimento associado de operações bancárias e negócios *online* e também os complexos sistemas de disseminação de informações para mercados financeiros.
A sociedade da informação	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.
Setores de criação e entretenimento	O surgimento de jogos *online* 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 *streaming*; 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).
Assistência médica	O crescimento da informática médica como disciplina, com sua ênfase nos registros eletrônicos de pacientes *online* 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.
Educação	O surgimento do *e-learning* 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.
Transporte e logística	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.
Ciências	O surgimento das grades computacionais (*grid*) 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.
Gerenciamento ambiental	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 agregrado (*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

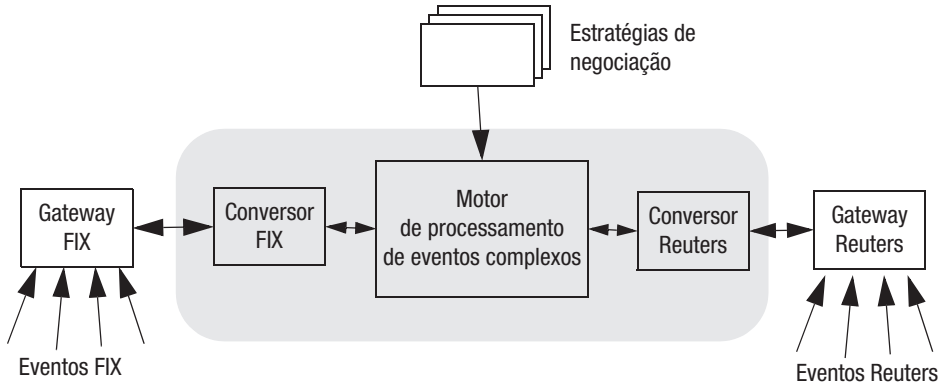


Figura 1.2 Um exemplo de sistema de negócios financeiros.

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 3ª e 4ª 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

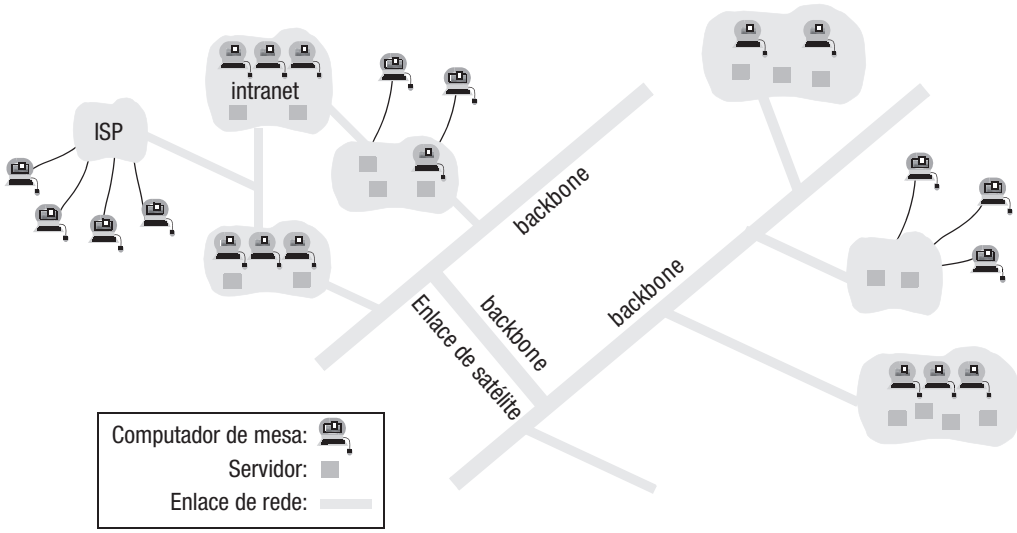


Figura 1.3 Uma parte típica da Internet.

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, *pagers*, 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 entranhados 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.

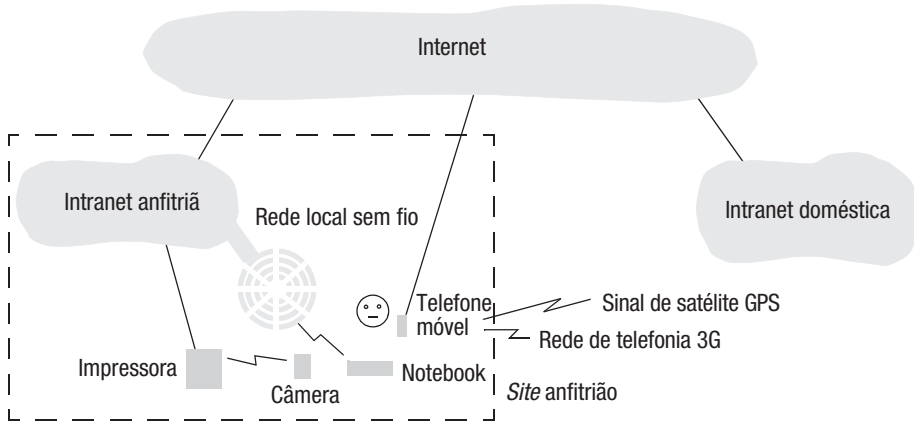


Figura 1.4 Equipamentos portáteis em um sistema distribuído.

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 console 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 usuário 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 I]. 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

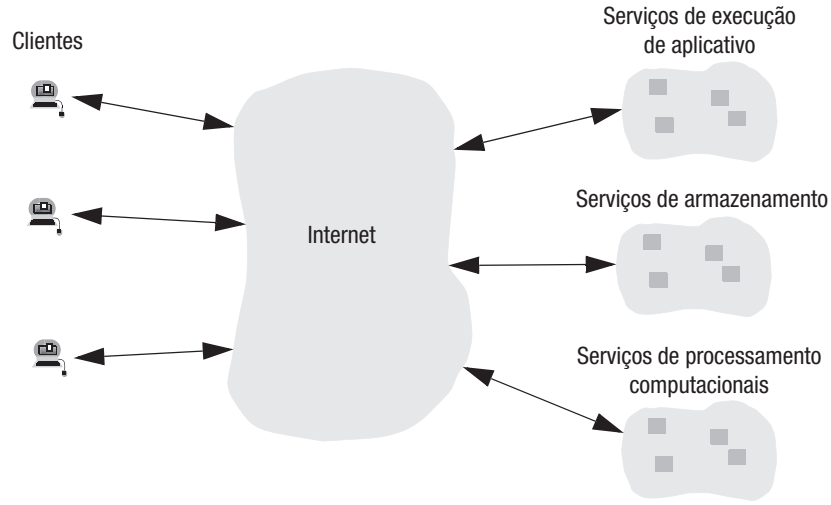


Figura 1.5 Computação em nuvem.

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 fisica 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 *hardware*s.

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 Java*script* 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 formos 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 leiaute 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

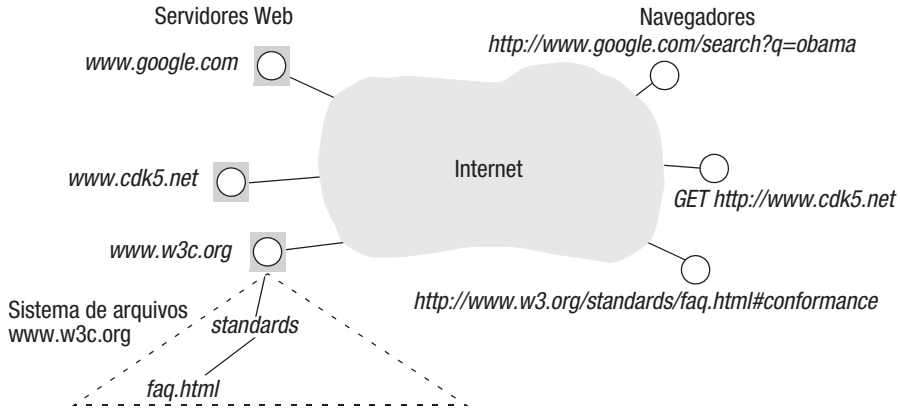


Figura 1.7 Servidores e navegadores Web.

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.

Discutiremos 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* nelas existentes.

HTML • A HyperText Markup Language [www.w3.org II] 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 leiaute criado graficamente. Um texto em HTML típico aparece a seguir:

```
<IMG SRC = "http://www.cdk5.net/WebExample/Images/earth.jpg">          1
<P>                                                                    2
Welcome to Earth! Visitors may also be interested in taking a look at the   3
<A HREF = "http://www.cdk5.net/WebExample/moon.html">Moon</A>.         4
</P>                                                                   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 de 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 *</A>*. 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 subárvores (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:

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

O primeiro URL designa a página padrão fornecida por *www.cdk5.net*. O seguinte 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] 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) (consute 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. Coletivamente, 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 explítica 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 modelo 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.

Sistemas distribuídos	*Primitivos*	*Adaptados para Internet*	*Contemporâneos*
Escala	Pequenos	Grandes	Ultragrandes
Heterogeneidade	Limitada (normalmente, configurações relativamente homogêneas)	Significativa em termos de plataformas, linguagens e *middleware*	Maiores dimensões introduzidas, incluindo estilos de arquitetura radicalmente diferentes
Sistemas abertos	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
Qualidade de serviço	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 comuninação.

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 oferem (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 chamador 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 especificada 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 Processos	Objetos Componentes Serviços Web	Passagem de mensagem Soquetes Multicast	Requisição-resposta RPC RMI	Comunicação em grupo Publicar-assinar 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.

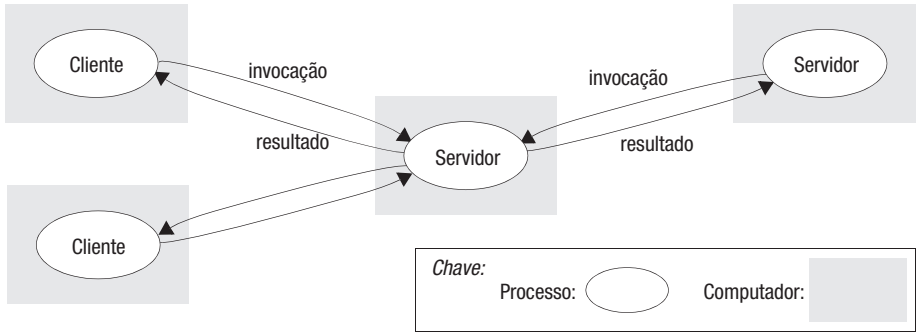


Figura 2.3 Os clientes chamam o servidor individual.

(*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 (*peer*s) executados em diferentes com-

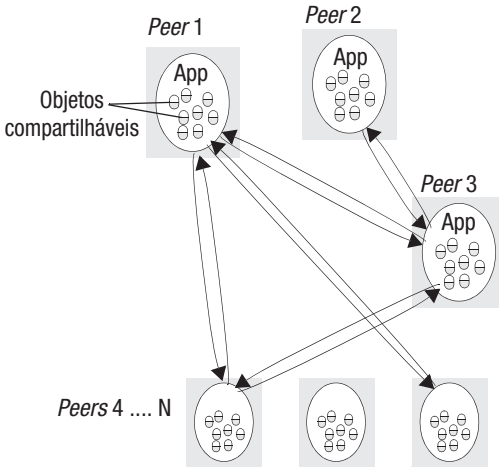


Figura 2.4a Arquitetura *peer-to-peer*.

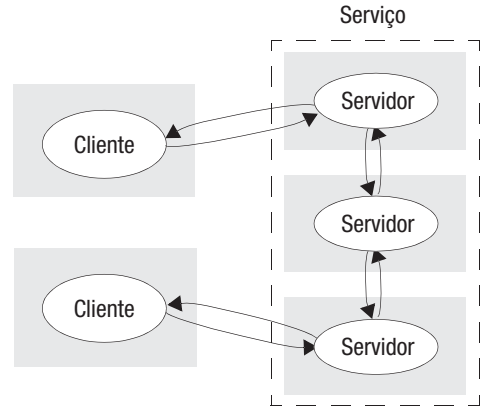


Figura 2.4b Um serviço fornecido por vários servidores.

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 heterogênas 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, mais 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**, 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.

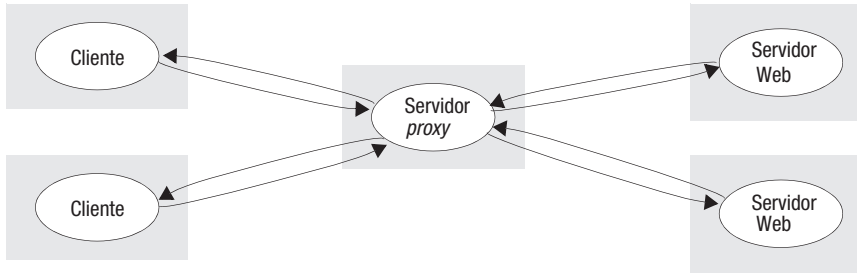


Figura 2.5 Servidor *proxy* Web.

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.

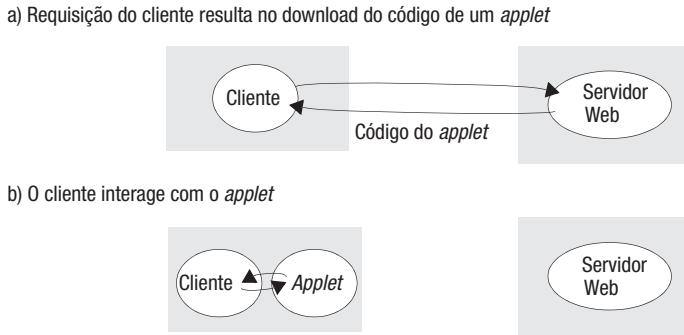


Figura 2.6 *Applets* Web.

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,

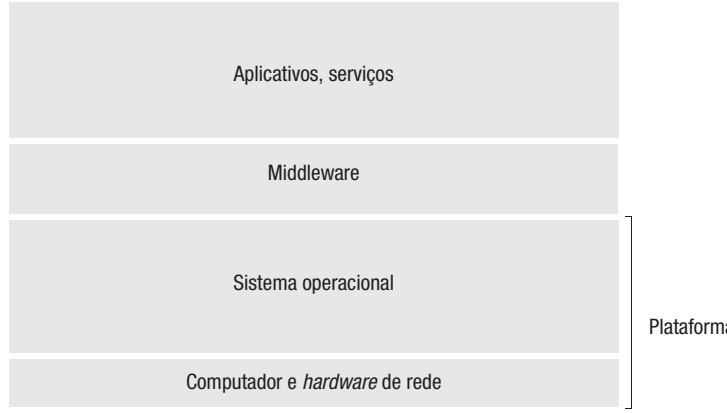


Figura 2.7 Camadas lógicas de serviço de *software* e *hardware* em sistemas distribuídos.

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 intrínsico 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 particionado em n elementos lógicos, cada um mapeado em determinado elemento servidor. Como exemplo, a Wikipedia,

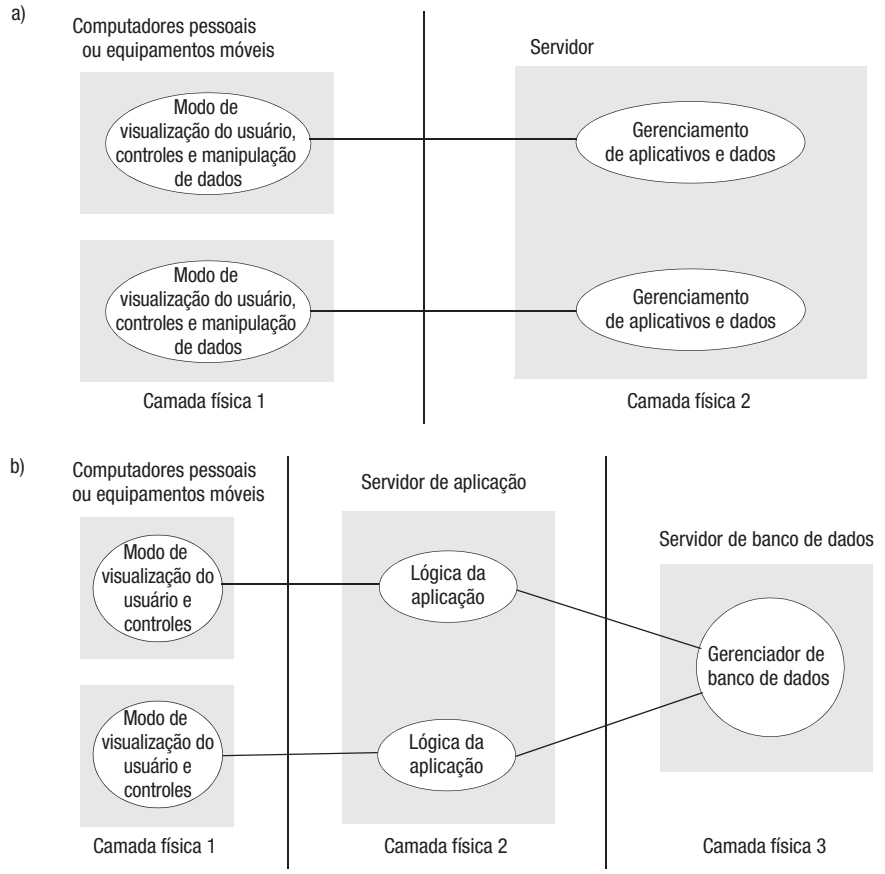


Figura 2.8 Arquitetura de duas e de três camadas físicas.

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 reposicionadas 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-

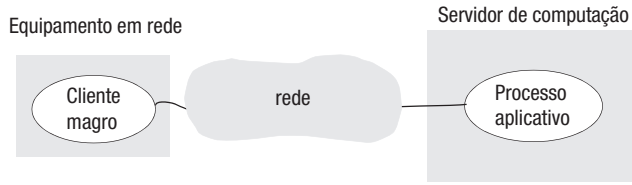


Figura 2.10 Clientes "magros" e servidores.

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].

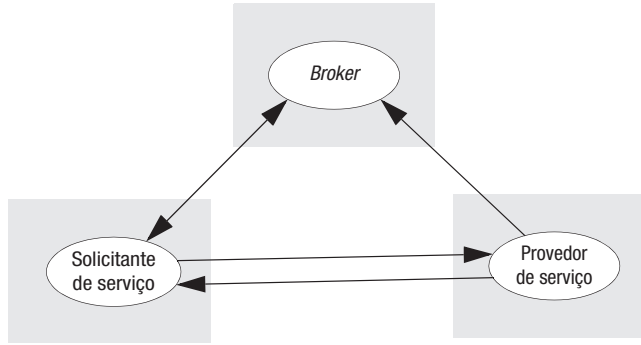


Figura 2.11 O padrão arquitetônico do serviço Web.

* 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 prepararemos 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, incluiundo 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-

Principais categorias	*Subcategoria*	*Exemplos de sistemas*
Objetos distribuídos (Capítulos 5, 8)	Padrão	RM-ODP
	Plataforma	CORBA
	Plataforma	Java RMI
Componentes distribuídos (Capítulo 8)	Componentes leves	Fractal
	Componentes leves	OpenCOM
	Servidores de aplicação	SUN EJB
	Servidores de aplicação	CORBA Component Model
	Servidores de aplicação	JBoss
Sistemas publicar-assinar (Capítulo 6)	–	CORBA Event Service
	–	Scribe
	–	JMS
Filas de mensagem (Capítulo 6)	–	Websphere MQ
	–	JMS
Serviços web (Capítulo 9)	Serviços web	Apache Axis
	Serviços de grade	The Globus Toolkit
Peer-to-peer (Capítulo 10)	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 abstraímos 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 microssegundo. 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. Pepers 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 picossegundo 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.

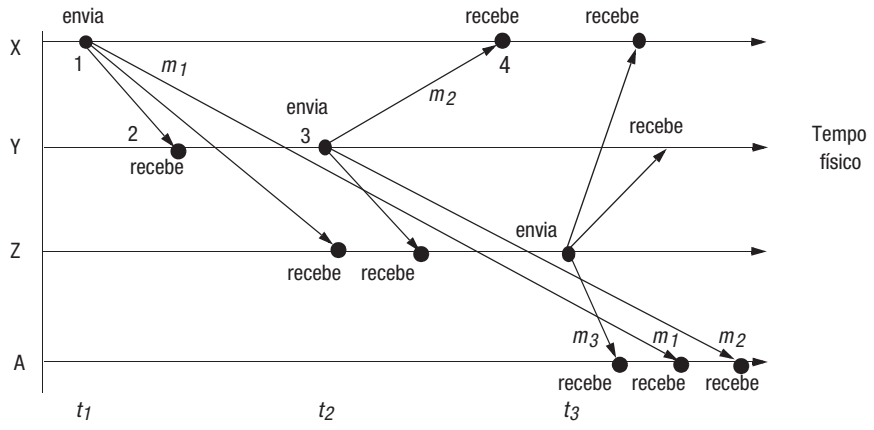


Figura 2.13 Ordenação de eventos no tempo físico.

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, ou 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 *buffer* de envio nunca chega no *buffer* de recepção do destinatário.
Omissão de envio	Processo	Um processo conclui um envio, mas a mensagem não é colocada em seu *buffer* de envio.
Omissão de recepção	Processo	Uma mensagem é colocada no *buffer* de recepção de um processo, mas esse processo não a recebe efetivamente.
Arbitrária (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 predeterminado.

Em um sistema distribuído assíncrono, um servidor sobrecarregado 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

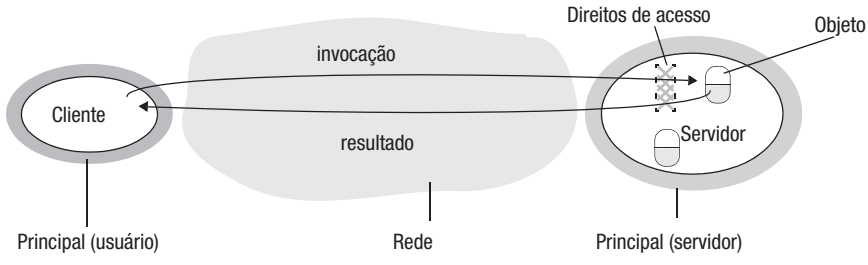


Figura 2.17 Objetos e principais.

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 gere 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*.

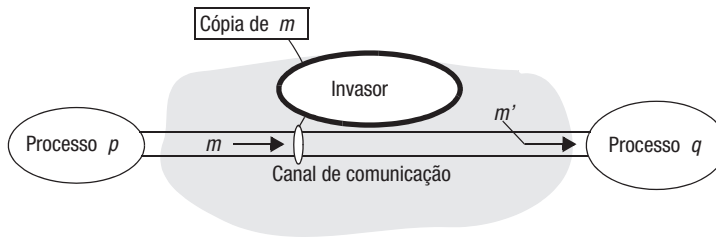


Figura 2.18 O invasor (atacante).

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. Reiteramos essas ameaças como possíveis oportunidades para o invasor romper as atividades dos processos:

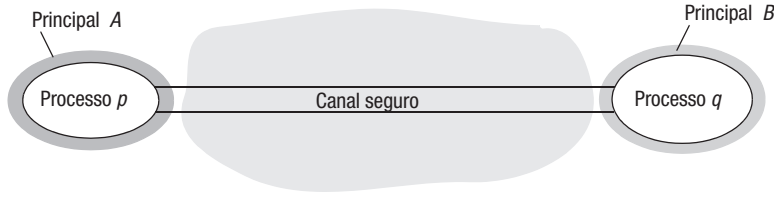


Figura 2.19 Canais seguros.

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 embaraçosas. 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*