2

Modelos de Sistema

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	Objetos	Passagem de mensagem	Requisição-resposta	Comunicação em grupo
Processos	Componentes	Soquetes	RPC	Publicar-assinar
	Serviços Web	Multicast	RMI	Fila de mensagem
				Espaço de tupla
				DSM

Figura 2.2 Entidades e paradigmas de comunicação.

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

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

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

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

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

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

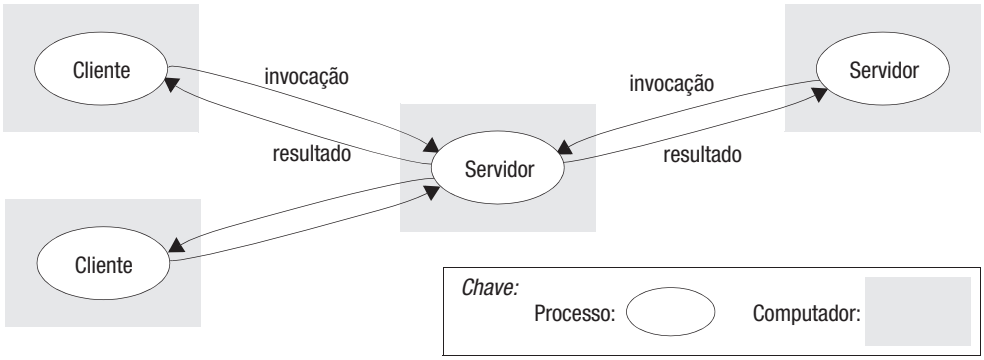

Figura 2.3 Os clientes chamam o servidor individual.

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

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

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

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

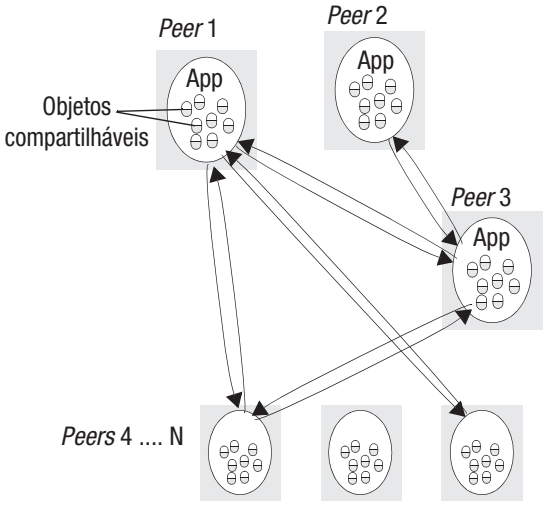

Figura 2.4a Arquitetura *peer-to-peer*.

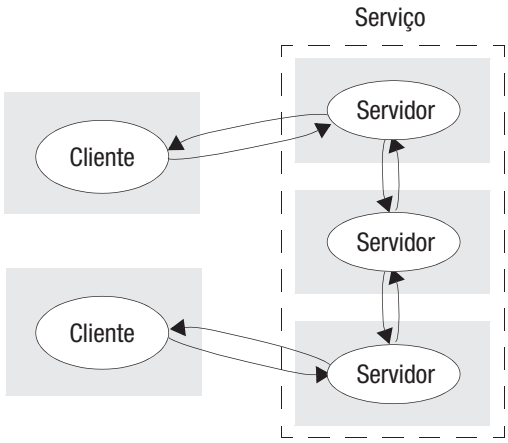

Figura 2.4b Um serviço fornecido por vários servidores.

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.

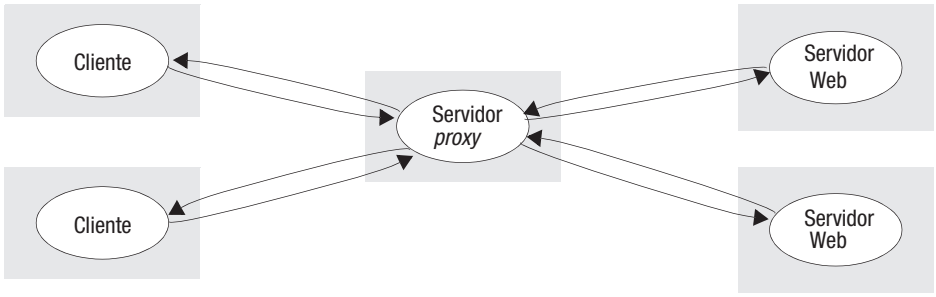

Figura 2.5 Servidor *proxy* Web.

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.

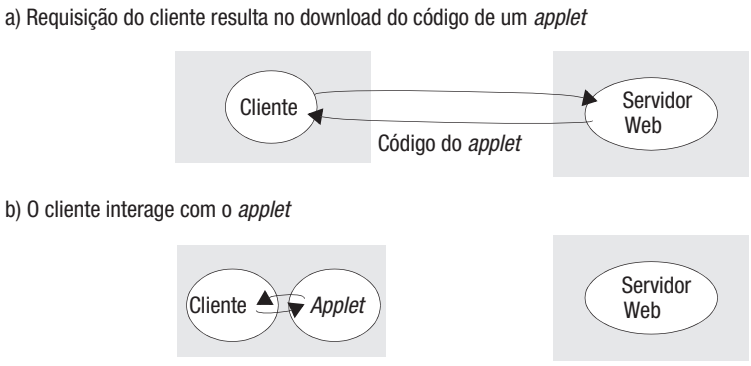

Figura 2.6 *Applets* Web.

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,

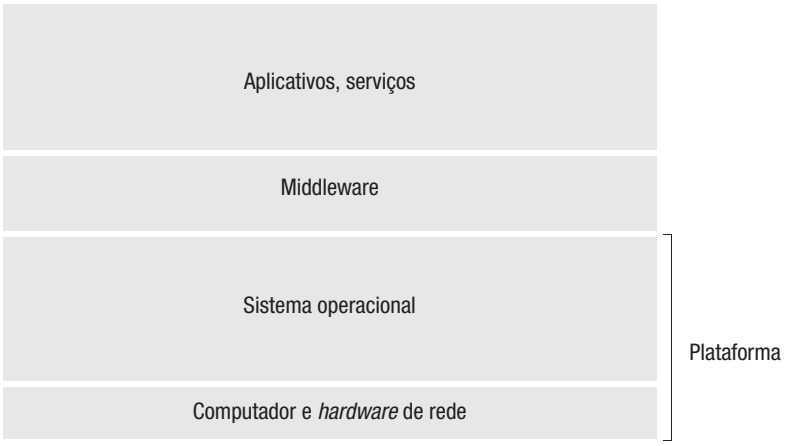

Figura 2.7 Camadas lógicas de serviço de *software* e *hardware* em sistemas distribuídos.

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,

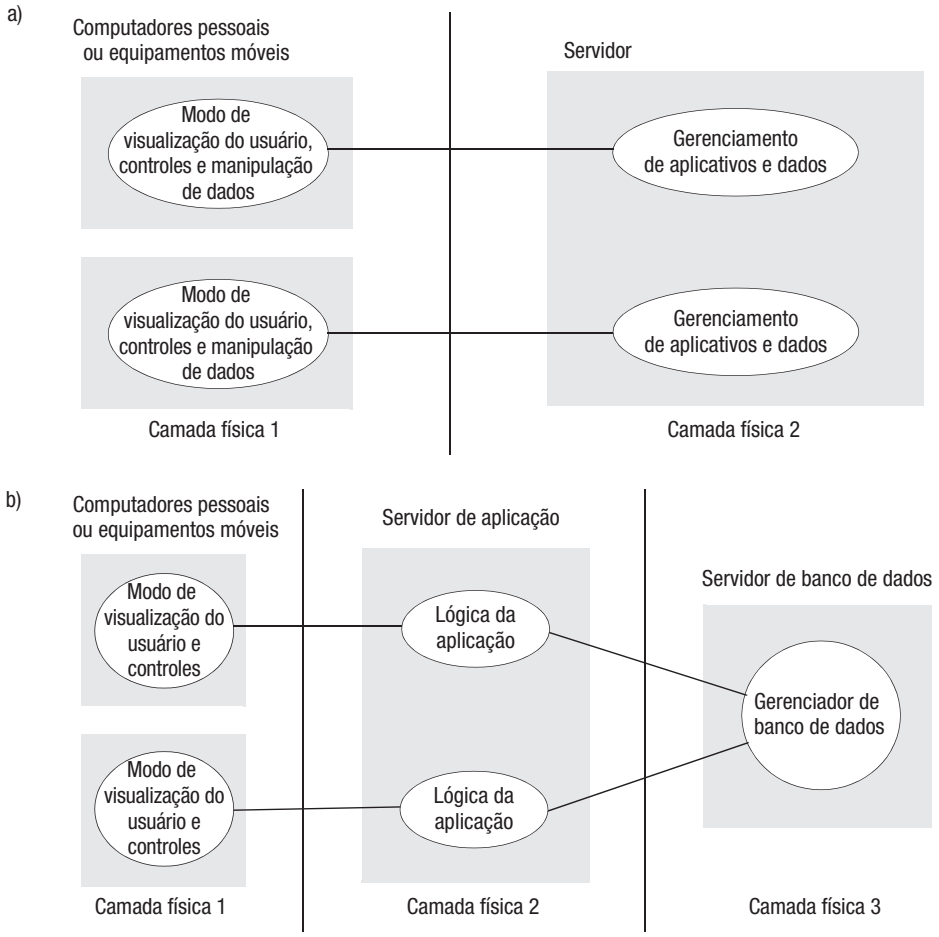

Figura 2.8 Arquitetura de duas e de três camadas físicas.

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-

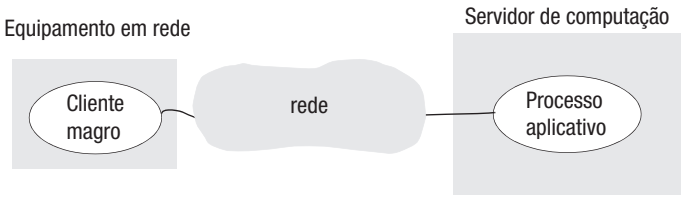

Figura 2.10 Clientes "magros" e servidores.

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].

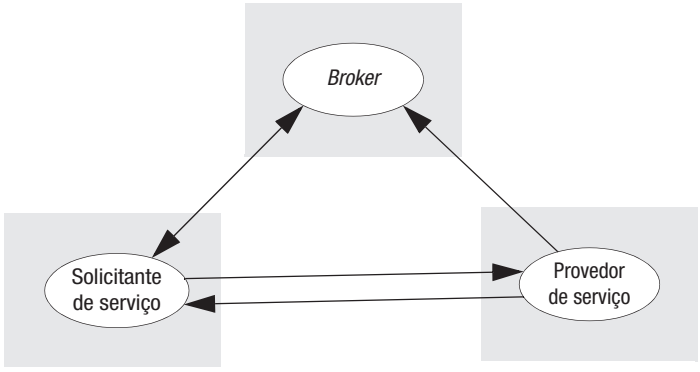

Figura 2.11 O padrão arquitetônico do serviço Web.

* 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.

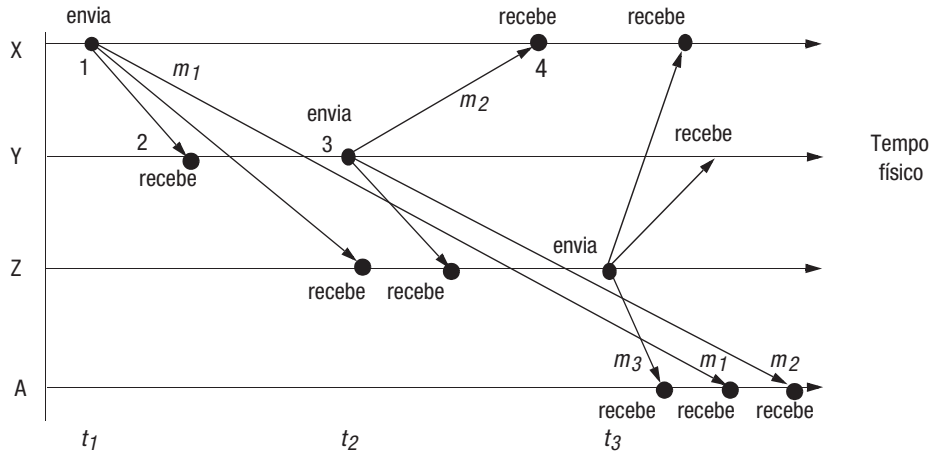

Figura 2.13 Ordenação de eventos no tempo físico.

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

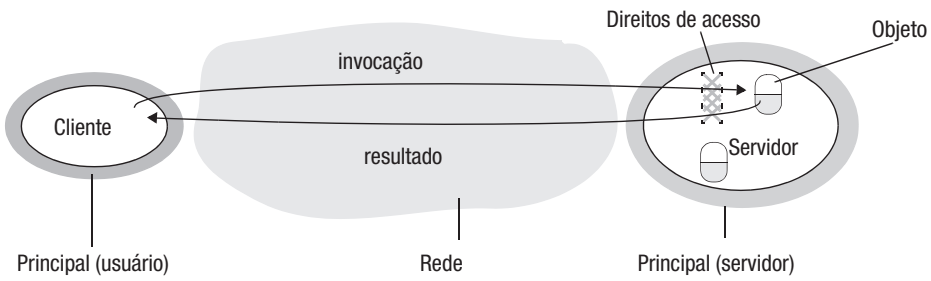

Figura 2.17 Objetos e principais.

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*.

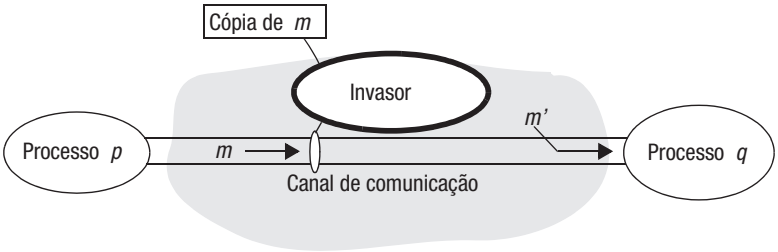

Figura 2.18 O invasor (atacante).

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:

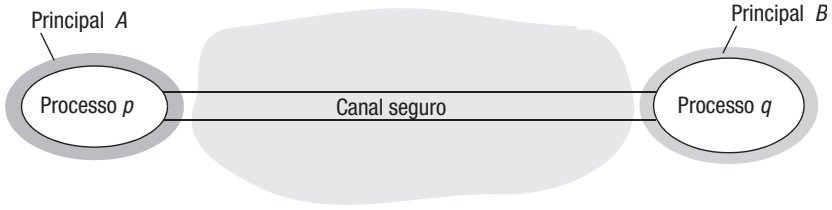

Figura 2.19 Canais seguros.

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*