4

Comunicação Entre Processos

Este capítulo considera as características dos protocolos para comunicação entre processos em um sistema distribuído, isto é, comunicação entre processos.

A comunicação entre processos na Internet fornece tanto comunicação por datagrama como por fluxo (*stream*). As APIs para esses tipos de comunicação são apresentadas em Java, junto a uma discussão sobre seus modelos de falha. Elas fornecem blocos de construção alternativos para os protocolos de comunicação. Isso é complementado com um estudo dos protocolos para a representação de conjuntos de objetos de dados em mensagens e referências a objetos remotos. Juntos, esses serviços oferecem suporte para a construção de serviços de comunicação, conforme discutido nos dois próximos capítulos.

Todas as primitivas de comunicação entre processos discutidas anteriormente suportam comunicação ponto a ponto, sendo igualmente úteis para o envio de uma mensagem de um remetente para um grupo de destinatários. O capítulo também considera a comunicação por difusão seletiva (*multicast*), incluindo *multicast* IP e os conceitos fundamentais de confiabilidade e ordenamento de mensagens.

O *multicast* é um requisito importante para os aplicativos distribuídos e deve ser fornecido mesmo que o suporte subjacente para *multicast* IP não esteja disponível. Normalmente, isso é fornecido por uma rede de sobreposição construída sobre a rede TCP/IP subjacente. As redes de sobreposição também podem fornecer suporte para compartilhamento de arquivos, maior confiabilidade e distribuição de conteúdo.

O MPI (Message Passing Interface) é um padrão desenvolvido para fornecer uma API para um conjunto de operações de passagem de mensagens, com variantes síncronas e assíncronas.

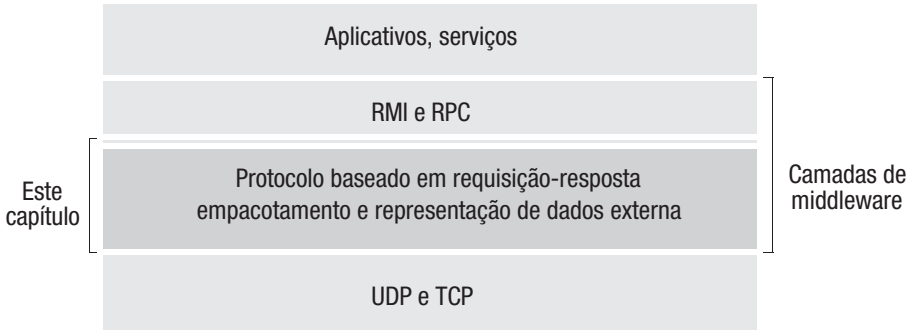

Figura 4.1 Camadas de *middleware*.

4.1 Introdução

Este capítulo e os dois seguintes estão relacionados aos aspectos da comunicação do *middleware*, embora os princípios sejam aplicados mais amplamente. Aqui, tratamos do projeto dos componentes mostrados na camada mais escura da Figura 4.1. A camada acima será discutida no Capítulo 5, que examina a invocação remota, e no Capítulo 6, que considera os paradigmas de comunicação indireta.

O Capítulo 3 discutiu os protocolos em nível de transporte UDP e TCP da Internet, sem dizer como *middlewares* e aplicativos poderiam utilizar esses protocolos. A próxima seção deste capítulo apresentará as características da comunicação entre processos e discutirá os protocolos UDP e TCP do ponto de vista do programador, apresentando a interface Java para cada um deles, junto com uma discussão sobre seus modelos de falha.

A interface de programa aplicativo para UDP fornece uma abstração de *passagem de mensagem* – a forma mais simples de comunicação entre processos. Isso permite que um processo remetente transmita uma única mensagem para um processo destino. Os pacotes independentes contendo essas mensagens são chamados de *datagramas*. Nas APIs Java e UNIX, o remetente especifica o destino usando um soquete – uma referência indireta a uma porta em particular usada pelo processo de destino que executa em um computador.

A interface de programa aplicativo para TCP fornece a abstração de um *fluxo* (*stream*) bidirecional entre pares de processos. A informação transmitida consiste em um fluxo contínuo de dados sem dar a noção de limites da mensagem, isto é, a noção de que ela tem um início e um fim. Os fluxos fornecem um bloco de construção para a comunicação produtor–consumidor. Um produtor e um consumidor formam um par de processos no qual a função do primeiro é produzir itens de dados e do segundo é consumi-los. Os itens de dados enviados pelo produtor para o consumidor são enfileirados na chegada do *host* destino até que o consumidor esteja pronto para recebê-los. O consumidor deve esperar quando nenhum item de dados estiver disponível. O produtor deve esperar, caso o armazenamento usado para conter os itens de dados enfileirados esteja cheio.

A Seção 4.3 se preocupa com o modo como os objetos e as estruturas de dados usados nos programas aplicativos podem ser transformados em uma forma conveniente para envio de mensagens pela rede, levando em consideração o fato de que diferentes computadores podem utilizar diferentes representações para tipos simples de dados. A seção também discutirá uma representação conveniente para referências a objeto em um sistema distribuído.

A Seção 4.4 abordará a comunicação por *multicast*: uma forma de comunicação entre processos na qual um processo de um grupo transmite a mesma mensagem para todos os membros do grupo de processos. Após explicar o *multicast* IP, a seção discutirá a necessidade de formas de *multicast* mais confiáveis.

A Seção 4.5 examinará o assunto cada vez mais importante das redes de sobreposição. Uma rede de sobreposição é uma rede construída sobre outra para permitir aos aplicativos direcionar mensagens para destinos não especificados por um endereço IP. As redes de sobreposição podem melhorar as redes TCP/IP por fornecer serviços alternativos, mais especializados. Elas são importantes no suporte para comunicação por *multicast* e na comunicação *peer-to-peer*.

Por fim, a Seção 4.6 apresentará o estudo de caso de um importante mecanismo de passagem de mensagens, o MPI, desenvolvido pela comunidade de computação de alto desempenho.

4.2 A API para protocolos Internet

Nesta seção, discutiremos as características gerais da comunicação entre processos e depois veremos os protocolos Internet como um exemplo, explicando como os programadores podem utilizá-los por meio de mensagens UDP ou por fluxos TCP.

A Seção 4.2.1 revê as operações de comunicação *send* e *receive* apresentadas na Seção 2.3.2, junto a uma discussão sobre como elas são sincronizadas e como os destinos das mensagens são especificados em um sistema distribuído. A Seção 4.2.2 apresenta os *soquetes*, que são empregados na interface para programação de aplicativos baseados em UDP e TCP. A Seção 4.2.3 discute o UDP e sua API Java. A Seção 4.2.4 discute o TCP e sua API Java. As APIs Java são orientadas a objetos, mas são semelhantes àquelas projetadas originalmente no sistema operacional Berkeley BSD 4.x UNIX; um estudo de caso sobre este último está disponível no *site* do livro [www.cdk5.net/IPC] (em inglês). Os leitores que estiverem estudando os exemplos de programação desta seção devem consultar a documentação Java *online*, ou Flanagan [2002], para ver a especificação completa das classes discutidas, que estão no pacote *java.net*.

4.2.1 As características da comunicação entre processos

A passagem de mensagens entre um par de processos pode ser suportada por duas operações de comunicação de mensagem: *send* e *receive*, definidas em termos de destinos e de mensagens. Para que um processo se comunique com outro, um deles envia (*send*) uma mensagem (uma sequência de bytes) para um destino e o outro processo, no destino, recebe (*receive*) a mensagem. Essa atividade envolve a comunicação de dados do processo remetente para o processo destino e pode implicar na sincronização dos dois processos. A Seção 4.2.3 fornece as definições para as operações *send* e *receive* na API Java para os protocolos Internet, com mais um estudo de caso sobre passagem de mensagens (MPI) oferecido na Seção 4.6.

Comunicação síncrona e assíncrona • Uma fila é associada a cada destino de mensagem. Os processos origem fazem as mensagens serem adicionadas em filas remotas, e os processos destino removem mensagens de suas filas locais. A comunicação entre os processos origem e destino pode ser síncrona ou assíncrona. Na forma *síncrona* de comunicação, os processos origem e destino são sincronizados a cada mensagem. Nesse caso, *send* e *receive* são operações que causam *bloqueio*. Quando um *envio* (*send*) é feito,

o processo origem (ou *thread*) é bloqueado até que a *recepção* (*receive*) correspondente seja realizada. Quando uma *recepção* é executada, o processo (ou *thread*) é bloqueado enquanto a mensagem não chegar.

Na forma *assíncrona* de comunicação, o uso da operação *send* é *não bloqueante*, no sentido de que o processo origem pode prosseguir assim que a mensagem tenha sido copiada para um *buffer* local, e a transmissão da mensagem ocorre em paralelo com o processo origem. A operação *receive* pode ter variantes com e sem bloqueio. Na variante *não bloqueante*, o processo destino prossegue sua execução após ter realizado a operação *receive*, a qual fornece um *buffer* para ser preenchido em *background.* Nesse caso, o processo deve receber separadamente uma notificação de que seu *buffer* possui dados a serem lidos, isso pode ser feito baseado em *polling* ou em interrupção.

Em um ambiente de sistema como Java, que suporta múltiplas *threads* em um único processo, a *recepção* bloqueante não tem desvantagens, pois ela pode ser executada por uma *thread*, enquanto outras *threads* do processo permanecem ativas; e a simplicidade de sincronização das *threads* destinos com a mensagem recebida é uma vantagem significativa. A comunicação não bloqueante parece ser mais eficiente, mas ela envolve uma complexidade extra no processo destino: a necessidade de ler uma mensagem recebida fora de seu fluxo normal de execução. Por esses motivos, os sistemas atuais geralmente não fornecem a forma de *recepção* não bloqueante.

Destinos de mensagem • O Capítulo 3 explicou que, nos protocolos Internet, as mensagens são enviadas para destinos identificados pelo par (*endereço IP, porta local*). Uma porta local é um destino de mensagem dentro de um computador, especificado como um valor inteiro. Uma porta tem exatamente um destino (as portas de *multicast* são uma exceção, veja a Seção 4.5.1), mas pode ter vários remetentes. Os processos podem usar várias portas para receber mensagens. Qualquer processo que saiba o número de uma porta pode enviar uma mensagem para ela. Geralmente, os servidores divulgam seus números de porta para os clientes acessarem.

Se o cliente usa um endereço IP fixo para se referir a um serviço, então esse serviço sempre deve ser executado no mesmo computador para que seu endereço permaneça válido. Para proporcionar transparência de localização, isso pode ser evitado com o uso da seguinte estratégia:

- Os programas clientes se referem aos serviços pelo nome e usam um servidor de nomes ou de associação (*binder*), veja a Seção 5.4.2, para transformar seus nomes em localizações de servidor no momento da execução. Isso permite que os serviços sejam movidos enquanto o sistema está em execução.

Confiabilidade • O Capítulo 2 definiu a comunicação confiável em termos de validade e integridade. No que diz respeito à propriedade da validade, um serviço de mensagem ponto a ponto pode ser descrito como confiável se houver garantia de que as mensagens foram entregues, independentemente da quantidade de pacotes que possam ter sido eliminados ou perdidos. Em contraste, um serviço de mensagem ponto a ponto pode ser descrito como não confiável se não houver garantia de entrega das mensagens. Quanto à integridade, as mensagens devem chegar não corrompidas e sem duplicação.

Ordenamento • Algumas aplicações exigem que as mensagens sejam entregues na *ordem de emissão* – isto é, na ordem em que foram transmitidas pela origem. A entrega de mensagens fora da ordem da origem é considerada uma falha por tais aplicações.

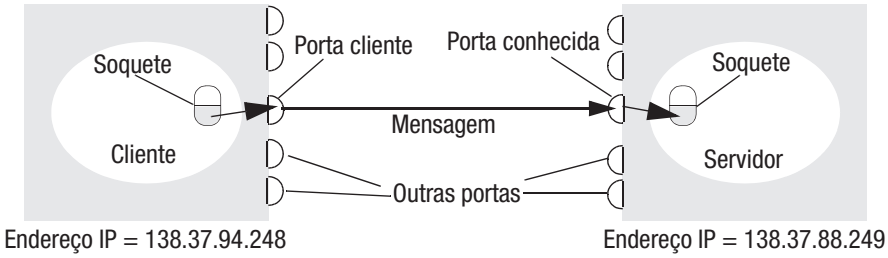

Figura 4.2 Soquetes e portas.

4.2.2 Soquetes

As duas formas de comunicação (UDP e TCP) usam a abstração de *soquete*, um ponto de destino para a comunicação entre processos. Os soquetes são originários do UNIX BSD, mas também estão presentes na maioria das versões do UNIX, incluindo o Linux, assim como no Windows e no Macintosh OS. A comunicação entre processos consiste em transmitir uma mensagem entre um soquete de um processo e um soquete de outro processo, conforme ilustrado na Figura 4.2. Para que um processo receba mensagens, seu soquete deve estar vinculado a uma porta local e a um dos endereços IP do computador em que é executado. As mensagens enviadas para um endereço IP e um número de porta em particular só podem ser recebidas por um processo cujo soquete esteja associado a esse endereço IP e a esse número de porta. Um processo pode usar o mesmo soquete para enviar e receber mensagens. Cada computador tem 2^{16} números de portas disponíveis para serem usados pelos processos para envio e recepção de mensagens. Qualquer processo pode fazer uso de várias portas para receber mensagens, mas um processo não pode compartilhar portas com outros processos no mesmo computador. (Os processos que usam *multicast* IP são uma exceção, pois compartilham portas – veja a Seção 4.4.1.) Entretanto, qualquer número de processos pode enviar mensagens para a mesma porta. Cada soquete é associado a um protocolo em particular – UDP ou TCP.

API Java para endereços Internet • A linguagem Java fornece uma classe, *InetAddress*, que representa endereços IP, para permitir a utilização dos protocolos TCP e UDP. Os usuários dessa classe se referem aos computadores pelos nomes de *host* DNS (Domain Name Service) (veja a Seção 3.4.7). As instâncias de *InetAddress* que contêm endereços IP podem ser criadas pela chamada ao método estático *InetAddress*, fornecendo-se um nome de *host* DNS como argumento. O método usa o DNS para obter o endereço IP correspondente. Por exemplo, para obter um objeto representando o endereço IP do *host* cujo nome DNS é *bruno.dcs.qmul.ac.uk*, use:

InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk")

Esse método pode disparar a exceção *UnknownHostException*. Note que o usuário da classe não precisa informar o valor explícito de um endereço IP. Na verdade, a classe encapsula os detalhes da representação dos endereços IP. Assim, a interface dessa classe não depende do número de bytes necessários para representar o endereço IP – 4 bytes no IPv4 e 16 bytes no IPv6.

4.2.3 Comunicação por datagrama UDP

Um datagrama enviado pelo protocolo UDP é transmitido de um processo origem para um processo destino sem a existência de confirmações ou novas tentativas de envio. Se ocorrer uma falha, a mensagem poderá não chegar. Um datagrama é transmitido entre processos quando um deles efetua um *send* e o outro, um *receive*. Para enviar ou receber mensagens, um processo precisa primeiro criar uma associação entre um soquete com um endereço IP e com uma porta do *host* local. Um servidor associa seu soquete a uma *porta de serviço* – que ele torna conhecida dos clientes para que estes possam enviar mensagens a ela. Um cliente vincula seu soquete a qualquer porta local livre. O método *receive* retorna, além da mensagem, o endereço IP e a porta da origem permitindo que o destinatário envie uma resposta a este.

Os parágrafos a seguir discutem alguns problemas relacionados à comunicação por datagrama:

Tamanho da mensagem: o processo destino precisa especificar um vetor de bytes de um tamanho em particular para receber as mensagens. Se a mensagem for grande demais para esse vetor, ela será truncada na chegada. O protocolo IP permite datagramas de até 2^{16} bytes (64 KB), incluindo seu cabeçalho e a área de dados. Entretanto, a maioria dos ambientes impõe uma restrição de tamanho de 8 kilobytes. Qualquer aplicativo que exija mensagens maiores do que o tamanho máximo deve fragmentá-las em porções desse tamanho. Geralmente, um aplicativo, por exemplo, o DNS, usará um tamanho que não seja excessivamente grande, mas adequado para o uso pretendido.

Bloqueio: normalmente, os soquetes fornecem operações *send* não bloqueantes e *receive* bloqueantes para comunicação por datagrama (um *receive* não bloqueante é uma opção possível em algumas implementações). A operação *send* retorna quando tiver repassado a mensagem para as camadas UDP e IP subjacentes, que são responsáveis por transmiti-la para seu destino. Ao chegar, a mensagem é posta em uma fila de recepção vinculada ao soquete associado à porta de destino. A mensagem é recuperada dessa fila quando uma operação *receive* for realizada, ou estiver com sua execução pendente, nesse soquete. Se nenhum processo tiver um soquete associado à porta de destino, as mensagens são descartadas.

O método *receive* bloqueia a execução do processo até que um datagrama seja recebido, a não ser que um tempo de espera limite tenha sido fornecido ao soquete. Se o processo que invoca o método *receive* tiver outra tarefa para fazer enquanto espera pela mensagem, ele deve tomar providências para usar *threads* separadas. As *threads* serão discutidas no Capítulo 7. Por exemplo, quando um servidor recebe uma mensagem de um cliente, normalmente uma tarefa é realizada; se o servidor for implementado usando várias *threads*, elas podem executar essas tarefas enquanto outra *thread* espera pelas mensagens de novos clientes.

Timeouts: a *recepção* bloqueante é conveniente para uso por um servidor que esteja esperando para receber requisições de seus clientes. Contudo, em algumas situações, não é adequado que um processo espere indefinidamente para receber algo, pois o processo remetente pode ter falhado ou a mensagem esperada pode ter se perdido. Para atender a tais requisitos, limites temporais (*timeouts*) podem ser configurados nos soquetes. É difícil escolher um *timeout* apropriado, porém ele deve ser grande, em comparação com o tempo exigido para transmitir uma mensagem.

Recepção anônima: o método *receive* não especifica uma origem para as mensagens. A invocação ao método *receive* obtém uma mensagem endereçada para seu

soquete, independentemente da origem. O método *receive* retorna o endereço IP e a porta local do processo origem, permitindo que o destinatário verifique de onde ela veio. Entretanto, é possível associar um soquete de datagrama a uma porta remota e a um endereço IP em particular, no caso em que se deseje apenas enviar e receber mensagens desse endereço.

Modelo de falhas • O Capítulo 2 apresentou um modelo de falhas para canais de comunicação e definiu a comunicação confiável em termos de duas propriedades: integridade e validade. A propriedade da integridade exige que as mensagens não devam estar corrompidas nem estejam duplicadas. O uso de uma soma de verificação garante que haja uma probabilidade insignificante de que qualquer mensagem recebida esteja corrompida. Os datagramas UDP sofrem das seguintes falhas:

Falhas por omissão: Ocasionalmente, mensagens podem ser descartadas devido a erros de soma de verificação ou porque não há espaço disponível no *buffer*, na origem ou no destino. Para simplificar a discussão, consideraremos as falhas por omissão de envio e por omissão de recepção (veja a Figura 2.15) como falhas por omissão no canal de comunicação.

Ordenamento: às vezes, as mensagens podem ser entregues em uma ordem diferente da que foram emitidas.

Os aplicativos que usam datagramas UDP podem efetuar seus próprios controles para atingir a qualidade de comunicação confiável que suas finalidades exigem. Um serviço de entrega confiável pode ser construído a partir de outro que sofra de falhas por omissão, pelo uso de confirmações. A Seção 5.2 discutirá como protocolos requisição-resposta confiáveis para comunicação cliente-servidor podem ser construídos sobre UDP.

Emprego de UDP • Para algumas aplicações, é aceitável usar um serviço que esteja exposto a falhas por omissão ocasionais. Por exemplo, o Domain Name Service, que pesquisa nomes DNS na Internet, é implementado sobre UDP. O Voice Over IP (VoIP) também é executado sobre UDP. Às vezes, os datagramas UDP são uma escolha atraente, pois não sofrem as sobrecargas necessárias à entrega de mensagens garantida. Existem três fontes de sobrecarga principais:

- a necessidade de armazenar informações de estado na origem e no destino;
- a transmissão de mensagens extras;
- a latência do remetente.

Os motivos dessas sobrecargas serão discutidos na Seção 4.2.4.

API Java para datagramas UDP • A API Java fornece comunicação por datagrama por meio de duas classes: *DatagramPacket* e *DatagramSocket*.

DatagramPacket: esta classe fornece um construtor para uma instância composta por um vetor de bytes (mensagem), o comprimento da mensagem, o endereço IP e o número da porta local do soquete de destino, como segue:

Pacote de datagrama

Instâncias de *DatagramPacket* podem ser transmitidas entre processos quando um processo realiza uma operação *send* e outro, *receive*.

Essa classe fornece outro construtor para ser usado na recepção de mensagens. Seus argumentos especificam um vetor de bytes, para armazenamento

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[ ]){
      // args fornece o conteúdo da mensagem e o nome de host do servidor
      DatagramSocket aSocket = null;
      try {
            aSocket = new DatagramSocket( );
            byte [ ] m = args[0].getBytes( );
            InetAddress aHost = InetAddress.getByName(args[1]);
            int serverPort = 6789;
            DatagramPacket request =
                  new DatagramPacket(m, m.length( ), aHost, serverPort);
            aSocket.send(request);
            byte[ ] buffer = new byte[1000];
            DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
            aSocket.receive(reply);
            System.out.println("Reply: " + new String(reply.getData( )));
      } catch (SocketException e){System.out.println("Socket: " + e.getMessage( ));
      } catch (IOException e){System.out.println("IO: "+ e.getMessage( ));
      } finally { if(aSocket!= null) aSocket.close( );}
   }
}
```

Figura 4.3 O cliente UDP envia uma mensagem para o servidor e obtém uma resposta.

da mensagem a ser recebida, e seu comprimento. Uma mensagem recebida é colocada no *DatagramPacket*, junto a seu comprimento e o endereço IP e a porta do soquete origem. A mensagem pode ser recuperada do *DatagramPacket* por meio do método *getData*. Os métodos *getPort* e *getAddress* acessam a porta e o endereço IP.

DatagramSocket: esta classe oferece mecanismos para criação de soquetes para envio e recepção de datagramas UDP. Ela fornece um construtor que recebe como argumento um número de porta, para os processos que precisam utilizar uma porta em particular, e um construtor sem argumentos que permite a obtenção dinâmica de um número de porta. Esses construtores podem provocar uma exceção *SocketException*, caso a porta já esteja em uso ou se, em ambientes UNIX, for especificada uma porta reservada (um número abaixo de 1024).

A classe *DatagramSocket* inclui os seguintes métodos:

send e *receive:* esses métodos servem para transmitir datagramas entre dois soquetes. O argumento de *send* é uma instância de *DatagramPacket* contendo uma mensagem e seu destino. O argumento de *receive* é um *DatagramPacket* vazio para se receber a mensagem, seu comprimento e origem. Os métodos *send* e *receive* podem causar exceções *IOException*.

setSoTimeout: este método permite o estabelecimento de um *timeout*. Com um *timeout* configurado, o método *receive* bloqueará pelo tempo especificado e, depois, causará uma exceção *InterruptedIOException*.

```
import java.net.*;
import java.io.*;
public class UDPServer{
public static void main(String args[ ]){
      DatagramSocket aSocket = null;
      try{
            aSocket = new DatagramSocket(6789);
            byte[ ] buffer = new byte[1000];
           while(true){
             DatagramPacket request = new DatagramPacket(buffer, buffer.length);
               aSocket.receive(request);
               DatagramPacket reply = new DatagramPacket(request.getData( ),
                    request.getLength( ), request.getAddress( ), request.getPort( ));
            aSocket.send(reply);
            }
      } catch (SocketException e){System.out.println("Socket: " + e.getMessage( ));
      } catch (IOException e) {System.out.println("IO: " + e.getMessage( ));
      } finally {if (aSocket!= null) aSocket.close( );}
      }
}
```

Figura 4.4 O servidor UDP recebe uma mensagem e a envia de volta para o cliente.

connect: este método é usado para contactar uma porta remota e um endereço IP em particular, no caso em que o soquete é capaz apenas de enviar e receber mensagens desse endereço.

A Figura 4.3 mostra o programa de um cliente que cria um soquete, envia uma mensagem para um servidor na porta 6789 e depois espera para receber uma resposta. Os argumentos do método *main* fornecem uma mensagem e o nome DNS do servidor. A mensagem é convertida em um vetor de bytes e o nome DNS é convertido em um endereço IP. A Figura 4.4 mostra o programa do servidor correspondente, o qual cria um soquete vinculado à porta de serviço (6789) e depois, em um laço, espera pelo recebimento de uma mensagem e responde, enviando-a de volta.

4.2.4 Comunicação por fluxo TCP

A API do protocolo TCP, que se originou do UNIX BSD 4.x, fornece a abstração de um fluxo de bytes no qual dados podem ser lidos (*receive*) e escritos (*send*). As seguintes características da rede são ocultas pela abstração de fluxo (*stream*):

Tamanho das mensagens: o aplicativo pode escolher o volume de dados que vai ser enviado ou recebido em um fluxo. Ele pode lidar com conjuntos de dados muito pequenos ou muito grandes. A implementação da camada TCP decide o volume de dados a coletar, antes de transmiti-los efetivamente como um ou mais datagramas IP. Ao chegar, os dados são entregues ao aplicativo, conforme solicitado. Se necessário, os aplicativos podem obrigar os dados a serem enviados imediatamente.

Mensagens perdidas: o protocolo TCP usa um esquema de confirmação. Como um simples exemplo desses esquemas (não é o usado no TCP), o lado remetente mantém um registro de cada datagrama IP enviado, e o lado destino confirma todas as

chegadas. Se o remetente não receber uma confirmação dentro de um tempo limite, ele retransmite a mensagem. O esquema de janela deslizante [Comer 2006], mais sofisticado, reduz o número de mensagens de confirmação exigidas.

Controle de fluxo: o protocolo TCP tenta combinar a velocidade dos processos que leem e escrevem em um fluxo. Se o processo que escreve (envia) for rápido demais para o que lê (recebe), então ele será bloqueado até que o leitor tenha consumido dados suficientes.

Duplicação e ordenamento de mensagens: identificadores de mensagem são associados a cada datagrama IP, o que permite ao destinatário detectar e rejeitar duplicatas ou reordenar as mensagens que chegam fora da ordem de emissão.

Destinos de mensagem: dois processos que estão em comunicação estabelecem uma conexão antes de poderem se comunicar por meio de um fluxo. Uma vez estabelecida a conexão, os processos simplesmente leem ou escrevem no fluxo, sem necessidade de usar endereços IP e portas. O estabelecimento de uma conexão envolve uma requisição de *connect,* do cliente para o servidor, seguido de uma requisição de *accept,* do servidor para o cliente, antes que qualquer comunicação possa ocorrer. Em um modelo cliente-servidor, isso causa uma sobrecarga considerável para cada requisição-resposta.

A API para comunicação por fluxo pressupõe que, quando dois processos estão estabelecendo uma conexão, um deles desempenha o papel de cliente e o outro desempenha o papel de servidor, mas daí em diante eles poderiam ser iguais. O papel de cliente envolve a criação de um soquete de fluxo vinculado a qualquer porta e, depois, um pedido *connect* solicitando uma conexão a um servidor, em uma determinada porta. O papel de servidor envolve a criação de um soquete de "escuta" (*listen*), vinculado à porta de serviço, para esperar que os clientes solicitem conexões. O soquete de "escuta" mantém uma fila de pedidos de conexão recebidos. Na abstração de soquete, quando o servidor aceita uma conexão, um novo soquete de fluxo é criado para que o servidor se comunique com um cliente, mantendo nesse meio-tempo seu soquete de "escuta" na porta de serviço para receber os pedidos *connect* de outros clientes.

O par de soquetes no cliente e no servidor são, na realidade, conectados por dois fluxos, um em cada direção. Assim, cada soquete tem um fluxo de entrada e um fluxo de saída. Um dos dois processos pode enviar informações para o outro, escrevendo em seu fluxo de saída, e o outro processo obtém as informações lendo seu fluxo de entrada.

Quando um aplicativo *encerra* (operação *close*) um soquete, isso indica que ele não escreverá mais nenhum dado em seu fluxo de saída. Os dados de seu *buffer* de saída são enviados para o outro lado do fluxo e colocados na fila de entrada do soquete de destino com uma indicação de que o fluxo está desfeito. O processo no destino pode ler os dados da fila, mas todas as outras leituras depois que a fila estiver vazia resultarão em uma indicação de fim de fluxo. Quando um processo termina, ou falha, todos os seus soquetes são encerrados e qualquer processo que tente se comunicar com ele descobrirá que sua conexão foi desfeita.

Os parágrafos a seguir tratam de alguns problemas importantes relacionados à comunicação por fluxo:

Correspondência de itens de dados: dois processos que estejam se comunicando precisam concordar quanto ao conteúdo dos dados transmitidos por um fluxo. Por exemplo, se um processo escreve (envia) um valor *int* em um fluxo, seguido de um valor *double*, então o outro lado deverá ler um valor *int*, seguido de um valor

double. Quando dois processos não cooperam corretamente no uso de um fluxo, o processo leitor pode causar erros ao interpretar os dados, ou ser bloqueado devido a dados insuficientes no fluxo.

Bloqueio: os dados gravados em um fluxo são mantidos em uma fila no soquete de destino. Quando um processo tentar ler dados de um canal de entrada, obterá dados da fila ou será bloqueado até que dados se tornem disponíveis. O processo que escreve dados em um fluxo pode ser bloqueado pelo mecanismo de controle de fluxo TCP, caso o soquete no outro lado já esteja armazenando o volume máximo de dados permitido pelo protocolo.

Threads: quando um servidor aceita uma conexão, ele geralmente cria uma nova *thread* para se comunicar com o novo cliente. A vantagem de usar uma *thread* separada para cada cliente é que o servidor pode bloquear quando estiver esperando por dados, sem atrasar os outros clientes. Em um ambiente em que *threads* não são suportadas, uma alternativa é testar, antes de tentar lê-lo, se a entrada está disponível; por exemplo, em um ambiente UNIX, a chamada de sistema *select* pode ser usada para esse propósito.

Modelo de falhas • Para satisfazer a propriedade da integridade da comunicação confiável, os fluxos TCP usam somas de verificação para detectar e rejeitar pacotes corrompidos, assim como números de sequência para detectar e rejeitar pacotes duplicados. Quanto à propriedade da validade, os fluxos TCP usam *timeout* e retransmissões para lidar com pacotes perdidos. Portanto, há garantia de que as mensagens sejam entregues, mesmo quando alguns dos pacotes das camadas inferiores são perdidos.

No entanto, se a perda de pacotes em uma conexão ultrapassar um limite, ou se a rede que está conectando dois processos for rompida ou se tornar seriamente congestionada, o *software* TCP responsável pelo envio de mensagens não receberá nenhum tipo de confirmação e, após certo tempo, declarará que a conexão está desfeita. Assim, o protocolo TCP não fornece comunicação confiável, pois não garante a entrega de mensagens diante de todas as dificuldades possíveis.

Quando uma conexão é desfeita, um processo, ao tentar ler ou escrever algo nela, receberá uma notificação de erro. Isso tem os seguintes efeitos:

- os processos que estão usando a conexão não poderão distinguir entre falha de rede e falha do processo no outro lado da conexão;
- os processos que estão se comunicando não poderão identificar se as mensagens que enviaram recentemente foram recebidas ou não.

Emprego de TCP • Muitos serviços frequentemente usados são executados em conexões TCP com números de porta reservados. Eles incluem os seguintes:

HTTP: o protocolo de transferência de hipertexto é usado para comunicação entre navegadores e servidores Web; ele será discutido na Seção 5.2.

FTP: o protocolo de transferência de arquivos permite a navegação em diretórios em um computador remoto e que arquivos sejam transferidos de um computador para outro por meio de uma conexão.

Telnet: o serviço telnet dá acesso a um computador remoto por meio de uma sessão de terminal.

SMTP: o protocolo de transferência de correio eletrônico é usado para enviar correspondência entre computadores.

```
import java.net.*;
import java.io.*;
public class TCPClient {
      public static void main (String args[ ]) {
      // os argumentos fornecem a mensagem e o nome de host de destino
        Socket s = null;
         try{
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream(s.getInputStream( ));
            DataOutputStream out =
                  new DataOutputStream( s.getOutputStream( ));
            out.writeUTF(args[0]); // UTF é uma codificação de string; veja a Seção 4.3
             String data = in readUTF( );
            System.out.println("Received: "+ data);
           }catch (UnknownHostException e){
            System.out.println("Sock:"+e.getMessage( ));
        } catch (EOFException e){System.out.println("EOF:"+e.getMessage( ));
        } catch (IOException e){System.out.println("IO:"+e.getMessage( ));
      } finally {if(s!=null) try {s.close( );}catch (IOException e){/*close falhou*/}}
      }
}
```

Figura 4.5 O cliente TCP estabelece uma conexão com o servidor, envia uma requisição e recebe uma resposta.

API Java para fluxos TCP • A interface Java para fluxos TCP é fornecida pelas classes *ServerSocket* e *Socket*.

ServerSocket: esta classe se destina a ser usada por um servidor para criar um soquete em uma porta de serviço para receber requisições de *connect* dos clientes. Seu método *accept* recupera um pedido *connect* da fila ou, se a fila estiver vazia, bloqueia até que chegue um. O resultado da execução de *accept* é uma instância de *Socket* – um soquete para dar acesso aos fluxos para comunicação com o cliente.

Socket: esta classe é usada pelos dois processos de uma conexão. O cliente usa um construtor para criar um soquete, especificando o nome DNS do *host* e a porta do servidor. Esse construtor não apenas cria um soquete associado a uma porta local, mas também o conecta com o computador remoto e com o número de porta especificado. Ele pode causar uma exceção *UnknownHostException*, caso o nome de *host* esteja errado, ou uma exceção *IOException*, caso ocorra um erro de E/S.

A classe *Socket* fornece os métodos *getInputStream* e *getOutputStream* para acessar os dois fluxos associados a um soquete. Os tipos de retorno desses métodos são *InputStream* e *OutputStream*, respectivamente – classes abstratas que definem métodos para ler e escrever os bytes. Os valores de retorno podem ser usados como argumentos de construtores para fluxos de entrada e saída. Nosso exemplo usa *DataInputStream* e *DataOutputStream*, que permitem que representações binárias de tipos de dados primitivos sejam lidas e escritas de forma independente de máquina.

```
import java.net.*;
import java.io.*;
public class TCPServer {
      public static void main (String args[ ]) {
         try{
            int serverPort = 7896;
            ServerSocket listenSocket = new ServerSocket(serverPort);
            while(true) {
               Socket clientSocket = listen Socket.accept( );
               Connection c = new Connection(clientSocket);
            }
         } catch(IOException e) {System.out.println("Listen :"+e.getMessage( ));}
      }
 }
class Connection extends Thread {
      DataInputStream in;
      DataOutputStream out;
      Socket clientSocket;
      public Connection (Socket aClientSocket) {
         try {
            clientSocket = aClientSocket;
            in = new DataInputStream(clientSocket.getInputStream( ));
            out =new DataOutputStream(clientSocket.getOutputStream( ));
            this.start( );
         } catch(IOException e) {System.out.println("Connection:"+e.getMessage( ));}
      }
      public void run( ){
         try {              // an echo server
            String data = in.readUTF( );
            out.writeUTF(data);
         } catch(EOFException e) {System.out.println("EOF:"+e.getMessage( ));
         } catch(IOException e) {System.out.println("IO:"+e.getMessage( ));
         } finally { try {clientSocket.close( );}catch (IOException e){/*close falhou*/ }}
      }
}
```

Figura 4.6 O servidor TCP estabelece uma conexão para cada cliente e, em seguida, ecoa o pedido do cliente.

A Figura 4.5 mostra um programa cliente no qual os argumentos do método *main* fornecem uma mensagem e o nome DNS do servidor. O cliente cria um soquete vinculado ao nome DNS do servidor e à porta de serviço 7896. Ele produz um *DataInputStream* e um *DataOutputStream* a partir dos fluxos de entrada e saída do soquete e, em seguida, escreve a mensagem em seu fluxo de saída e espera para ler uma resposta em seu fluxo de entrada. O programa servidor da Figura 4.6 abre um soquete em sua porta de "escuta", ou *listen*, (7896) e recebe os pedidos *connect*. A cada pedido que chega, uma nova *thread* é criada para se comunicar com o cliente. A *thread* cria um *DataInputStream* e um *DataOutputStream* a partir dos fluxos de entrada e saída de seu soquete e, em seguida, espera para ler uma mensagem e enviá-la de volta.

Como nossa mensagem consiste em uma cadeia de caracteres (*string*), os processos cliente e servidor usam o método *writeUTF* de *DataOutputStream* para escrevê-la no fluxo de saída e o método *readUTF* de *DataInputStream* para lê-la do fluxo de entrada. UTF-8 é uma codificação que representa *strings* em um formato específico, que será descrito na Seção 4.3.

Quando um processo tiver encerrado (*close*) seu soquete, não poderá mais usar seus fluxos de entrada e saída. O processo para o qual ele tiver enviado dados ainda poderá lê-los em sua fila, mas as leituras feitas após essa fila ficar vazia resultarão em uma exceção *EOFException*. As tentativas de usar um soquete já encerrado ou de escrever em um fluxo desfeito resultarão em uma exceção *IOException*.

4.3 Representação externa de dados e empacotamento

As informações armazenadas nos programas em execução são representadas como estruturas de dados – por exemplo, pela associação de um conjunto de objetos –, enquanto que as informações presentes nas mensagens são sequências puras de bytes. Independente da forma de comunicação usada, as estruturas de dados devem ser simplificadas (convertidas em uma sequência de bytes) antes da transmissão e reconstruídas na sua chegada. Os dados transmitidos nas mensagens podem corresponder a valores de tipos de dados primitivos diferentes, e nem todos os computadores armazenam tipos de dados primitivos, como os inteiros, na mesma ordem. A representação interna de números em ponto flutuante também difere entre as arquiteturas de processadores. Existem duas variantes para a ordenação de inteiros: ordem *big-endian*, na qual o byte mais significativo aparece na primeira posição, e a ordem *little-endian*, na qual ele aparece por último. Outro problema é o conjunto de códigos usado para representar caracteres: por exemplo, a maioria dos aplicativos em sistemas como o UNIX usa codificação de caracteres ASCII, com um byte por caractere, enquanto o padrão Unicode permite a representação de textos em muitos idiomas diferentes e usa dois bytes por caractere.

Um dos métodos a seguir pode ser usado para permitir que dois computadores troquem valores de dados binários:

- Os valores são convertidos para um formato externo, acordado antes da transmissão e convertidos para a forma local, na recepção; se for sabido que os dois computadores são do mesmo tipo, a conversão para o formato externo pode ser omitida.
- Os valores são transmitidos no formato do remetente, junto a uma indicação do formato usado, e o destinatário converte os valores, se necessário.

Note, entretanto, que os bytes em si nunca têm a ordem de seus bits alterada durante a transmissão. Para suportar RMI ou RPC, todo tipo de dados que possa ser passado como argumento, ou retornado como resultado, deve ser simplificado, e os valores de dados primitivos individuais, representados em um formato comum. Um padrão aceito para a representação de estruturas de dados e valores primitivos é chamado de *representação externa de dados*.

Empacotamento (*marshalling*) é o procedimento de pegar um conjunto de itens de dados e montá-los em uma forma conveniente para transmissão em uma mensagem. *Desempacotamento* (*unmarshalling*) é o procedimento inverso de desmontá-los na chegada para produzir um conjunto de itens de dados equivalente no destino. Assim, o empacotamento consiste na transformação de itens de dados estruturados e valores primitivos em uma representação externa de dados. Analogamente, o desempacotamento consiste na

geração de valores primitivos a partir de sua representação externa de dados e na reconstrução das estruturas de dados.

Serão discutidas três estratégias alternativas para representação externa de dados e empacotamento (com uma quarta considerada no Capítulo 21, quando examinarmos a estratégia do Google para a representação de dados estruturados):

- A representação comum de dados do CORBA, que está relacionada a uma representação externa dos tipos estruturados e primitivos que podem ser passados como argumentos e resultados na invocação a métodos remotos no CORBA. Ela pode ser usada por diversas linguagens de programação (veja o Capítulo 8).
- A serialização de objetos da linguagem Java, que está relacionada à simplificação e à representação externa de dados de um objeto, ou de uma árvore de objetos, que precise ser transmitida em uma mensagem ou armazenada em um disco. Isso é usado apenas pela linguagem Java.
- A XML ou Extensible Markup Language, que define um formato textual para representar dados estruturados. Ela se destinava, originalmente, a documentos contendo dados estruturados textuais autodescritivos; por exemplo, documentos Web. Porém, agora também é usada para representar dados enviados em mensagens trocadas por clientes e servidores em serviços Web (veja o Capítulo 9).

Nos dois primeiros casos, as atividades de empacotamento e desempacotamento se destinam a serem executadas por uma camada de *middleware*, sem nenhum envolvimento por parte do programador de aplicativo. Mesmo no caso da XML, que é textual e, portanto, mais acessível para tratar de codificação, o *software* para empacotar e desempacotar está disponível para praticamente todas as plataformas e ambientes de programação comumente usados. Como o empacotamento exige a consideração de todos os mínimos detalhes da representação dos componentes primitivos de objetos compostos, esse procedimento é bastante propenso a erros se executado manualmente. A compactação é outro problema que pode ser tratado no projeto de procedimentos de empacotamento gerados automaticamente.

Nas duas primeiras estratégias, os tipos de dados primitivos são empacotados em uma forma binária. Na terceira estratégia (XML), os tipos de dados primitivos são representados textualmente. A representação textual de um valor de dados geralmente será maior do que a representação binária equivalente. O protocolo HTTP, que será descrito no Capítulo 5, é outro exemplo de estratégia textual.

Outro problema com relação ao projeto de métodos de empacotamento é se os dados empacotados devem incluir informações relativas ao tipo de seu conteúdo. Por exemplo, a representação usada pelo CORBA inclui apenas os valores dos objetos transmitidos – nada a respeito de seus tipos. Por outro lado, tanto a serialização Java, como a XML, incluem informações sobre o tipo, mas de maneiras diferentes. A linguagem Java coloca todas as informações de tipo exigidas na forma serializada, mas os documentos XML podem se referir a conjuntos de nomes (com tipos) definidos externamente, chamados *espaços de nomes*.

Embora estejamos interessados no uso de representação externa de dados para os argumentos e resultados de RMIs e de RPCs, ela tem um uso mais genérico quando é empregada para representar estruturas de dados, objetos ou documentos estruturados em uma forma conveniente para transmissão em mensagens ou para armazenamento em arquivos.

Duas outras técnicas para a representação de dados externos são dignas de nota. O Google usa uma estratégia chamada de *buffers de protocolo* para capturar a representação de dados armazenados e de dados transmitidos. Essa estratégia vai ser examinada

Tipo	*Representação*
sequence	comprimento (*unsigned long*) seguido de seus elementos, em ordem
string	comprimento (*unsigned long*) seguido pelos caracteres que o compõem (um caractere pode ocupar mais de um *byte*)
array	elementos de vetor, fornecidos em ordem (nenhum comprimento especificado, pois é fixo)
struct	na ordem da declaração dos componentes
enumerated	*unsigned long* (os valores são especificados pela ordem declarada)
union	identificador de tipo seguido do membro selecionado

Figura 4.7 CDR do CORBA para tipos construídos.

na Seção 20.4.1. Também há um interesse considerável em JSON (JavaScript Object Notation) como uma estratégia de representação de dados externos [www.json.org]. Considerados em conjunto, os *buffers* de protocolo e JSON representam um passo na direção de estratégias mais leves para a representação de dados (quando comparadas, por exemplo, à XML).

4.3.1 Representação comum de dados (CDR) do CORBA

O CDR do CORBA é a representação externa de dados definida no CORBA 2.0 [OMG 2004a]. O CDR pode representar todos os tipos de dados que são usados como argumentos e valores de retorno em invocações a métodos remotos no CORBA. Eles consistem em 15 tipos primitivos, os quais incluem *short* (16 bits), *long* (32 bits), *unsigned short*, *unsigned long*, *float* (32 bits), *double* (64 bits), *char*, *boolean* (TRUE, FALSE), *octet* (8 bits) e *any* (que pode representar qualquer tipo primitivo ou construído), junto a uma variedade de tipos compostos. Eles estão descritos na Figura 4.7. Cada argumento ou resultado em uma invocação remota é representado por uma sequência de bytes na mensagem de invocação ou resultado.

Tipos primitivos: o CDR define uma representação para as ordens *big-endian* e *little-endian*. Os valores são transmitidos na ordem do remetente, que é especificada em cada mensagem. Se exigir uma ordem diferente, o destinatário a transforma. Por exemplo, um valor *short* de 16 bits ocupa dois bytes na mensagem e, para a ordem *big-endian*, os bits mais significativos ocupam o primeiro byte e os bits menos significativos ocupam o segundo byte. Cada valor primitivo é colocado em um índice na sequência de bytes, de acordo com seu tamanho. Suponha que a sequência de bytes seja indexada a partir de zero. Então, um valor primitivo com tamanho de n bytes (onde $n = 1, 2, 4$ ou 8) é anexado à sequência, em um índice que é um múltiplo de n no fluxo de bytes. Os valores em ponto flutuante seguem o padrão IEEE – no qual o sinal, o expoente e a parte fracionária estão nos bytes 0–n para a ordem *big-endian* e ao contrário na ordem *little-endian*. Os caracteres são representados por um código acordado entre cliente e servidor.

Tipos construídos ou compostos: os valores primitivos que compreendem cada tipo construído são adicionados a uma sequência de bytes, em uma ordem específica, como se vê na Figura 4.7.

Índice na sequência de bytes	← 4 *bytes* →	*Observações sobre a representação*
0–3	5	*Comprimento do string*
4–7	"Smit"	*'Smith'*
8–11	"h___"	
12–15	6	*Comprimento do string*
16–19	"Lond"	*'London'*
20–23	"on__"	
24–27	1984	*unsigned long*

Figura 4.8 Mensagem no CDR do CORBA.

A Figura 4.8 mostra uma mensagem no CDR do CORBA contendo três campos de um *struct* cujos tipos respectivos são *string*, *string* e *unsigned long*. A figura mostra a sequência de bytes, com quatro bytes em cada linha. A representação de cada *string* consiste em um valor *unsigned long*, dando seu comprimento, seguido dos caracteres do *string*. Por simplicidade, presumimos que cada caractere ocupa apenas um byte. Os dados de comprimento variável são preenchidos com zero para que tenha uma forma padrão para permitir a comparação de dados empacotados ou de sua soma de verificação. Note que cada valor *unsigned long*, que ocupa quatro bytes, começa em um índice que é múltiplo de quatro. A figura não distingue entre as ordens *big-endian* e *little-endian*. Embora o exemplo da Figura 4.8 seja simples, o CDR do CORBA pode representar qualquer estrutura de dados composta por tipos primitivos e construídos, mas sem usar ponteiros.

Outro exemplo de representação de dados externa é o padrão XDR da Sun, que está especificado na RFC 1832 [Srinivasan 1995b] e é descrito em www.cdk5.net/ipc (em inglês). Ele foi desenvolvido pela Sun para uso nas mensagens trocadas entre clientes e servidores NFS (veja o Capítulo 13).

O tipo de um item de dados não é fornecido com a representação de dados na mensagem, seja no CDR do CORBA ou no padrão XDR da Sun. Isso porque pressupõe-se que o remetente e o destinatário tenham conhecimento comum da ordem e dos tipos dos itens de dados de uma mensagem. Em particular para RMI, ou para RPC, cada invocação de método passa argumentos de tipos específicos e o resultado é um valor de um tipo em particular.

Empacotamento no CORBA • As operações de empacotamento (*marshalling*) podem ser geradas automaticamente a partir da especificação dos tipos dos itens de dados a serem transmitidos em uma mensagem. Os tipos das estruturas de dados e os tipos dos itens de dados básicos estão descritos no IDL (Interface Definition Language) do CORBA (veja a Seção 8.3.1), que fornece uma notação para descrever os tipos dos argumentos e resultados dos métodos RMI. Por exemplo, poderíamos usar o IDL do CORBA para descrever a estrutura de dados na mensagem da Figura 4.8 como segue:

```
struct Person{
        string name;
        string place;
        unsigned long year;
};
```

O compilador da interface CORBA (veja o Capítulo 5) gera as operações de empacotamento e desempacotamento apropriadas para os argumentos e resultados dos métodos remotos a partir das definições dos tipos de seus parâmetros e resultados.

4.3.2 Serialização de objetos Java

No Java RMI, tanto objetos como valores de dados primitivos podem ser passados como argumentos e resultados de invocações de método. Um objeto é uma instância de uma classe Java. Por exemplo, a classe Java equivalente a *struct Person* definida no IDL do CORBA poderia ser:

```
public class Person implements Serializable {
        private String name;
        private String place;
        private int year;
        public Person(String aName, String aPlace, int aYear) {
                name = aName;
                place = aPlace;
                year = aYear;
        }
        // seguido dos métodos para acessar as variáveis de instância
}
```

Essa classe diz que implementa a interface *Serializable*, a qual não tem métodos. Dizer que uma classe implementa a interface *Serializable* (que é fornecida no pacote *java.io*) tem o efeito de permitir que suas instâncias sejam serializadas.

Em Java, o termo *serialização* se refere à atividade de simplificar um objeto, ou um conjunto de objetos conectados, em uma forma sequencial conveniente para ser armazenada em disco ou transmitida em uma mensagem; por exemplo, como um argumento ou resultado de uma RMI. A *desserialização* consiste em restaurar o estado de um objeto ou conjunto de objetos a partir de sua forma serializada. Pressupõe-se que o processo que faz a desserialização não tenha nenhum conhecimento anterior dos tipos dos objetos na forma serializada. Portanto, qualquer informação sobre a classe de cada objeto é incluída na forma serializada. Essa informação permite que o destinatário carregue a classe apropriada quando um objeto é desserializado.

A informação sobre uma classe consiste em seu nome e em um número de versão. O número da versão deve mudar quando forem feitas alterações na classe. Ele pode ser estabelecido pelo programador ou calculado automaticamente como uma mistura do nome da classe, suas variáveis de instância, métodos e interfaces. O processo que desserializa um objeto pode verificar se ele tem a versão correta da classe.

Os objetos Java podem conter referências para outros objetos. Quando um objeto é serializado, todos os objetos que ele referencia são serializados junto, para garantir que, quando o objeto for reconstruído, todas as suas referências possam ser completadas no destino. As referências são serializadas por meio de *identificadores* (*handlers*) – neste caso, o *identificador* é uma referência a um objeto dentro da forma serializada; por exemplo, o próximo número em uma sequência de valores inteiros positivos. O procedimento de serialização deve garantir que exista uma correspondência biunívoca entre referências de objeto e seus identificadores. Ele também deve garantir que cada objeto seja gravado apenas uma vez – na segunda ocorrência de um objeto, ou em ocorrências subsequentes, é gravado o identificador, em vez de o objeto.

Valores serializados				*Explicação*
Person	Número da versão de 8 bytes		h0	*Nome da classe, número da versão*
3	int year	java.lang.String name	java.lang.String place	*Número, tipo e nome das variáveis de instância*
1984	5 Smith	6 London	h1	*Valores das variáveis de instância*

Na realidade, a forma serializada inclui marcas adicionais de tipos; h0 e h1 são identificadores.

Figura 4.9 Indicação da forma serializada Java.

Para serializar um objeto, a informação de sua classe é escrita por extenso, seguida dos tipos e nomes de suas variáveis de instância. Se as variáveis de instância pertencerem a novas classes, então suas informações de classe também deverão ser escritas por extenso, seguidas dos tipos e nomes de suas variáveis de instância. Esse procedimento recursivo continua até que a informação da classe e os tipos e nomes das variáveis de instância de todas as classes necessárias tenham sido escritas por extenso. Cada classe recebe um identificador (*handle*) e nenhuma classe é gravada mais do que uma vez no fluxo de bytes – os identificadores são gravados em seu lugar, onde for necessário.

O conteúdo das variáveis de instância que são tipos primitivos, como inteiros, caracteres, booleanos, bytes e longos, são gravados em um formato binário portável, usando métodos da classe *ObjectOutputStream*. Os *strings* e os caracteres são gravados pelo método *writeUTF*, usando o formato Universal Transfer Format (UTF-8), o qual permite que caracteres ASCII sejam representados de forma inalterada (em um byte), enquanto os caracteres Unicode são representados por vários bytes. Os *strings* são precedidos pelo número de bytes que ocupam no fluxo.

Como exemplo, considere a serialização do objeto a seguir:

Person p = new Person("Smith", "London", 1984);

A forma serializada está ilustrada na Figura 4.9, que omite os valores dos identificadores (*handlers*) e das informações de tipo que indicam objetos, classes, *strings* e outros recursos na forma serializada completa. A primeira variável de instância (1984) é um valor inteiro de comprimento fixo; a segunda e a terceira variáveis de instância são *strings* e são precedidas por seus comprimentos.

Para fazer uso da serialização Java, por exemplo, para serializar o objeto *Person*, é necessário criar uma instância da classe *ObjectOutputStream* e invocar seu método *writeObject*, passando o objeto *Person* como argumento. Para desserializar um objeto de um fluxo de dados, é necessário abrir o fluxo como *ObjectInputStream* e utilizar o método *readObject* para reconstruir o objeto original. O uso dessas duas classes é semelhante ao uso de *DataOutputStream* e *DataInputStream*, ilustrado nas Figuras 4.5 e 4.6.

A serialização e desserialização dos argumentos e resultados de invocações remotas geralmente são executadas automaticamente pela camada de *middeware*, sem nenhuma participação do programador do aplicativo. Se necessário, os programadores que tiverem requisitos especiais podem fazer sua própria versão dos métodos que leem e escrevem objetos. Para descobrir como fazer isso e para obter mais informações sobre serialização em Java, leia o exercício dirigido sobre serialização de objetos [java.sun.com II]. Outra maneira pela qual um programador pode modificar os efeitos da serialização é declarando as variáveis que não devem ser serializadas como *transientes*. Exemplos de variáveis que não devem ser serializadas são referências a recursos locais, como arquivos e soquetes.

O uso de reflexão • A linguagem Java suporta *reflexão* – a capacidade de fazer perguntas sobre as propriedades de uma classe, como os nomes e tipos de suas variáveis de instância e métodos. Isso também permite que classes sejam criadas a partir de seus nomes e que seja criado, para uma determinada classe, um construtor com argumentos de determinados tipos de dados. A reflexão torna possível fazer serialização e desserialização de maneira completamente genérica. Isso significa que não há necessidade de gerar funções de empacotamento especiais para cada tipo de objeto, conforme descrito anteriormente para CORBA. Para saber mais sobre reflexão, veja Flanagan [2002].

A serialização de objetos Java usa reflexão para descobrir o nome da classe do objeto a ser serializado e os nomes, tipos e valores de suas variáveis de instância. Isso é tudo que é necessário para a forma serializada.

Para a desserialização, o nome da classe na forma serializada é usado para criar uma classe. Isso é usado, então, para criar um novo construtor, com tipos de argumento correspondentes àqueles especificados na forma serializada. Finalmente, o novo construtor é usado para criar um novo objeto, com variáveis de instância cujos valores são lidos da forma serializada.

4.3.3 XML (Extensible Markup Language)

A XML é uma linguagem de marcação que foi definida pelo World Wide Web Consortium (W3C) para uso na Web. Em geral, o termo *linguagem de marcação* se refere a uma codificação textual que representa um texto e os detalhes de sua estrutura ou de sua aparência. Tanto a XML como a HTML foram derivadas da SGML (Standardized Generalized Markup Language) [ISO 8879], uma linguagem de marcação muito complexa. A HTML (veja a Seção 1.6) foi projetada para definir a aparência de páginas Web. A XML foi projetada para elaborar documentos estruturados para a Web.

Os itens de dados XML são rotulados com *strings* de marcação (*tags*). As *tags* são usadas para descrever a estrutura lógica dos dados e para associar pares atributo-valor às estruturas lógicas. Isto é, na XML, as *tags* estão relacionadas à estrutura do texto que englobam, em contraste com a HTML, na qual as *tags* especificam como um navegador poderia exibir o texto. Para ver uma especificação da XML, consulte as páginas sobre XML fornecidas pelo W3C, no endereço [www.w3.org VI].

A XML é usada para permitir que clientes se comuniquem com serviços Web e para definir as interfaces e outras propriedades desses mesmos serviços. Entretanto, a XML também é usada de muitas outras maneiras. Ela é utilizada no arquivamento e na recuperação de sistemas – embora um repositório de arquivos XML possa ser maior do que seu equivalente binário, ele tem a vantagem de poder ser lido em qualquer computador. Outros exemplos do uso da XML incluem a especificação de interfaces com o usuário e a codificação de arquivos de configuração em sistemas operacionais.

A XML é *extensível*, pois os usuários podem definir suas próprias *tags*, em contraste com a HTML, que usa um conjunto fixo de *tags*. Entretanto, se um documento XML se destina a ser usado por mais de um aplicativo, os nomes das *tags* devem ser combinados entre eles. Por exemplo, os clientes normalmente usam mensagens SOAP para se comunicar com serviços Web. O SOAP (veja a Seção 9.2.1) é um formato XML cujas *tags* são publicadas para serem usadas pelos serviços Web e seus clientes.

Algumas representações externas de dados (como o CDR do CORBA) não precisam ser autodescritivas, pois pressupõe-se que o cliente e o servidor que estejam trocando uma mensagem têm conhecimento anterior da ordem e dos tipos das informações que ela contém. Entretanto, a XML foi projetada para ser usada por vários aplicativos,

```
<person id="123456789">
        <name>Smith</name>
        <place>London</place>
        <year>1984</year>
        <!-- a comment -->
</person >
```

Figura 4.10 Definição em XML da estrutura *Person*.

para diferentes propósitos. A capacidade de prover *tags*, junto com o uso de espaços de nomes para definir o significado das próprias *tags*, tornou isso possível. Além disso, o uso de *tags* permite que os aplicativos selecionem apenas as partes de um documento que precisam processar, e isso não será afetado pela adição de informações que são relevantes para outros aplicativos.

Os documentos XML, sendo textuais, podem ser lidos por seres humanos. Na prática, a maioria dos documentos XML é gerada e lida por *software* de processamento de XML, mas a capacidade de ler código XML pode ser útil quando as coisas dão errado. Além disso, o uso de texto torna a XML independente de qualquer plataforma específica. O uso de uma representação textual, em vez de binária, junto com o uso de *tags*, torna as mensagens muito maiores, o que faz com que elas exijam tempos de processamento e transmissão maiores, assim como mais espaço de armazenamento. Uma comparação da eficiência das mensagens usando o formato XML SOAP e o CDR do CORBA é dada na Seção 9.2.4. Entretanto, os arquivos e as mensagens podem ser compactados – a HTTP versão 1.1 permite que os dados sejam compactados, o que economiza largura de banda durante a transmissão.

Elementos e atributos XML • A Figura 4.10 mostra a definição XML da estrutura *Person* que foi usada para ilustrar o empacotamento no CDR do CORBA e em Java. Ela mostra que a XML consiste em *tags* e dados do tipo caractere. Os dados do tipo caractere, por exemplo, *Smith* ou *1984*, são os dados reais. Assim como na HTML, a estrutura de um documento XML é definida por pares de *tags* incluídas entre sinais de menor e maior. Na Figura 4.10, <*name*> e <*place*> são *tags*. Assim como na HTML, o leiaute geralmente pode ser usado para melhorar a legibilidade. Na XML, os comentários são denotados da mesma maneira que na HTML.

Elementos: um elemento na XML consiste em um conjunto de dados do tipo caractere delimitados por *tags* de início e de fim correspondentes. Por exemplo, um dos elementos na Figura 4.10 consiste no dado *Smith*, contido dentro do par de *tags* <*name*>... </*name*>. Note que o elemento com a *tag* <*name*> é incluído no elemento com o par de *tags* <*person id="123456789"*>... </*person* >. A capacidade de um elemento de incluir outro permite a representação de dados hierárquicos – um aspecto muito importante da XML. Uma *tag* vazia não tem conteúdo e é terminada com />, em vez de >. Por exemplo, a tag vazia <*european*/> poderia ser incluída dentro da *tag* <*person*>...</*person*>.

Atributos: opcionalmente, uma *tag* de início pode incluir pares de nomes e valores de atributo associados, como em *id="123456789"*, conforme mostrado anteriormente. A sintaxe é igual à da HTML, em que um nome de atributo é seguido de um sinal de igualdade e um valor de atributo entre aspas. Múltiplos valores de atributo são separados por espaços.

É uma questão de escolha definir quais itens serão representados como elementos e quais serão representados como atributos. Um elemento geralmente é um contêiner

para dados, enquanto um atributo é usado para rotular esses dados. Em nosso exemplo, *123456789* poderia ser um identificador usado pelo aplicativo, enquanto *name*, *place* e *year* poderiam ser exibidos. Além disso, se os dados contêm subestruturas ou várias linhas, eles devem ser definidos como um elemento. Os atributos servem para valores simples.

Nomes: os nomes de *tags* e atributos na XML geralmente começam com uma letra, mas também podem começar com um sublinhado ou com dois-pontos. Os nomes continuam com letras, dígitos, hífens, sublinhados, dois-pontos ou pontos-finais. Letras maiúsculas e minúsculas são levadas em consideração, isto é, os nomes em XML são *case-sensitive*. Os nomes que começam com *xml* são reservados.

Dados binários: todas as informações nos elementos XML devem ser expressas com dados do tipo caractere, mas a questão é: como representamos elementos criptografados ou *hashing* de códigos de segurança – os quais, como veremos na Seção 9.5, são usados em XML. A resposta é que eles podem ser representados na notação *base64* [Freed e Borenstein 1996], que utiliza apenas os caracteres alfanuméricos, junto a +, / e =, que têm significado especial.

Análise (parsing) e documentos bem formados • Um documento XML deve ser bem formado – isto é, ele deve obedecer às regras sobre sua estrutura. Uma regra básica é que toda *tag* de início tem uma *tag* de fim correspondente. Outra regra básica é que todas as *tags* devem ser corretamente aninhadas, por exemplo <*x*>..<*y*>..</*y*>..</*x*> está correto, enquanto <*x*>..<*y*>....</*x*>.. </*y*>, não. Finalmente, todo documento XML deve ter um único elemento-raiz que englobe todos os outros elementos. Essas regras tornam muito simples implementar analisadores gramaticais (*parsers*) de documentos XML. Um analisador, ao ler um documento XML que não está bem formado, relata um erro fatal.

CDATA: normalmente, os analisadores de XML verificam o conteúdo dos elementos, pois ele pode conter mais estruturas aninhadas. No entanto, se o texto precisa conter um sinal de maior (ou menor) ou aspas, ele deve ser representado de uma maneira especial; por exemplo, *&lt* representa o sinal de menor. Entretanto, se uma seção não deve ser analisada por qualquer motivo – por exemplo, se contiver caracteres especiais – ela pode ser denotada como *CDATA*. Por exemplo, se um nome de lugar precisasse incluir um apóstrofo, ele poderia ser especificado de uma das duas maneiras a seguir:

<place> King&apos Cross </place >
<place> <![CDATA [King's Cross]]></place >

Prólogo XML: todo documento XML deve ter um prólogo como sua primeira linha. O prólogo deve especificar pelo menos a versão de XML que está sendo usada (que, atualmente, é a 1.0). Por exemplo:

<?XML version = "1.0" encoding = "UTF-8" standalone = "yes"?>

O prólogo também pode especificar a codificação (UTF-8 é o padrão e foi explicado na Seção 4.3.2). O termo *codificação* se refere ao conjunto de códigos usados para representar caracteres – sendo o código ASCII o melhor exemplo conhecido. Note que, no prólogo XML, o código ASCII é especificado como *us-ascii*. Outras codificações possíveis incluem ISO-8859-1 (ou Latin-1), uma codificação de oito bits cujos primeiros 128 valores são ASCII, sendo o restante usado para representar os caracteres dos idiomas da Europa Ocidental. Outras codificações de oito bits estão disponíveis para representar outros alfabetos; por exemplo, grego ou cirílico.

Um atributo adicional pode ser usado para informar se o documento é único ou se é dependente de definições externas.

```
<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">
    <pers:name> Smith </pers:name>
    <pers:place> London </pers:place >
    <pers:year> 1984 </pers:year>
</person>
```

Figura 4.11 Ilustração do uso de espaço de nomes na estrutura *Person*.

Espaços de nomes na XML • Tradicionalmente, os espaços de nomes fornecem uma maneira para dar escopo aos nomes. Um espaço de nomes XML é um conjunto de nomes para uma coleção de tipos e atributos de elemento, que é referenciado por um URL. Um espaço de nomes da XML pode ser usado por qualquer outro documento XML, referindo-se ao seu URL.

Qualquer elemento que utilize um espaço de nomes XML pode especificar esse espaço como um atributo chamado *xmlns*, cujo valor é um URL que faz referência a um arquivo que contém as definições do espaço de nomes. Por exemplo:

xmlns:pers = "http://www.cdk5.net/person"

O nome que aparece após *xmlns*, neste caso, *pers*, pode ser usado como prefixo para se referir aos elementos de um espaço de nomes em particular, como mostrado na Figura 4.11. O prefixo *pers* está vinculado a *http://www.cdk5.net/person* para o elemento *person*. Um espaço de nomes se aplica dentro do contexto do par de *tags* de início e fim que o engloba, a não ser que seja sobrescrita por uma nova declaração de espaço de nomes interna a si. Um documento XML pode ser definido em termos de vários espaços de nomes diferentes, cada um dos quais seria referenciado por um prefixo exclusivo.

A convenção de espaço de nomes permite que um aplicativo utilize vários conjuntos de definições externas em diferentes espaços de nomes, sem o risco de conflito de nomes.

Esquemas XML • Um esquema XML [www.w3.org VIII] define os elementos e atributos que podem aparecer em um documento, o modo como os elementos são aninhados, a ordem, o número de elementos e se um elemento está vazio ou se pode conter texto. Para cada elemento, ele define o tipo e o valor padrão. A Figura 4.12 fornece um exemplo de esquema que define os tipos de dados e a estrutura da definição XML da estrutura *Person* da Figura 4.10.

A intenção é que uma definição de esquema possa ser compartilhada por muitos documentos diferentes. Um documento XML, definido de forma a obedecer um esque-

```
<xsd:scheme xmlns:xsd = URL das definições de esquema XML >
    <xsd:element name= "person" type ="personType"/>
        <xsd:complexType name="personType">
            <xsd:sequence>
                <xsd:element name = "name"  type="xs:string"/>
                <xsd:element name = "place"  type="xs:string"/>
                <xsd:element name = "year"  type="xs:positiveInteger"/>
            </xsd:sequence>
            <xsd:attribute name= "id"   type = "xs:positiveInteger"/>
        </xsd:complexType>
</xsd:schema>
```

Figura 4.12 Um esquema XML para a estrutura *Person*.

ma em particular, também pode ser validado por meio desse esquema. Por exemplo, o remetente de uma mensagem SOAP pode usar um esquema XML para codificá-la, e o destinatário usará o mesmo esquema XML para validá-la e decodificá-la.

Definições de tipo de documento: As definições de tipo de documento (DTDs, Document Type Definitions) [www.w3.org VI] foram fornecidas como parte da especificação XML 1.0 para definir a estrutura de documentos XML e ainda são amplamente usadas com esse propósito. A sintaxe das DTDs é diferente do restante da XML e é bastante limitada no sentido do que pode especificar; por exemplo, ela não pode descrever tipos de dados, e suas definições são globais, impedindo que nomes de elemento sejam duplicados. As DTDs não são usadas para definir serviços Web, embora possam ser usadas para definir documentos que são transmitidos por esses.

APIs para acessar XML • Analisadores e geradores de XML estão disponíveis para as linguagens de programação mais usadas. Por exemplo, existe *software* Java para escrever objetos Java como XML (empacotamento) e para criar objetos Java a partir de tais estruturas (desempacotamento). *Software* semelhante está disponível em Python para tipos de dados e objetos Python.

4.3.4 Referências a objetos remotos

Esta seção se aplica apenas às linguagens que suportam o modelo de objeto distribuído, como Java e CORBA. Ela não é relevante para XML.

Quando um cliente invoca um método em um objeto remoto, uma mensagem de invocação é enviada para o processo servidor que contém o objeto remoto. Essa mensagem precisa especificar qual objeto em particular deve ter seu método executado. Uma *referência de objeto remoto* é o identificador de um objeto remoto, válido em todo um sistema distribuído. A referência de objeto remoto é passada na mensagem de invocação para especificar qual objeto deve ser ativado. O Capítulo 5 mostrará que as referências de objeto remoto também são passadas como argumentos e retornadas como resultados de invocações a métodos remotos, que cada objeto remoto tem uma única referência e que essas referências podem ser comparadas para ver se dizem respeito ao mesmo objeto remoto. Agora, discutiremos a representação externa das referências de objeto remoto.

As referências de objeto remoto devem ser geradas de uma forma que garanta sua exclusividade no espaço e no tempo. Em geral, podem existir muitos processos contendo objetos remotos; portanto, as referências de objeto remoto devem ser únicas entre todos os processos, nos vários computadores de um sistema distribuído. Mesmo após um objeto remoto, associado a uma determinada referência, ter sido excluído, é importante que a referência de objeto remoto não seja reutilizada, pois seus invocadores em potencial podem manter referências obsoletas. Qualquer tentativa de invocar um objeto excluído deve produzir um erro, em vez de permitir o acesso a um objeto diferente.

Existem várias maneiras de garantir a exclusividade de uma referência de objeto remoto. Uma delas é construir uma referência concatenando o endereço IP de seu computador e o número de porta do processo que a criou, com a hora de sua criação e um número de objeto local. O número de objeto local é incrementado sempre que um objeto é criado nesse processo.

Juntos, o número de porta e a hora produzem um identificador de processo exclusivo nesse computador. Com essa estratégia, as referências de objeto remoto podem ser representadas com um formato como o que aparece na Figura 4.13. Nas implementações mais simples de RMI, os objetos remotos residem no processo que os criou e existem apenas enquanto esse processo continua a ser executado. Nestes casos, a referência de objeto remoto

32 bits	*32 bits*	*32 bits*	*32 bits*	
Endereço IP	Número de porta	Hora	Número do objeto	Interface do objeto remoto

Figura 4.13 Representação de uma referência de objeto remoto.

pode ser usada como endereço para objeto remoto. Em outras palavras, as mensagens de invocação são enviadas para o endereço IP, o processo e a porta fornecidos pela referência remota.

Para permitir que os objetos remotos sejam migrados para um processo diferente em outro computador, a referência de objeto remoto não deve ser usada como endereço do objeto remoto. A Seção 8.3.3 discutirá uma forma de referência de objeto remoto que permite aos objetos serem invocados em diferentes servidores enquanto existirem.

Os sistemas *peer-to-peer*, Pastry e Tapestry, descritos no Capítulo 10, usam uma forma de objeto remoto que é completamente independente da localização. As mensagens são direcionadas para recursos por meio de um algoritmo de roteamento distribuído.

O último campo da referência de objeto remoto mostrada na Figura 4.13 contém informações sobre a interface do objeto remoto; por exemplo, o nome da interface. Essas informações são importantes para todo processo que recebe uma referência de objeto remoto como argumento ou resultado de uma invocação remota, pois ele precisa conhecer os métodos oferecidos pelo objeto remoto. Esse ponto será abordado novamente na Seção 5.4.2.

4.4 Comunicação por multicast (difusão seletiva)

A troca de mensagens aos pares não é o melhor modelo para a comunicação de um processo com um grupo de outros processos, como o que ocorre, por exemplo, quando um serviço é implementado por meio de diversos processos em computadores diferentes para fornecer tolerância a falhas ou melhorar a disponibilidade. Nestes casos, o emprego de *multicast* é mais apropriado – trata-se de uma operação que permite o envio de uma única mensagem para cada um dos membros de um grupo de processos de tal forma que membros participantes do grupo ficam totalmente transparentes para o remetente. Existem diversas possibilidades para o comportamento desejado de *multicast*. A mais simples não fornece garantias a respeito da entrega ou do ordenamento das mensagens.

As mensagens *multicast* fornecem uma infraestrutura útil para a construção de sistemas distribuídos com as seguintes características:

1. *Tolerância à falha baseada em serviços replicados*: um serviço replicado consiste em um grupo de servidores. As requisições do cliente são difundidas para todos os membros do grupo, cada um dos quais executando uma operação idêntica. Mesmo quando alguns dos membros falham, os clientes ainda podem ser atendidos.
2. *Localização de servidores de descoberta na interligação em rede espontânea*: a Seção 1.3.2 discute os serviços de descoberta para interligação em rede espontânea. Mensagens *multicast* podem ser usadas por servidores e clientes para localizar os serviços de descoberta disponíveis, para registrar suas interfaces ou para pesquisar as interfaces de outros serviços no sistema distribuído.
3. *Melhor desempenho através da replicação de dados*: os dados são replicados para aumentar o desempenho de um serviço – em alguns casos, as réplicas são postas

nos computadores dos usuários. Sempre que os dados mudam, o novo valor é enviado por *multicas*t para os processos que gerenciam as réplicas.

4. *Propagação de notificações de evento*: o *multicast* para um grupo pode ser usado para notificar os processos de quando algo acontece. Por exemplo, no Facebook, quando alguém muda seu status, todos os seus amigos recebem notificações. Do mesmo modo, os protocolos de publicar-assinar podem fazer uso de *multicast* de grupo para disseminar eventos para os assinantes.

A seguir, apresentamos o *multicast* IP para, depois, examinarmos as necessidades do uso de comunicação em grupo para ver quais delas são atendidas pelo *multicast* IP. Para as que não são, propomos mais algumas propriedades dos protocolos de comunicação em grupo, além daquelas fornecidas pelo *multicast* IP.

4.4.1 Multicast IP – uma implementação de comunicação por difusão seletiva

Esta seção discute o *multicast* IP e apresenta a API Java que suporta essa funcionalidade por meio da classe *MulticastSocket*.

Multicast IP • O *multicast* IP permite que o remetente transmita um único datagrama IP para um conjunto de computadores que formam um grupo de *multicast*. O remetente não conhece as identidades dos destinatários individuais nem o tamanho do grupo. Um *grupo multicast* é especificado por um endereço IP classe D (veja a Figura 3.15) – isto é, um endereço cujos primeiros 4 bits são 1110 no protocolo IPv4. Note que os datagramas IP são endereçados para computadores – as portas pertencem aos níveis TCP e UDP.

O fato de ser membro de um grupo *multicast* permite a um computador receber datagramas IP enviados para o grupo. A participação como membro de grupos *multicast* é dinâmica, permitindo aos computadores entrarem ou saírem a qualquer momento e participarem de um número arbitrário de grupos. É possível enviar datagramas para um grupo *multicast* sem ser membro.

Para a programação de aplicativos, o *multicast* IP está disponível apenas por meio de UDP. Um programa aplicativo faz uso de *multicast* enviando datagramas UDP com endereços *multicast* e números de porta normais. Um aplicativo se junta a um grupo *multicast* fazendo seu soquete se unir ao grupo. Em nível IP, um computador pertence a um grupo *multicast* quando um ou mais de seus processos tem soquetes pertencentes a esse grupo. Quando uma mensagem *multicast* chega em um computador, cópias são encaminhadas para todos os soquetes locais que tiverem se juntado ao endereço de *multicast* especificado e estejam vinculados ao número de porta especificado. Os detalhes a seguir são específicos do protocolo IPv4:

Roteadores multicast: os datagramas IP podem ser enviados em *multicast* em uma rede local e na Internet. As transmissões locais exploram a capacidade de *multicast* da rede local, por exemplo, de uma Ethernet. Na Internet, fazem uso de roteadores com suporte a *multicast*, os quais encaminham um único datagrama para roteadores membros em outras redes, onde são novamente transmitidos para os membros locais. Para limitar a distância da propagação de um datagrama *multicast*, o remetente pode especificar o número de roteadores pelos quais pode passar – o que é chamado tempo de vida (*time to live*) ou, abreviadamente, TTL. Para entender como os roteadores sabem quais outros roteadores possuem membros de um grupo *multicast*, veja Comer [2007].

Alocação de endereços multicast: conforme discutido no Capítulo 3, os endereços da Classe D (isto é, endereços no intervalo 224.0.0.0 a 239.255.255.255) são reservados para tráfego *multicast* e são gerenciados globalmente pelo IANA (Internet Assigned Numbers Authority). O gerenciamento desse espaço de endereçamento é revisto anualmente, com a prática atual documentada na RPC 3171 [Albanna *et al.* 2001]. Esse documento define um particionamento do espaço de endereçamento em vários blocos, incluindo:

- Bloco de controle de rede local (224.0.0.0 a 224.0.0.225), para tráfego *multicast* dentro de determinada rede local.
- Bloco de controle de Internet (224.0.1.0 a 224.0.1.225).
- Bloco de controle *ad hoc* (224.0.2.0 a 224.0.255.0) para tráfego que não se encaixa em nenhum outro bloco.
- Bloco de escopo administrativo (239.0.0.0 a 239.255.255.255), que é usado para implementar mecanismo de escopo para tráfego *multicast* (para restringir a propagação).

Os endereços *multicast* podem ser permanentes ou temporários. Existem grupos permanentes mesmo quando não existem membros – seus endereços são atribuídos pela autoridade da Internet e abrangem os vários blocos mencionados anteriormente. Por exemplo, 224.0.1.1 no bloco Internet é reservado para o protocolo NTP (Network Time Protocol), conforme discutido no Capítulo 14, e o intervalo de 224.0.6.000 a 224.0.6.127 no bloco *ad hoc* é reservado para o projeto ISIS (consulte os Capítulos 6 e 18). Existem endereços reservados para diversos propósitos, desde protocolos específicos da Internet até determinadas organizações que fazem muito uso de tráfego *multicast*, incluindo emissoras de multimídia e instituições financeiras. Uma lista completa dos endereços reservados pode ser encontrada no *site* do IANA [www.iana.org II].

O restante dos endereços *multicast* está disponível para os grupos temporários, os quais devem ser criados antes de serem usados e que deixam de existir quando todos os membros tiverem saído. Quando um grupo temporário é criado, ele exige um endereço *multicast* livre para evitar participação acidental em um grupo já existente. O *multicast* provido pelo protocolo IP não trata desse problema diretamente. Se for usado de forma local, soluções relativamente simples são possíveis, por exemplo, configurando o TTL com um valor pequeno, tornando improvável a escolha do mesmo endereço de outro grupo. Entretanto, os programas que usam *multicast* IP em toda a Internet exigem uma solução mais sofisticada para o problema. A RFC 2908 [Thaler *et al.* 2000] descreve uma arquitetura de alocação de endereços *multicast* (MALLOC, Multicast Address Allocation Architecture) para aplicativos em nível de Internet, a qual aloca endereços exclusivos por determinado período e em determinado escopo. Assim, a proposta está intrinsicamente ligada aos mecanismos de escopo mencionados anteriormente. É adotada uma solução cliente-servidor, por meio da qual os clientes solicitam um endereço *multicast* de um servidor de alocação de endereços *multicast* (MAAS, Multicast Address Allocation Server), o qual deve, então, comunicar-se entre os domínios para garantir que as alocações sejam exclusivas por determinado tempo e escopo.

Modelo de falhas para datagramas multicast • No IP, o envio de datagramas *multicast* tem as mesmas características de falhas dos datagramas UDP – isto é, sofre de falhas por omissão. O efeito sobre um *multicast* é que não há garantia de que as mensagens sejam

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
      public static void main(String args[ ]){
      // args fornece o conteúdo da mensagem e o grupo multicast de destino (por exemplo, "228.5.6.7")
         MulticastSocket s =null;
         try {
            InetAddress group = InetAddress.getByName(args[1]);
            s = new MulticastSocket(6789);
            s.joinGroup(group);
            byte [ ] m = args[0].getBytes( );
            DatagramPacket messageOut =
                  new DatagramPacket(m, m.length, group, 6789);
            s.send(messageOut);
            byte[ ] buffer = new byte[1000];
               for(int i=0; i< 3; i++) { // obtém mensagens de outros participantes do grupo
                DatagramPacket messageIn =
                        new DatagramPacket(buffer, buffer.length);
                s.receive(messageIn);
                System.out.println("Received:" + new String(messageIn.getData( )));
            }
            s.leaveGroup(group);
         } catch (SocketException e){System.out.println("Socket: " + e.getMessage( ));
         } catch (IOException e){System.out.println("IO: " + e.getMessage( ));
         } finally { if(s != null) s.close( );}
      }
}
```

Figura 4.14 Um processo se une a um grupo *multicast* para enviar e receber datagramas.

entregues para um membro do grupo em particular, mesmo em face de uma única falha por omissão; ou seja, alguns dos membros do grupo, mas não todos, podem recebê-las. Como não há garantias de que uma mensagem será entregue para um membro de um grupo, esse tipo de comunicação é denominado de *multicast não confiável*. O *multicast confiável* será discutido no Capítulo 15.

API Java para multicast IP • A API Java fornece uma interface de datagrama para *multicast* IP por meio da classe *MulticastSocket*, que é uma subclasse de *DatagramSocket* com a capacidade adicional de se unir a grupos *multicast*. A classe *MulticastSocket* fornece dois construtores alternativos, permitindo que soquetes sejam criados de forma a usar uma porta local especificada (como, por exemplo, a 6789, ilustrada na Figura 4.14) ou qualquer porta local livre. Um processo pode se unir a um grupo *multicast*, invocando o método *joinGroup* em seu soquete. Efetivamente, o soquete se junta a um grupo *multicast* em determinada porta e receberá, nessa porta, os datagramas enviados para este grupo por processos existentes em outros computadores. Um processo pode sair de um grupo especificado invocando o método *leaveGroup* em seu soquete *multicast*.

No exemplo da Figura 4.14, os argumentos do método *main* especificam uma mensagem e o endereço *multicast* de um grupo (por exemplo, *"228.5.6.7").* Após se unir a

esse grupo *multicast*, o processo cria uma instância de *DatagramPacket* contendo a mensagem e a envia por intermédio de seu soquete *multicast* para o grupo *multicast* na porta 6789. Depois disso, por meio desse mesmo soquete, o processo recebe três mensagens *multicast* destinadas a esse grupo e a essa porta. Quando várias instâncias desse programa são executadas simultaneamente em diferentes computadores, todas elas se unem ao mesmo grupo e cada uma deve receber sua própria mensagem e as mensagens daqueles que se uniram depois dela.

A API Java permite que o TTL seja configurado para um soquete *multicast* por meio do método *setTimeToLive*. O padrão é 1, permitindo que o *multicast* se propague apenas na rede local.

Um aplicativo implementado sobre *multicast* IP pode usar mais de uma porta. Por exemplo, o aplicativo MultiTalk [mbone], que permite a grupos de usuários manterem conversas baseadas em texto, possui uma porta para enviar e receber dados e outra para trocar dados de controle.

4.4.2 Confiabilidade e ordenamento

A seção anterior expôs o modelo de falhas do *multicast* IP, isto é, ele sofre de falhas por omissão. Para os envios *multicast* feitos em uma rede local que possui suporte nativo para que um único datagrama chegue a vários destinos, qualquer um deles pode, individualmente, descartar a mensagem porque seu *buffer* está cheio. Além disso, um datagrama enviado de um roteador *multicast* para outro pode ser perdido, impedindo, assim, que todos os destinos que estejam além desse roteador recebam a mensagem.

Outro fator é que qualquer processo pode falhar. Se um roteador *multicast* falhar, os membros do grupo que estiverem além desse roteador não receberão a mensagem, embora os membros locais possam receber.

O ordenamento é outro problema. Os datagramas IP enviados por várias redes interligadas não chegam necessariamente na ordem em que foram emitidos, com o possível efeito de que alguns membros do grupo recebam os datagramas de um único remetente em uma ordem diferente dos outros membros. Além disso, as mensagens enviadas por dois processos diferentes não chegarão necessariamente na mesma ordem em todos os membros do grupo.

Alguns exemplos dos efeitos da confiabilidade e do ordenamento • Agora, consideremos o efeito da semântica da falha no *multicast* IP nos quatro exemplos de uso de replicação da introdução da Seção 4.4.

1. *Tolerância a falhas baseada em serviços replicados*: considere um serviço replicado que consiste nos membros de um grupo de servidores que começam no mesmo estado inicial e sempre executam as mesmas operações, na mesma ordem, de modo a permanecerem consistentes uns com os outros. Essa aplicação *multicast* impõe que todas as réplicas, ou nenhuma delas, devam receber cada pedido para executar uma operação – se uma perder um pedido, ela se tornará inconsistente com relação às outras. Na maioria dos casos, esse serviço exige que todos os membros recebam as mensagens de requisição na mesma ordem dos outros.
2. *Localização dos servidores de descoberta na interligação em rede espontânea*: desde que qualquer processo que queira localizar os servidores de descoberta faça, após sua inicialização, o envio periódico de requisições em *multicast*, uma requisi-

ção ocasionalmente perdida não será um problema na localização de um servidor de descoberta. Na verdade, o Jini usa *multicast* IP em seu protocolo de localização dos servidores de descoberta. Isso será descrito na Seção 19.2.1.

3. *Melhor desempenho através de dados replicados*: considere o caso em que os próprios dados replicados, em vez de as operações sobre eles, são distribuídos por meio de mensagens de *multicast*. O efeito de mensagens perdidas e ordem inconsistente dependeria do método de replicação e da importância de todas as réplicas estarem totalmente atualizadas.
4. *Propagação de notificações de evento*: o aplicativo em particular determina a qualidade exigida do *multicast*. Por exemplo, os serviços de pesquisa Jini utilizam *multicast* IP para anunciar sua existência (veja a Seção 19.2.1).

Esses exemplos sugerem que algumas aplicações exigem um protocolo *multicast* que seja mais confiável do que o oferecido pelo IP. Portanto, há necessidade de um *multicast confiável* – no qual qualquer mensagem transmitida ou é recebida por todos os membros de um grupo ou não é recebida por nenhum deles. Os exemplos também sugerem que algumas aplicações têm forte necessidade de ordem, cujo máximo rigor é chamado de *multicast totalmente ordenado*, em que todas as mensagens transmitidas para um grupo chegam a todos os membros na mesma ordem.

O Capítulo 15 definirá e mostrará como implementar *multicast confiável* e várias garantias de ordenamento, incluindo o totalmente ordenado.

4.5 Virtualização de redes: redes de sobreposição

A vantagem dos protocolos de comunicação da Internet é que eles fornecem, por intermédio de suas APIs (Seção 4.2), um conjunto muito eficiente de elementos básicos para a construção de *software* distribuído. Contudo, uma crescente variedade de diferentes tipos de aplicativo (incluindo, por exemplo, o compartilhamento de arquivos *peer-to-peer* e o Skype) coexistem na Internet. Seria impraticável tentar alterar os protocolos da Internet de acordo com cada um dos muitos aplicativos executados por meio deles – o que poderia melhorar um, poderia ser prejudicial para outro. Além disso, o serviço de transporte de datagramas IP é implementado por um grande e sempre crescente número de tecnologias de rede. Esses dois fatores têm provocado o interesse na virtualização das redes.

A virtualização de redes [Petersen *et al*. 2005] ocupa-se com a construção de muitas redes virtuais diferentes sobre uma rede já existente, como a Internet. Cada rede virtual pode ser projetada para suportar um aplicativo distribuído em particular. Por exemplo, uma rede virtual poderia suportar *streaming* de multimídia, como iPlayer da BBC, BoxeeTV [boxee.tv] ou Hulu [hulu.com], e coexistir com outra que suportasse um game *online* para vários jogadores, ambas funcionando na mesma rede subjacente. Isso sugere uma resposta para o dilema levantado pelo princípio fim-a-fim de Salzer (consulte a Seção 2.3.3): uma rede virtual específica para um aplicativo pode ser construída sobre uma rede já existente e ser otimizada para esse aplicativo em particular, sem alterar as características da rede subjacente.

O Capítulo 3 mostrou que as redes de computador possuem esquemas de endereçamento, protocolos e algoritmos de roteamento; da mesma maneira, cada rede virtual tem seu próprio esquema de endereçamento, protocolos e algoritmos de roteamento específicos, mas redefinidos para satisfazer as necessidades de tipos de aplicativo em particular.

4.5.1 Redes de sobreposição

Uma *rede de sobreposição (overlay)* é uma rede virtual consistindo em nós e enlaces virtuais, a qual fica sobre uma rede subjacente (como uma rede IP) e oferece algo que de outro modo não é fornecido:

- um serviço personalizado de acordo com as necessidades de um tipo de aplicativo ou um serviço de nível mais alto em particular, como distribuição de conteúdo multimídia;
- funcionamento mais eficiente em determinado ambiente interligado em rede; por exemplo, roteamento em uma rede *ad hoc*;
- um recurso adicional; por exemplo, comunicação por *multicast* ou segura.

Isso leva a uma grande variedade de tipos de sobreposição, conforme mostra a Figura 4.15. As redes de sobreposição têm as seguintes vantagens:

- Elas permitem a definição de novos serviços de rede sem exigir mudanças na rede subjacente – um ponto crucial, dado o nível de padronização nessa área e as dificuldades de correção da funcionalidade de roteador subjacente.
- Elas estimulam a experimentação com serviços de rede e a personalização de serviços para tipos de aplicativo específicos.
- Várias sobreposições podem ser definidas e coexistir, sendo o resultado final uma arquitetura de rede mais aberta e extensível.

As desvantagens são que as redes de sobreposição introduzem um nível extra de indireção (e portanto podem acarretar queda no desempenho) e aumentam a complexidade dos serviços de rede, quando comparadas, por exemplo, com a arquitetura relativamente simples das redes TCP/IP.

As redes de sobreposição podem ser relacionadas ao conhecido conceito de camadas lógicas (conforme apresentado nos Capítulos 2 e 3). Sobreposições são camadas lógicas, mas que existem fora da arquitetura padrão (como a pilha TCP/IP) e exploram os graus de liberdade resultantes. Em particular, os desenvolvedores de sobreposição estão livres para redefinir os elementos básicos de uma rede, conforme mencionado anteriormente, inclusive o modo de endereçamento, os protocolos empregados e a estratégia de roteamento, frequentemente introduzindo estratégias radicalmente diferentes, mais personalizadas para os tipos de aplicativo específicos dos ambientes operacionais. Por exemplo, as tabelas de *hashing* distribuídas introduzem um estilo de endereçamento baseado em um espaço de chaves e também constróem uma topologia de maneira tal que um nó ou possui a chave ou tem uma referência para um nó que está mais próximo do proprietário da chave. Esse estilo de roteamento é conhecido como roteamento baseado em chave. A topologia aparece mais comumente na forma de anel.

Exemplificamos o uso bem-sucedido de uma rede de sobreposição discutindo o Skype. Mais exemplos de sobreposições vão ser dados ao longo de todo o livro. Por exemplo, o Capítulo 10 apresenta detalhes dos protocolos e das estruturas adotadas pelo compartilhamento de arquivos *peer-to-peer*, junto a mais informações sobre as tabelas de *hashing* distribuídas. O Capítulo 19 considera as redes *ad hoc* sem fio e as redes tolerantes à interrupção no contexto da computação móvel e ubíqua, e o Capítulo 20 examina o suporte de redes de sobreposição para *streaming* de multimídia.

Motivação	*Tipo*	*Descrição*
Adequado às necessidades do aplicativo	Tabelas de *hashing* distribuídas	Uma das classes mais importantes de redes de sobreposição, oferecendo um serviço que gerencia um mapeamento de chaves para valores em um número potencialmente grande de nós, de maneira completamente descentralizada (semelhante a uma tabela *de hashing* padrão, mas em um ambiente de rede).
	Compartilhamento de arquivos *peer-to-peer*	Estruturas de sobreposição que se concentram na construção de mecanismos de endereçamento e roteamento personalizados para suportar a descoberta cooperativa e o uso de arquivos (por exemplo, *download*).
	Redes de distribuição de conteúdo	Sobreposições que incluem diversas estratégias de replicação, uso de cache e posicionamento para fornecer desempenho aprimorado em termos de entrega de conteúdo para usuários web; usados para aceleração na web e para oferecer o desempenho em tempo real exigido por fluxos de vídeo [www.kontiki.com].
Adequado ao estilo de rede	Redes *ad hoc* sem fio	Redes de sobreposição que fornecem protocolos de roteamento personalizados para redes *ad hoc* sem fio, incluindo esquemas proativos que efetivamente constroem uma topologia de roteamento sobre nós subjacentes e esquemas reativos que estabelecem rotas sob demanda, normalmente suportadas por inundação.
	Redes tolerantes a rompimento	Sobreposições projetadas para operar em ambientes hostis que sofrem de falhas significativas de nó ou enlace e, potencialmente, grandes atrasos.
Oferecimento de recursos adicionais	*Multicast*	Um dos primeiros usos das redes de sobreposição na Internet, fornecendo acesso a serviços *multicast* em que não há roteadores *multicast*; complementa o trabalho de Van Jacobsen, Deering e Casner, com sua implementação do MBone (ou *Multicast* Backbone) [mbone].
	Resiliência	Busca melhoria na robusteza e na disponibilidade de caminhos de Internet [nms.csail.mit.edu].
	Segurança	Redes de sobreposição que oferecem maior segurança sobre a rede IP subjacente, incluem redes privadas virtuais, por exemplo, conforme discutido na Seção 3.4.8.

Figura 4.15 Tipos de sobreposição.

4.5.2 Skype: um exemplo de rede de sobreposição

O Skype é um aplicativo *peer-to-peer* que oferece voz sobre IP (VoIP). Possui também troca de mensagens instantâneas, videoconferência e interfaces para o serviço de telefonia padrão, por meio de SkypeIn e SkypeOut. O *software* foi desenvolvido

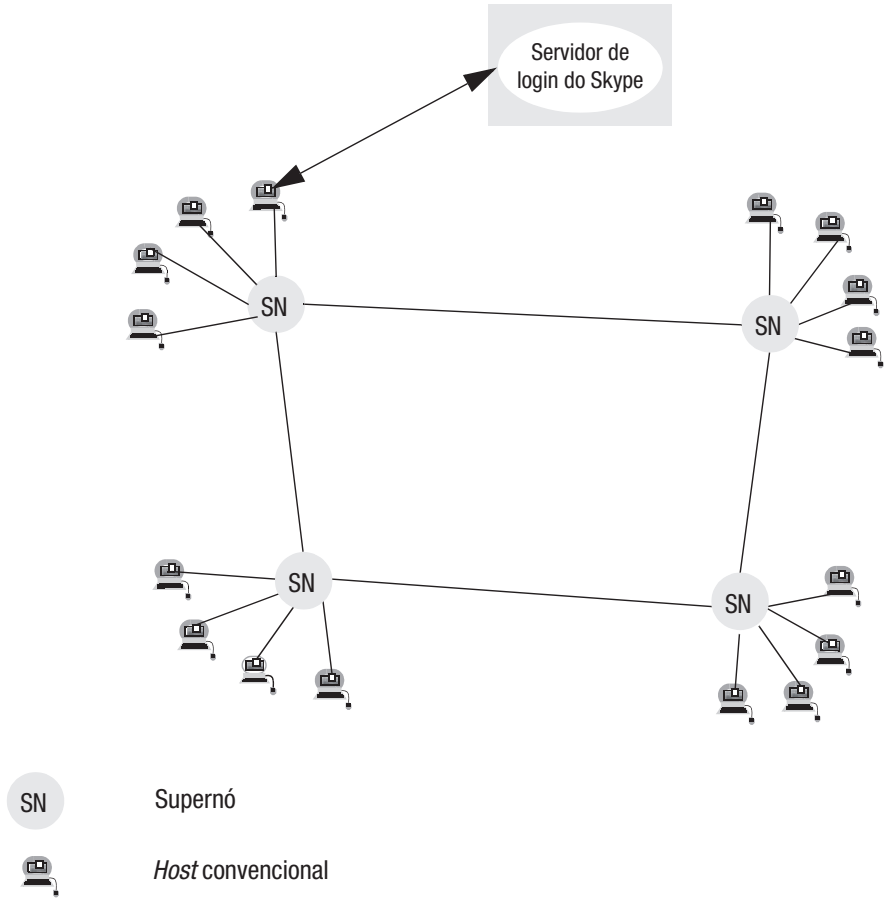

Figura 4.16 Arquitetura de sobreposição do Skype.

pelo Kazaa em 2003 e, assim, compartilha muitas das características do aplicativo de compartilhamento de arquivos *peer-to-peer* do Kazaa [Leibowitz *et al.* 2003]. Ele é amplamente distribuído, com uma estimativa de 370 milhões de usuários no início de 2009.

O Skype é um excelente estudo de caso do uso de redes de sobreposição em sistemas reais (e de grande escala), indicando como funcionalidade avançada pode ser fornecida de maneira específica do aplicativo e sem modificar a arquitetura básica da Internet. O Skype é uma rede virtual no sentido de que estabelece conexões entre pessoas (assinantes do Skype correntemente ativos). Nenhum endereço IP ou porta é necessária para estabelecer uma chamada. A arquitetura da rede virtual que dá suporte ao Skype não é amplamente publicada, mas pesquisadores estudaram o Skype usando diversos métodos, incluindo análise de tráfego, sendo que agora seus princípios são de domínio público. Muitos dos detalhes da descrição a seguir foram extraídos do artigo de Baset e Schulzrinne [2006], o qual contém um estudo pormenorizado do comportamento do Skype.

Arquitetura do Skype • O Skype é baseado em uma infraestrutura *peer-to-peer* que consiste em máquinas normais de usuário (referidas como *hosts*) e supernós – os quais são *hosts* normais do Skype que têm recursos suficientes para cumprir sua função avançada. Os supernós são selecionados de acordo com a demanda, com base em diversos critérios, incluindo a largura de banda disponível, a acessibilidade (a máquina deve ter um endereço IP global e não estar oculta por um roteador com NAT habilitado, por exemplo) e também a disponibilidade (com base no período de tempo em que o Skype esteve funcionando continuamente nesse nó). Essa estrutura global está ilustrada na Figura 4.16.

Conexão de usuário • Os usuários do Skype são autenticados por meio de um servidor de *login* conhecido. Então, eles entram em contato com um supernó selecionado. Para se conseguir isso, cada cliente mantém uma cache de identidades de supernó (isto é, pares endereço IP e número de porta). No primeiro *login*, essa cache é preenchida com os endereços de cerca de sete supernós e, com o passar do tempo, o cliente constrói e mantém um conjunto muito maior (talvez várias centenas).

Busca de usuários • O principal objetivo dos supernós é fazer a pesquisa eficiente de índices globais de usuários, os quais são distribuídos pelos supernós. A pesquisa é orquestrada pelo supernó escolhido do cliente e envolve expandir a busca para outros supernós até que o usuário seja encontrado. Em média, oito supernós são contactados. Uma busca de usuário normalmente leva de três a quatro segundos para *hosts* que têm um endereço IP global (e um pouco mais – de cinco a seis segundos – se estiver atrás de um roteador com NAT habilitado). De acordo com as experiências, parece que os nós intermediários envolvidos na busca colocam os resultados na cache para melhorar o desempenho.

Conexão de voz • Uma vez encontrado o usuário desejado, o Skype estabelece uma conexão de voz entre as duas partes, usando TCP para sinalizar pedidos e términos de chamada e UDP ou TCP para o *streaming* de áudio. UDP é preferido, mas TCP, junto ao uso de um nó intermediário, é usado em determinadas circunstâncias para contornar *firewalls* (consulte Baset e Schulzrinne [2006] para detalhes). O *software* usado para codificar e decodificar áudio desempenha um papel chave no fornecimento da chamada de excelente qualidade normalmente obtida no Skype, e os algoritmos associados são cuidadosamente personalizados para operar no ambiente da Internet em 32 kbit/s e acima.

4.6 Estudo de caso: MPI

A passagem de mensagens foi apresentada na Seção 4.2.1, que esboça os princípios básicos da troca de mensagens entre dois processos usando operações *send* e *receive*. A variante síncrona da passagem de mensagens é conseguida com o bloqueio de chamadas de *send* e *receive*, enquanto a variante assíncrona exige uma forma de *send* sem bloqueio. O resultado final é um paradigma de programação distribuída leve, eficiente e mínima de muitas maneiras.

Esse estilo de programação distribuída é atraente nos tipos de sistema em que o desempenho é fundamental, mais notadamente na computação de alto desempenho. Nesta seção, apresentamos um estudo de caso do padrão Message Passing Interface, desenvolvido pela comunidade de computação de alto desempenho. O padrão MPI apareceu pela primeira vez em 1994 no MPI Forum [www.mpi-forum.org], como uma reação contra a ampla variedade de estratégias patenteadas que estavam sendo usadas para passagem de

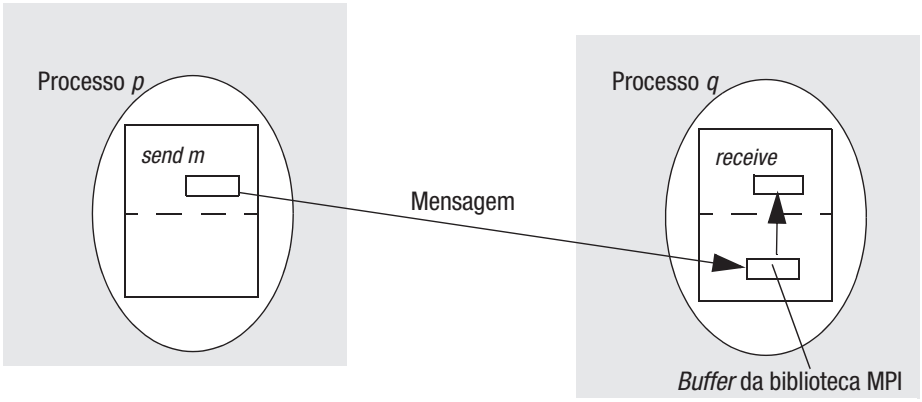

Figura 4.17 Uma visão geral da comunicação ponto a ponto no padrão MPI.

mensagens nesse setor. O padrão também teve forte influência na área de grades computacionais (*grid*), discutida no Capítulo 9 – por exemplo, por meio do desenvolvimento do GridMPI [www.gridmpi.org]. O objetivo do MPI Forum era manter a simplicidade, a praticabilidade e a eficiência inerentes da estratégia da passagem de mensagens, mas melhorar isso com portabilidade, apresentando uma interface padronizada, independente do sistema operacional ou da interface de soquete específica da linguagem de programação. O padrão MPI também foi projetado de forma a ser flexível, e o resultado é uma especificação de passagem de mensagens abrangente em todas as suas variantes (com mais de 115 operações). Os aplicativos usam a interface MPI por intermédio de uma biblioteca de passagem de mensagens – disponível para diversos sistemas operacionais e linguagens de programação, incluindo C++ e Fortran.

O modelo arquitetônico subjacente do padrão MPI é relativamente simples e aparece na Figura 4.17. Ele é semelhante ao modelo apresentado na Seção 4.2.1, mas com a capacidade acrescentada de ter *buffers* da biblioteca MPI explicitamente no remetente e no destinatário, gerenciados pela biblioteca MPI e utilizados para manter dados em trânsito. Note que essa figura mostra um único caminho do remetente para o destinatário por intermédio do *buffer* da biblioteca MPI do destinatário (outras opções, por exemplo usando o *buffer* da biblioteca MPI do remetente, aparecerão a seguir).

Para termos uma ideia dessa complexidade, vamos examinar diversas variantes da operação *send* resumidas na Figura 4.18. Isso é um refinamento da visão da passagem de mensagens apresentada na Seção 4.2.1, oferecendo mais escolhas e controle e realmente separando semântica da passagem de mensagens síncrona/assíncrona e com bloqueio/sem bloqueio.

Vamos começar examinando as quatro operações de bloqueio apresentadas na coluna associada da Figura 4.18. O segredo para entender esse conjunto de operações é compreender que bloqueio é interpretado como "bloqueado até que seja seguro retornar", no sentido de que dados do aplicativo foram copiados no ambiente MPI e, assim, estão em trânsito ou foram entregues e, portanto, o *buffer* do aplicativo pode ser reutilizado (por exemplo, para a próxima operação *send*). Isso possibilita várias interpretações do que significa "ser seguro para retornar". A operação *MPI_Send* é genérica e simplesmente exige que esse nível de segurança seja fornecido (na prática, isso é frequentemente implementado usando-se *MPI_Ssend*). *MPI_Ssend* é exatamente igual à passagem de mensagens síncrona (e com bloqueio), conforme apresentado na Seção 4.2.1, com segurança

Operações send	*Com bloqueio*	*Sem bloqueio*
Genéricas	*MPI_Send*: o remetente bloqueia até que seja seguro retornar; isto é, até que a mensagem esteja em trânsito ou tenha sido entregue e que, portanto, o *buffer* de aplicativo do remetente possa ser reutilizado.	*MPI_Isend*: a chamada retorna imediatamente e o programador recebe um identificador de pedido de comunicação, o qual, então, pode ser usado para verificar o andamento da chamada via *MPI_Wait* ou *MPI_Test*.
Síncronas	*MPI_Ssend*: o remetente e o destinatário são sincronizados e a chamada só retornará quando a mensagem tiver sido entregue no endereço do destinatário.	*MPI_Issend*: semelhante a *MPI_Isend*, mas com *MPI_Wait* e *MPI_Test* indicando se a mensagem foi entregue no endereço do destinatário.
Com buffer	*MPI_Bsend*: o remetente aloca explicitamente um *buffer da* biblioteca MPI (usando uma chamada de *MPI_Buffer_attach* separada), e a chamada retorna quando os dados são copiados com sucesso nesse *buffer*.	*MPI_Ibsend*: semelhante a MPI_Isend, mas com *MPI_Wait* e *MPI_Test* indicando se a mensagem foi copiada no *buffer* MPI do remetente e, portanto, está em trânsito.
Prontas	*MPI_Rsend*: a chamada retorna quando o *buffer* de aplicativo do remetente pode ser reutilizado (como com *MPI_Send*), mas o programador também está indicando para a biblioteca que o destinatário está pronto para receber a mensagem, resultando em uma possível otimização da implementação subjacente.	*MPI_Irsend*: o efeito é igual ao de *MPI_Isend*, mas com *MPI_Rsend* o programador está indicando para a implementação subjacente que o destinatário está realmente pronto para receber (resultando nas mesmas otimizações).

Figura 4.18 Operações *send* selecionadas no padrão MPI.

interpretada como entregue, enquanto *MPI_Bsend* tem semântica mais fraca, no sentido de que a mensagem foi copiada para o *buffer* da biblioteca MPI previamente alocado e ainda está em trânsito. *MPI_Rsend* é uma operação bastante curiosa, na qual o programador especifica que sabe que o destinatário está pronto para receber a mensagem. Se isso é conhecido, a implementação subjacente pode ser otimizada, pois não há necessidade de verificar se existe um *buffer* disponível para receber a mensagem, evitando um *handshake*. Claramente, essa é uma operação muito perigosa, que vai falhar se a suposição a respeito de estar pronto for inválida. Na figura é possível observar a elegante simetria das operações *send* sem bloqueio, desta vez definidas na semântica das operações associadas *MPI_Wait* e *MPI_Test* (note também a convenção de atribuição de nomes coerente em todas as operações).

O padrão também suporta operações *receive* com e sem bloqueio (*MPI_recv* e *MPI_Irecv,* respectivamente), e as variantes de *send* e *receive* podem formar pares em qualquer combinação, oferecendo ao programador um rico controle sobre a semântica da passagem de mensagens. Além disso, o padrão define um rico conjunto de primitivas de comunicação entre vários processos (referido como comunicação coletiva), incluindo, por exemplo, as operações *scatter* (um para muitos) e *gather* (muitos para um).

4.7 Resumo

A primeira seção deste capítulo mostrou que os protocolos de transmissão da Internet fornecem dois blocos básicos a partir dos quais os protocolos em nível de aplicativos podem ser construídos. Há um compromisso interessante entre os dois protocolos: o protocolo UDP fornece um recurso simples de passagem de mensagem, que sofre de falhas por omissão, mas não tem penalidades de desempenho incorporadas. Por outro lado, em boas condições, o protocolo TCP garante a entrega das mensagens, mas à custa de mensagens adicionais e com latência e custos de armazenamento mais altos.

A segunda seção mostrou três estilos alternativos de empacotamento. O CORBA e seus predecessores optam por empacotar os dados para uso pelos destinatários que têm conhecimento anterior dos tipos de seus componentes. Em contraste, quando a linguagem Java serializa dados, ela inclui informações completas sobre os tipos de seu conteúdo, permitindo ao destinatário reconstruí-los puramente a partir do conteúdo. A linguagem XML, assim como a Java, inclui informações completas sobre os tipos de dados. Outra grande diferença é que o CORBA exige uma especificação dos tipos dos dados a serem empacotados (no IDL) para gerar os métodos de empacotamento e desempacotamento, enquanto a linguagem Java usa reflexão para serializar e desserializar objetos. Uma variedade de meios é usada para gerar código XML, dependendo do contexto. Por exemplo, muitas linguagens de programação, incluindo a Java, fornecem processadores para fazer a transformação entre objetos em nível de XML e de linguagem.

Mensagens *multicast* são usadas na comunicação entre os membros de um grupo de processos. O *multicast* IP é disponibilizado tanto para redes locais como para a Internet e tem a mesma semântica de falhas que os datagramas UDP; porém, apesar de sofrer de falhas por omissão, é uma ferramenta útil para muitas aplicações *multicast*. Algumas aplicações têm requisitos mais restritos – em particular, o de que a distribuição *multicast* deva ser atômica; isto é, ela deve ter uma distribuição do tipo tudo ou nada. Outros requisitos do *multicast* estão relacionados ao ordenamento das mensagens, o mais exigente, que impõe que todos os membros de um grupo devem receber todas as mensagens na mesma ordem.

Multicast também pode ser suportado pelas redes de sobreposição nos casos em que, por exemplo, *multicast* IP não é suportado. Mais geralmente, as redes de sobreposição oferecem um serviço de virtualização da arquitetura da rede, permitindo que serviços especializados sejam criados sobre a infraestrutura de interligação em rede subjacente, por exemplo, UDP ou TCP. As redes de sobreposição resolvem parcialmente os problemas associados ao princípio fim-a-fim de Saltzer, permitindo a geração de abstrações de rede mais específicas para o aplicativo.

O capítulo terminou com um estudo de caso da especificação MPI desenvolvida pela comunidade de computação de alto desempenho e que apresenta suporte flexível para a passagem de mensagens, junto a suporte adicional para passagem de mensagens por comunicação coletiva.

Exercícios

4.1 É concebivelmente útil que uma porta tenha vários receptores? *páginas 148*

4.2 Um servidor cria uma porta que ele utiliza para receber pedidos dos clientes. Discuta os problemas de projeto relativos ao relacionamento entre o nome dessa porta e os nomes usados pelos clientes. *páginas 148*

4.3 Os programas das Figuras 4.3 e 4.4 estão disponíveis no endereço [www.cdk5.net/ipc] (em inglês). Utilize-os para fazer uma série de testes para determinar as condições nas quais os datagramas, às vezes, são descartados. Dica: o programa cliente deve ser capaz de variar o número de mensagens enviadas e seus tamanhos; o servidor deve detectar quando uma mensagem de um cliente específico é perdida. *páginas 150*

4.4 Use o programa da Figura 4.3 para fazer um programa cliente que leia repetidamente uma linha de entrada do usuário, a envie para o servidor em uma mensagem datagrama UDP e receba uma mensagem do servidor. O cliente estabelece um tempo limite em seu soquete para que possa informar o usuário quando o servidor não responder. Teste este programa cliente com o servidor da Figura 4.4. *páginas 150*

4.5 Os programas das Figuras 4.5 e 4.6 estão disponíveis no endereço [www.cdk5.net/ipc] (em inglês). Modifique-os de modo que o cliente leia repetidamente uma linha de entrada do usuário e a escreva no fluxo. O servidor deve ler repetidamente o fluxo, imprimindo o resultado de cada leitura. Faça uma comparação entre o envio de dados em mensagens de datagrama UDP e por meio de um fluxo. *página 153*

4.6 Use os programas desenvolvidos no Exercício 4.5 para testar o efeito sobre o remetente quando o receptor falha e vice-versa. *página 153*

4.7 O XDR da Sun empacota dados convertendo-os para uma forma *big-endian* antes de transmiti-los. Discuta as vantagens e desvantagens desse método em comparação com o CDR do CORBA. *página 160*

4.8 O XDR da Sun alinha cada valor primitivo em um limite de quatro bytes, enquanto o CDR da CORBA alinha um valor primitivo de tamanho *n* em um limite de *n* bytes. Discuta os compromissos na escolha dos tamanhos ocupados pelos valores primitivos. *página 160*

4.9 Por que não há nenhuma tipagem de dados explícita no CDR do CORBA? *página 160*

4.10 Escreva um algoritmo, em pseudocódigo, para descrever o procedimento de serialização mostrado na Seção 4.3.2. O algoritmo deve mostrar quando identificadores são definidos ou substituídos por classes e instâncias. Descreva a forma serializada que seu algoritmo produziria para uma instância da seguinte classe *Couple*.

```
class Couple implements Serializable{
        private Person one;
        private Person two;
        public Couple(Person a, Person b) {
                one = a;
                two = b;
        }
}
```

página 162

4.11 Escreva um algoritmo, em pseudocódigo, para descrever a desserialização da forma serializada produzida pelo algoritmo definido no Exercício 4.10. Dica: use reflexão para criar uma classe a partir de seu nome, um construtor a partir de seus tipos de parâmetro e uma nova instância de um objeto a partir do construtor e dos valores de argumento. *página 162*

4.12 Por que dados binários não podem ser representados diretamente em XML, por exemplo, como valores em Unicode? Os elementos XML podem transportar *strings* representados como *base64*. Discuta as vantagens ou desvantagens de usar esse método para representar dados binários. *página 164*

4.13 Defina uma classe cujas instâncias representem referências de objeto remoto. Ela deve conter informações semelhantes àquelas mostradas na Figura 4.13 e deve fornecer métodos de acesso necessários para os protocolos de nível mais alto (consulte requisição-resposta, no Capítulo 5, para ver um exemplo). Explique como cada um dos métodos de acesso será usado por esse protocolo. Dê uma justificativa para o tipo escolhido para a variável de instância que contém informações sobre a interface do objeto remoto. *página 168*

4.14 O *multicast* IP fornece um serviço que sofre de falhas por omissão. Faça uma série de testes, baseada no programa da Figura 4.14, para descobrir as condições sob as quais uma mensagem *multicast* às vezes é eliminada por um dos membros do grupo *multicast*. O kit de testes deve ser projetado de forma a permitir a existência de vários processos remetentes. *página 170*

4.15 Esboce o projeto de um esquema que utilize retransmissões de mensagem *multicast* IP para superar o problema das mensagens descartadas. Seu esquema deve levar em conta os seguintes pontos:

i) podem existir vários remetentes;

ii) geralmente apenas uma pequena parte das mensagens é descartada;

iii) os destinatários podem não enviar uma mensagem necessariamente dentro de um limite de tempo em particular.

Suponha que as mensagens que não são descartadas chegam na ordem do remetente. *página 173*

4.16 Sua solução para o Exercício 4.15 deve ter superado o problema das mensagens descartadas no *multicast* IP. Em que sentido sua solução difere da definição de *multicast* confiável? *página 173*

4.17 Imagine um cenário no qual mensagens *multicast* enviadas por diferentes clientes são entregues em ordens diferentes a dois membros do grupo. Suponha que esteja em uso alguma forma de retransmissões de mensagem, mas que as mensagens que não são descartadas cheguem na ordem do remetente. Sugira o modo como os destinos poderiam remediar essa situação. *página 173*

4.18 Reveja a arquitetura da Internet apresentada no Capítulo 3 (consulte as Figuras 3.12 e 3.14). Que impacto a introdução das redes de sobreposição tem sobre essa arquitetura, em particular sobre a visão conceitual do programador em relação à Internet? *página 175*

4.19 Quais são os principais argumentos para a adoção de uma estratégia de supernós no Skype? *página 177*

4.20 Conforme discutido na Seção 4.6, o padrão MPI oferece diversas variantes da operação *send*, incluindo *MPI_Rsend*, que presume que o destinatário está pronto para receber no momento do envio. Quais otimizações são possíveis na implementação se essa suposição estiver correta e quais são as repercussões no caso de ser falsa? *página 180*