5

Invocação Remota

Este capítulo examina os paradigmas de invocação remota apresentados no Capítulo 2 (as técnicas de comunicação indireta serão tratadas no Capítulo 6). Começaremos examinando o serviço mais primitivo, a comunicação por requisição-resposta, que representa aprimoramentos relativamente pequenos nas primitivas de comunicação entre processos, discutidas no Capítulo 4. Em seguida, o capítulo examina as duas técnicas de invocação remota mais importantes para comunicação em sistemas distribuídos:

- A estratégia de chamada de procedimento remoto (RPC) estende a abstração de programação comum da chamada de procedimento para os ambientes distribuídos, permitindo que um processo chame um procedimento em um nó remoto como se fosse local.
- A invocação a método remoto (RMI) é semelhante à RPC, mas para objetos distribuídos, com vantagens adicionais em usar conceitos de programação orientada a objetos em sistemas distribuídos e em estender o conceito de referência de objeto para o ambiente distribuído global e permitir o uso de referências de objeto como parâmetros em invocações remotas.

O capítulo também apresenta a RMI Java como um estudo de caso da estratégia de invocação a método remoto (uma maior compreensão pode ser obtida no Capítulo 8, no qual examinaremos o CORBA).

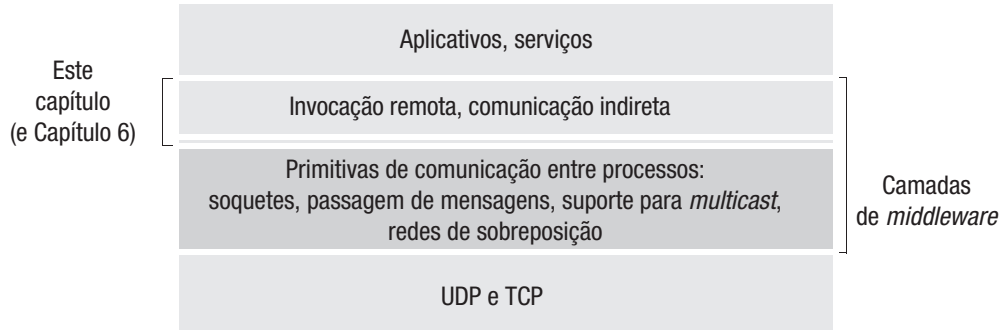

Figura 5.1 Camadas de *middleware.*

5.1 Introdução

Este capítulo trata de como os processos (ou entidades, em um nível de abstração mais alto, como objetos ou serviços) se comunicam em um sistema distribuído, examinando, em particular, os paradigmas de invocação remota definidos no Capítulo 2:

- Os *protocolos de requisição-resposta* representam um padrão sobre a passagem de mensagens e suportam a troca bilateral de mensagens, como a encontrada na computação cliente-servidor. Em particular, tais protocolos fornecem suporte de nível relativamente baixo para solicitar a execução de uma operação remota e suporte direto para RPC e RMI, discutidas a seguir.
- O mais antigo, e talvez mais conhecido exemplo de modelo mais amigável para o programador foi a extensão do modelo de chamada de procedimentos convencional para os sistemas distribuídos, a *chamada de procedimento remoto* (*RPC*, *Remote Procedure Call*), que permite aos programas clientes chamarem procedimentos de forma transparente em programas servidores que estejam sendo executados em processos separados e, geralmente, em computadores diferentes do cliente.
- Nos anos 90, o modelo da programação baseada em objetos foi ampliado para permitir que objetos de diferentes processos se comunicassem por intermédio da *invocação a método remoto* (*RMI*, *Remote Method Invocation*). A RMI é uma extensão da invocação a método local que permite a um objeto que está em um processo invocar os métodos de um objeto que está em outro processo.

Note que usamos o termo "RMI" para nos referirmos à invocação a método remoto de uma maneira genérica – isso não deve ser confundido com exemplos particulares de invocação a método remoto, como a RMI Java.

Retornando ao diagrama apresentado pela primeira vez no Capítulo 4 (e reproduzido na Figura 5.1), este capítulo, junto ao Capítulo 6, continua nosso estudo dos conceitos de *middleware*, abordando a camada acima da comunicação entre processos. Em particular, as Seções 5.2 até 5.4 enfocam os estilos de comunicação listados anteriormente, e a Seção 5.5 apresenta um estudo de caso mais complexo, a RMI Java.

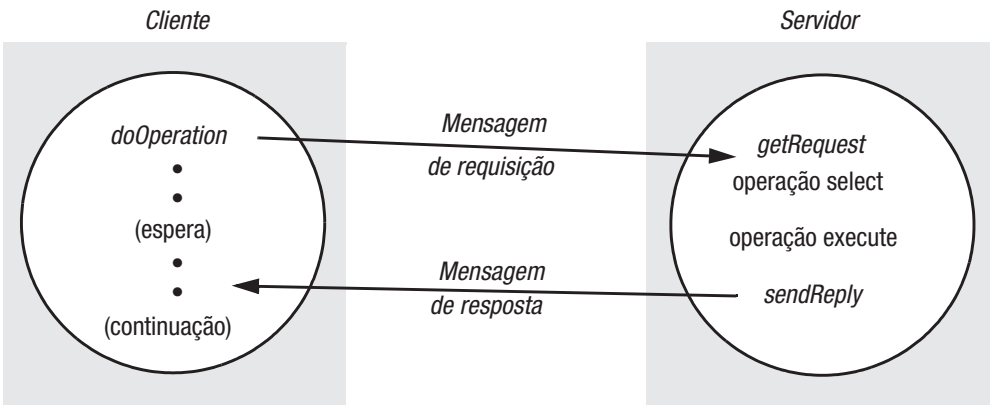

Figura 5.2 Comunicação por requisição-resposta.

5.2 Protocolos de requisição-resposta

Esta forma de comunicação é projetada para suportar as funções e as trocas de mensagens em interações cliente e servidor típicas. No caso normal, a comunicação por requisição-resposta é síncrona, pois o processo cliente é bloqueado até que a resposta do servidor chegue. Ela também pode ser confiável, pois a resposta do servidor é efetivamente um confirmação para o cliente. A comunicação por requisição-resposta assíncrona é uma alternativa útil em situações em que os clientes podem recuperar as respostas posteriormente – veja a Seção 7.5.2.

As trocas entre cliente e servidor estão descritas nos parágrafos a seguir, em termos das operações *send* e *receive* na API Java para datagramas UDP, embora muitas implementações atuais usem TCP. Um protocolo construído sobre datagramas evita sobrecargas desnecessárias associadas ao protocolo TCP. Em particular:

- As confirmações são redundantes, pois as requisições são seguidas por respostas.
- O estabelecimento de uma conexão envolve dois pares extras de mensagens, além do par exigido por uma requisição e uma resposta.
- O controle de fluxo é redundante para a maioria das invocações, que passam apenas pequenos argumentos e resultados.

O protocolo de requisição-resposta • O protocolo que descrevemos aqui é baseado em um trio de primitivas de comunicação: *doOperation*, *getRequest* e *sendReply*, como mostrado na Figura 5.2. Esse protocolo de requisição-resposta combina pedidos com respostas. Ele pode ser projetado para fornecer certas garantias de entrega. Se forem usados datagramas UDP, as garantias de entrega deverão ser fornecidas pelo protocolo de requisição-resposta, o qual pode usar a mensagem de resposta do servidor como confirmação da mensagem de requisição do cliente. A Figura 5.3 descreve, em linhas gerais, as três primitivas de comunicação.

O método *doOperation* é usado pelos clientes para invocar operações remotas. Seus argumentos especificam o servidor remoto e a operação a ser invocada, junto às informações adicionais (argumentos) exigidas pela operação. Seu resultado é um vetor de bytes contendo a resposta. Presume-se que o cliente que chama *doOperation* empacota os argumentos em um vetor de bytes e desempacota os resultados do vetor de bytes retornado. O primeiro argumento de *doOperation* é uma instância da classe *RemoteRef*, que representa

public byte[ ] doOperation (RemoteRef s, int operationId, byte[ ] arguments)
Envia uma mensagem de requisição para o servidor remoto e retorna a resposta. Os argumentos especificam o servidor remoto, a operação a ser invocada e os argumentos dessa operação.

public byte[ ] getRequest ( );
Lê uma requisição do cliente por meio da porta do servidor.

public void sendReply (byte[ ] reply, InetAddress clientHost, int clientPort);
Envia a mensagem de resposta *reply* para o cliente como seu endereço de Internet e porta.

Figura 5.3 Operações do protocolo de requisição-resposta.

referências para servidores remotos. Essa classe fornece métodos para obter o endereço de Internet e a porta do servidor associado. O método *doOperation* envia uma mensagem de requisição para o servidor, cujo endereço de Internet e porta são especificados na referência remota dada como argumento. Após enviar a mensagem de requisição, *doOperation* invoca *receive* para obter uma mensagem de resposta, a partir da qual extrai o resultado e o retorna para o chamador. O processo (cliente) que executa a operação *doOperation* é bloqueado até que o servidor execute a operação solicitada e transmita uma mensagem de resposta para ele.

getRequest é usado por um processo servidor para obter requisições de serviço, como mostrado na Figura 5.3. Quando o servidor tiver invocado a operação especificada, ele usa *sendReply* para enviar a mensagem de resposta ao cliente. Quando a mensagem de resposta é recebida pelo cliente, a operação *doOperation* original é desbloqueada, e a execução do programa cliente continua.

As informações a serem transmitidas em uma mensagem de requisição ou em uma mensagem de resposta aparecem na Figura 5.4. O primeiro campo indica se o tipo de mensagem (messageType) é *Request* ou *Reply*. O segundo campo, *requestId*, contém um identificador de mensagem. Um *doOperation* no cliente gera um *requestId* para cada mensagem de requisição, e o servidor copia esses identificadores nas mensagens de resposta correspondentes. Isso permite que *doOperation* verifique se uma mensagem de resposta é o resultado da requisição atual e não de uma chamada anterior atrasada. O terceiro campo é uma referência remota (remoteReference). O quarto campo é um identificador da operação (OperationId) a ser invocada. Por exemplo, as operações de uma interface poderiam ser numeradas como 1, 2, 3,...; se o cliente e o servidor usam uma linguagem comum que suporta reflexão, uma representação da operação em si pode ser colocada nesse campo.

messageType	*int (0=Request, 1=Reply)*
requestId	*int*
remoteReference	*RemoteRef*
OperationId	*int ou operação*
arguments	*// vetor de bytes*

Figura 5.4 Estrutura da mensagem de requisição-resposta.

Identificadores de mensagem • Qualquer esquema que envolva o gerenciamento de mensagens para fornecer propriedades adicionais, como entrega de mensagens confiável ou comunicação por requisição-resposta, exige que cada mensagem tenha um identificador exclusivo por meio do qual possa ser referenciada. Um identificador de mensagem consiste em duas partes:

1. um *requestId*, que é extraído pelo processo remetente a partir de uma sequência crescente de valores inteiros;
2. um identificador do processo remetente; por exemplo, sua porta e endereço de Internet.

A primeira parte torna o identificador exclusivo para o remetente e a segunda o torna exclusivo no sistema distribuído. (A segunda parte pode ser obtida independentemente – por exemplo, se UDP estiver em uso, a partir da mensagem recebida.)

Quando o valor de *requestId* atinge o máximo para um inteiro sem sinal (por exemplo, $2^{32} - 1$), ele volta a zero. A única restrição aqui é que o tempo de vida de um identificador de mensagem deve ser bem menor do que o tempo que leva para esgotar os valores na sequência de inteiros.

Modelo de falhas do protocolo de requisição-resposta • Se as três primitivas *doOperation*, *getRequest* e *sendReply* forem implementadas sobre datagramas UDP, elas sofrerão das mesmas falhas de comunicação. Ou seja:

- Sofrerão de falhas por omissão.
- Não há garantias de entrega das mensagens na ordem do envio.

Além disso, o protocolo pode sofrer uma falha de processos (veja a Seção 2.4.2). Supomos que os processos têm falhas de colapso, isto é, quando param, permanecem parados – e não produzem comportamento bizantino.

Para levar em conta as ocasiões em que um servidor falhou ou que uma mensagem de requisição ou resposta é perdida, *doOperation* usa um tempo limite (*timeout*) quando está esperando receber a mensagem de resposta do servidor. A ação executada quando ocorre um tempo limite depende das garantias de entrega oferecidas.

Tempos limite (timeouts) • Existem várias opções para o que *doOperation* deva fazer após esgotar um tempo limite (*timeout*). A opção mais simples é retornar imediatamente de *doOperation*, com uma indicação para o cliente de que *doOperation* falhou. Essa não é a estratégia mais comum – o tempo limite pode ter esgotado devido à perda da mensagem de requisição ou de resposta e, neste último caso, a operação foi executada. Para levar em conta a possibilidade de mensagens perdidas, *doOperation* envia a mensagem de requisição repetidamente, até receber uma resposta ou estar razoavelmente seguro de que o atraso se deve à falta de resposta do servidor e não à perda de mensagens. Finalmente, quando *doOperation* retornar, indicará isso para o cliente por meio de uma exceção, dizendo que nenhum resultado foi recebido.

Descarte de mensagens de requisição duplicadas • Nos casos em que a mensagem de requisição é retransmitida, o servidor pode recebê-la mais de uma vez. Por exemplo, o servidor pode receber a primeira mensagem de requisição, mas demorar mais do que o tempo limite do cliente para executar o comando e retornar a resposta. Isso pode levar o servidor a executar uma operação mais de uma vez para a mesma requisição. Para evitar isso, o protocolo é projetado de forma a reconhecer mensagens sucessivas (do mesmo cliente) com o mesmo identificador de requisição e eliminar as duplicatas. Se o servidor ainda não enviou a resposta, não precisa executar nenhuma ação especial – ele transmitirá a resposta quando tiver terminado de executar a operação.

Mensagens de resposta perdidas • Se o servidor já tiver enviado a resposta quando receber uma requisição duplicada, precisará executar a operação novamente para obter o resultado, a não ser que tenha armazenado o resultado da execução original. Alguns servidores podem executar suas operações mais de uma vez e obter os mesmos resultados. Uma *operação idempotente* é aquela que pode ser efetuada repetidamente com o mesmo efeito, como se tivesse sido executada exatamente uma vez. Por exemplo, uma operação para pôr um elemento em um conjunto é idempotente, pois sempre terá o mesmo efeito no conjunto, toda vez que for executada, enquanto uma operação para incluir um elemento em uma sequência não é idempotente, pois amplia a sequência sempre que é executada. Um servidor cujas operações são todas idempotentes não precisa adotar medidas especiais para evitar suas execuções mais de uma vez.

Histórico • Para os servidores que exigem retransmissão das respostas sem executar novamente as operações, pode-se usar um histórico. O termo *histórico* é usado para se referir a uma estrutura que contém um registro das mensagens (respostas) que foram transmitidas. Uma entrada em um histórico contém um identificador de requisição, uma mensagem e um identificador do cliente para o qual ela foi enviada. Seu objetivo é permitir que o servidor retransmita as mensagens de resposta quando os processos clientes as solicitarem. Um problema associado ao uso de um histórico é seu consumo de memória. Um histórico pode se tornar muito grande, a menos que o servidor possa identificar quando não há mais necessidade de retransmissão das mensagens.

Como os clientes só podem fazer uma requisição por vez, o servidor pode interpretar cada nova requisição como uma confirmação de sua resposta anterior. Portanto, o histórico precisa conter apenas a última mensagem de resposta enviada a cada cliente. Entretanto, o volume de mensagens de resposta no histórico de um servidor pode ser um problema quando ele tiver um grande número de clientes. Isso é combinado com o fato de que, quando um processo cliente termina, ele não confirma a última resposta recebida – portanto, as mensagens no histórico normalmente são descartadas após determinado período de tempo.

Estilos de protocolos de troca • Três protocolos, que produzem diferentes comportamentos na presença de falhas de comunicação, são usados para implementar vários tipos de comportamento de requisição. Eles foram identificados originalmente por Spector [1982]:

- o protocolo *request (R)*;
- o protocolo *request-reply (RR)*;
- o protocolo *request-reply-acknowledge reply (RRA)*.

As mensagens passadas nesses protocolos estão resumidas na Figura 5.5. No protocolo R, uma única mensagem *Request* é enviada pelo cliente para o servidor. O protocolo R pode ser usado quando não existe nenhum valor a ser retornado do método remoto e o cliente não exige confirmação de que a operação foi executada. O cliente pode prosseguir imediatamente após a mensagem de requisição ser enviada, pois não há necessidade de esperar por uma mensagem de resposta. Esse protocolo é implementado sobre datagramas UDP e, portanto, sofre das mesmas falhas de comunicação.

O protocolo RR é útil para a maioria das trocas cliente-servidor, pois é baseado no protocolo de requisição-resposta. Não são exigidas mensagens de confirmação especiais, pois uma mensagem de resposta (*reply*) do servidor é considerada como confirmação do recebimento da mensagem de requisição (*request*) do cliente. Analogamente, uma chamada subsequente de um cliente pode ser considerada como uma confirmação da mensagem de resposta de um servidor. Conforme vimos anteriormente, as falhas de

Nome	*Mensagens enviadas pelo*		
	Cliente	*Servidor*	*Cliente*
R	*Requisição*		
RR	*Requisição*	*Resposta*	
RRA	*Requisição*	*Resposta*	*Resposta de confirmação*

Figura 5.5 Protocolos RPC.

comunicação ocasionadas pela perda de datagramas UDP podem ser mascaradas pela retransmissão das requisições com filtragem duplicada e pelo registro das respostas em um histórico para retransmissão.

O protocolo RRA é baseado na troca de três mensagens: requisição, resposta e confirmação. A mensagem de *confirmação* contém o *requestId* da mensagem de resposta que está sendo confirmada. Isso permitirá que o servidor descarte entradas de seu histórico. A chegada de um *requestId* em uma mensagem de confirmação será interpretada como a acusação do recebimento de todas as mensagens de resposta com valores de *requestId* menores; portanto, a perda de uma mensagem de confirmação não é muito prejudicial ao sistema. Embora o protocolo RRA envolva uma mensagem adicional, ela não precisa bloquear o cliente, pois a confirmação pode ser transmitida após a resposta ter sido entregue ao cliente. Contudo, ela utiliza recursos de processamento e rede. O Exercício 5.10 sugere uma otimização para o protocolo RRA.

Uso de TCP para implementar o protocolo de requisição-resposta • A seção 4.2.3 mencionou que frequentemente é difícil decidir-se sobre um tamanho apropriado para o *buffer* de recebimento de datagramas. No protocolo de requisição-resposta, isso se aplica aos *buffers* usados pelo servidor para receber mensagens de requisição e pelo cliente para receber respostas. O comprimento limitado dos datagramas (normalmente, 8 quilobytes) pode não ser considerado adequado para uso em sistemas de RMI ou RPC transparentes, pois os argumentos ou resultados dos procedimentos podem ser de qualquer tamanho.

O desejo de evitar a implementação de protocolos que tratem de requisições e respostas em múltiplos pacotes é um dos motivos para se escolher a implementação de protocolos de requisição-resposta com TCP, permitindo a transmissão de argumentos e resultados de qualquer tamanho. Em particular, a serialização de objetos Java é um protocolo que permite o envio de argumentos e resultados por meio de fluxos entre cliente e servidor, tornando possível que conjuntos de objetos de qualquer tamanho sejam transmitidos de maneira confiável. Se o protocolo TCP for usado, isso garantirá que as mensagens de requisição e de resposta sejam entregues de modo confiável; portanto, não há necessidade de o protocolo de requisição-resposta tratar da retransmissão de mensagens, filtrar duplicatas ou lidar com históricos. Além disso, o mecanismo de controle de fluxo permite que argumentos e resultados grandes sejam passados, sem se tomar medidas especiais para não sobrecarregar o destinatário. Assim, a escolha do protocolo TCP simplifica a implementação de protocolos de requisição-resposta. Se sucessivas requisições e respostas entre o mesmo par cliente-servidor são enviadas por um mesmo fluxo, a sobrecarga de conexão necessária não se aplica a toda invocação remota. Além disso, a sobrecarga em razão das mensagens de confirmação TCP é reduzida quando uma mensagem de resposta é gerada logo após a mensagem de requisição.

Entretanto, se o aplicativo não exige todos os recursos oferecidos pelo TCP, um protocolo mais eficiente, especialmente personalizado, pode ser implementado sobre

UDP. Por exemplo, o NFS da Sun não exige suporte para mensagens de tamanho ilimitado, pois transmite blocos de arquivo de tamanho fixo entre o cliente e o servidor. Além disso, suas operações são projetadas para serem idempotentes, de modo que não importa se as operações são executadas mais de uma vez para retransmitir mensagens de resposta perdidas, tornando desnecessário manter um histórico.

HTTP: um exemplo de protocolo de requisição-resposta • O Capítulo 1 apresentou o protocolo HTTP (HyperText Transfer Protocol), usado pelos navegadores Web para fazer pedidos para servidores Web e receber suas respostas. Para recapitular, os servidores Web gerenciam recursos implementados de diferentes maneiras:

- como dados – por exemplo, o texto de uma página em HTML, uma imagem ou a classe de um *applet*;
- como um programa – por exemplo, *servlets* [java.sun.com III] ou programas em PHP ou Python, que podem ser executados no servidor Web.

As requisições do cliente especificam um URL que inclui o nome DNS de um servidor Web e um número de porta opcional no servidor Web, assim como o identificador de um recurso nesse servidor.

O protocolo HTTP especifica as mensagens envolvidas em uma troca de requisição-resposta, os métodos, os argumentos, os resultados e as regras para representá-los (empacotá-los) nas mensagens. Ele suporta um conjunto fixo de métodos (*GET*, *PUT*, *POST,* etc.) que são aplicáveis a todos os seus recursos. Ele é diferente dos protocolos descritos anteriormente, em que cada serviço tem seu próprio conjunto de operações. Além de invocar métodos sobre recursos Web, o protocolo permite negociação de conteúdo e autenticação com senha:

> *Negociação de conteúdo:* as requisições dos clientes podem incluir informações sobre qual representação de dados elas podem aceitar (por exemplo, linguagem ou tipo de mídia), permitindo que o servidor escolha a representação mais apropriada para o usuário.
>
> *Autenticação:* credenciais e desafios (*challenges*) são usados para suportar autenticação com senha. Na primeira tentativa de acessar uma área protegida com senha, a resposta do servidor contém um desafio aplicável ao recurso. O Capítulo 11 explicará os desafios. Quando um cliente recebe um desafio, obriga o usuário a digitar um nome e uma senha e envia as credenciais associadas às requisições subsequentes.

O protocolo HTTP é implementado sobre TCP. Na versão original do protocolo, cada interação cliente-servidor consiste nas seguintes etapas:

- O cliente solicita uma conexão com o servidor na porta HTTP padrão ou em uma porta especificada no URL.
- O cliente envia uma mensagem de requisição para o servidor.
- O servidor envia uma mensagem de resposta para o cliente.
- A conexão é fechada.

Entretanto, estabelecer e fechar uma conexão para cada troca de requisição-resposta é dispendioso, sobrecarregando o servidor e causando o envio de muitas mensagens pela rede. Lembremos que os navegadores geralmente fazem vários pedidos para o mesmo servidor – por exemplo, para obter uma imagem de uma página recentemente fornecida – uma versão posterior do protocolo (HTTP 1.1, veja RFC 2616 [Fielding *et al.* 1999]) usa *conexões persistentes* – conexões que permanecem abertas durante uma sequência de

método	*URL ou nome de caminho*	*versão de HTTP*	*cabeçalhos*	*corpo da mensagem*
GET	http://www.dcs.qmul.ac.uk/index.html	HTTP/ 1.1		

Figura 5.6 Mensagem de requisição HTTP.

trocas de requisição-resposta entre cliente e servidor. Uma conexão persistente pode ser encerrada a qualquer momento, tanto pelo cliente como pelo servidor, pelo envio de uma indicação para o outro participante. Os servidores encerrarão uma conexão persistente quando ela estiver ociosa por determinado período de tempo. É possível que um cliente receba uma mensagem do servidor dizendo que a conexão está encerrada no meio do envio de outra requisição (ou requisições). Neste caso, o navegador enviará novamente as requisições, sem envolvimento do usuário, desde que as operações envolvidas sejam idempotentes. Por exemplo, o método *GET,* descrito a seguir, é idempotente. Onde estão envolvidas operações não idempotentes, o navegador deve consultar o usuário para saber o que fazer em seguida.

As requisições e respostas são empacotadas nas mensagens como *strings* de texto ASCII, mas os recursos podem ser representados como sequências de bytes e podem ser compactados. A utilização de texto na representação externa de dados simplificou o uso de HTTP pelos programadores de aplicativos que trabalham diretamente com o protocolo. Neste contexto, uma representação textual não aumenta muito o comprimento das mensagens.

Os recursos implementados como dados são fornecidos como estruturas do tipo MIME em argumentos e resultados. MIME (Multipurpose Internet Mail Extensions) é um padrão para envio de dados de múltiplas partes, especificado no RFC 2045 [Freed e Borenstein 1996], contendo, por exemplo, texto, imagens e som em mensagens de correio eletrônico. Os dados são prefixados com seu *tipo MIME* para que o destinatário saiba como manipulá-los. Um tipo MIME especifica um tipo e um subtipo, por exemplo, *text/plain*, *text/html*, *image/gif* ou *image/jpeg*. Os clientes também podem especificar os tipos MIME que desejam aceitar.

Métodos HTTP • Cada requisição de cliente especifica o nome de um método a ser aplicado em um recurso no servidor e o URL desse recurso. A resposta fornece o *status* da requisição. As mensagens de requisição-resposta também podem conter dados de recurso, o conteúdo de um formulário ou a saída de um programa executado no servidor Web. Os métodos incluem o seguinte:

GET: solicita o recurso cujo URL é dado como argumento. Se o URL se referir a dados, o servidor Web responderá retornando os dados identificados por esse URL. Se o URL se referir a um programa, então o servidor Web executará o programa e retornará sua saída para o cliente. Argumentos podem ser adicionados no URL; por exemplo, o método *GET* pode ser usado para enviar o conteúdo de um formulário como argumento para um programa. A operação *GET* pode ser condicionada à data em que um recurso foi modificado pela última vez. *GET* também pode ser configurado para obter partes dos dados.

Com *GET*, toda a informação da requisição é fornecida no URL (veja, por exemplo, a *string* de consulta da Seção 1.6).

HEAD: esta requisição é idêntica a *GET*, mas não retorna nenhum dado. Entretanto, retorna todas as informações sobre os dados, como a hora da última modificação, seu tipo ou seu tamanho.

POST: especifica o URL de um recurso (por exemplo, um programa) que pode tratar dos dados fornecidos no corpo do pedido. O processamento executado nos dados depende da função do programa especificado no URL. Esse método é feito para lidar com:

- o fornecimento de um bloco de dados para um processo de manipulação de dados, como um *servlet* – por exemplo, enviando um formulário Web para comprar algo em um *site*;
- o envio de uma mensagem para uma lista de distribuição ou da atualização de detalhes de membros da lista;
- a ampliação de um banco de dados com uma operação *append*.

PUT: solicita que os dados fornecidos na requisição sejam armazenados no URL informado, como uma modificação de um recurso já existente ou como um novo recurso.

DELETE: o servidor exclui o recurso identificado pelo URL fornecido. Nem sempre os servidores permitem essa operação; nesse caso, a resposta indicará a falha.

OPTIONS: o servidor fornece ao cliente uma lista de métodos que podem ser aplicados no URL dado (por exemplo, *GET*, *HEAD*, *PUT*) e seus requisitos especiais.

TRACE: o servidor envia de volta a mensagem de requisição. Usado para propósitos de diagnóstico.

As operações *PUT* e *DELETE* são idempotentes, mas *POST* não necessariamente o é, pois pode alterar o estado de um recurso. As outras são operações *seguras*, pois não alteram nada.

As requisições descritas anteriormente podem ser interceptadas por um servidor *proxy* (veja a Seção 2.3.1). As respostas de *GET* e *HEAD* podem ser armazenadas em cache por servidores *proxy*.

Conteúdo da mensagem • A mensagem de requisição especifica o nome de um método, o URL de um recurso, a versão do protocolo, alguns cabeçalhos e um corpo de mensagem opcional. A Figura 5.6 mostra o conteúdo de uma mensagem de requisição HTTP cujo método é *GET*. Quando o URL especifica um recurso de dados, o método *GET* não tem corpo de mensagem.

As requisições para servidores *proxies* precisam do URL absoluto, como mostrado na Figura 5.6. As requisições para os servidores de origem (aqueles onde fica o recurso) especificam um nome de caminho e fornecem o nome DNS do servidor no campo de cabeçalho *Host*. Por exemplo,

GET /index.html HTTP/1.1
Host: www.dcs.qmul.ac.uk

Em geral, os campos de cabeçalho contêm modificadores de requisição e informações do cliente, como as condições do recurso na data de modificação mais recente ou os tipos de conteúdo aceitáveis (por exemplo, texto HTML, áudio ou imagens JPEG). Um campo de autorização pode ser usado para fornecer as credenciais do cliente, na forma de um certificado, definindo seus direitos de acesso a um recurso.

Uma mensagem de resposta possui a versão do protocolo, um código de *status* e um "motivo", alguns cabeçalhos e um corpo de mensagem opcional, como mostrado na Figura 5.7. O código de *status* e o motivo fornecem um relato sobre o sucesso ou não na execução da requisição: o primeiro é um valor inteiro de três dígitos para interpretação por um programa e o último é uma frase textual que pode ser entendida por uma pessoa. Os campos de ca-

versão de HTTP	*código de status*	*motivo*	*cabeçalhos*	*corpo da mensagem*
HTTP/1.1	200	OK		dados de recurso

Figura 5.7 Mensagem de resposta HTTP.

beçalho são usados para passar informações adicionais sobre o servidor ou dados relativos ao acesso ao recurso. Por exemplo, se a requisição exige autenticação, o *status* da resposta indica isso e um campo de cabeçalho contém um desafio. Alguns *status* de retorno têm efeitos bastante complexos. Em particular, a resposta de código de *status* 303 diz ao navegador para que examine um URL diferente, o qual é fornecido em um campo de cabeçalho na resposta. Ela é usada em uma resposta de um programa executado por meio de uma requisição *POST*, quando esse programa precisar redirecionar o navegador para um recurso selecionado.

O corpo das mensagens de requisição ou de resposta contém os dados associados ao URL especificado na requisição. O corpo da mensagem possui seus próprios cabeçalhos, especificando informações sobre os dados, como seu comprimento, tipo MIME, conjunto de caracteres, codificação do conteúdo e a data da última modificação. O campo de tipo MIME determina o tipo dos dados, por exemplo *image/jpeg* ou *text/plain*. O campo de codificação do conteúdo especifica o algoritmo de compactação a ser usado.

5.3 Chamada de procedimento remoto

Conforme mencionado no Capítulo 2, o conceito de chamada de procedimento remoto (RPC) representa um importante avanço intelectual na computação distribuída, com o objetivo de tornar a programação de sistemas distribuídos semelhante (se não idêntica) à programação convencional – isto é, obter transparência de distribuição de alto nível. Essa unificação é conseguida de maneira muito simples, estendendo a abstração de chamada de procedimento para os ambientes distribuídos. Em particular, na RPC, os procedimentos em máquinas remotas podem ser chamados como se fossem procedimentos no espaço de endereçamento local. Então, o sistema RPC oculta os aspectos importantes da distribuição, incluindo a codificação e a decodificação de parâmetros e resultados, a passagem de mensagens e a preservação da semântica exigida para a chamada de procedimento. Esse conceito foi apresentado pela primeira vez por Birrell e Nelson [1984] e abriu o caminho para muitos dos desenvolvimentos na programação de sistemas distribuídos utilizados atualmente.

5.3.1 Questões de projeto para RPC

Antes de examinarmos a implementação de sistemas de RPC, vamos ver três questões importantes para se entender esse conceito:

- o estilo de programação promovido pela RPC – programação com interfaces;
- a semântica de chamada associada à RPC;
- o problema da transparência e como ele se relaciona com as chamadas de procedimento remoto.

Programação com interfaces • A maioria das linguagens de programação modernas fornece uma maneira de organizar um programa como um conjunto de módulos que podem se comunicar. A comunicação entre os módulos pode ser feita por meio de chamadas

de procedimento entre eles ou pelo acesso direto às variáveis de outro módulo. Para controlar as possíveis interações entre os módulos, é definida uma *interface* explícita para cada módulo. A interface de um módulo especifica os procedimentos e as variáveis que podem ser acessadas a partir de outros módulos. Os módulos são implementados de forma a ocultar todas as informações sobre eles, exceto o que está disponível por meio de sua interface. Contanto que sua interface permaneça a mesma, a implementação pode ser alterada sem afetar os usuários do módulo.

Interfaces em sistemas distribuídos: em um programa distribuído, os módulos podem ser executados em processos distintos. No modelo cliente-servidor, em particular, cada servidor fornece um conjunto de procedimentos que estão disponíveis para uso pelos clientes. Por exemplo, um servidor de arquivos forneceria procedimentos para ler e escrever em arquivos. O termo *interface de serviço* é usado para se referir à especificação dos procedimentos oferecidos por um servidor, definindo os tipos dos argumentos de cada um dos procedimentos.

Há várias vantagens na programação com interfaces nos sistemas distribuídos, resultantes da importante separação entre interface e implementação:

- Assim como acontece com qualquer forma de programação modular, os programadores só se preocupam com a abstração oferecida pela interface de serviço e não precisam conhecer os detalhes da implementação.
- Extrapolando para sistemas distribuídos (potencialmente heterogêneos), os programadores também não precisam conhecer a linguagem de programação nem a plataforma de base utilizadas para implementar o serviço (um avanço importante no gerenciamento da heterogeneidade em sistemas distribuídos).
- Essa estratégia fornece suporte natural para a evolução de *software*, pois as implementações podem mudar, desde que a interface (a visão externa) permaneça a mesma. Mais corretamente, a interface também pode mudar, contanto que permaneça compatível com a original.

A definição de interfaces de serviço é influenciada pela natureza distribuída da infraestrutura subjacente:

- Não é possível um módulo ser executado em um processo e acessar as variáveis de um módulo em outro processo. Portanto, a interface de serviço não pode especificar acesso direto às variáveis. Note que as interfaces IDL do CORBA podem especificar atributos, o que parece violar essa regra. Entretanto, os atributos não são acessados diretamente, mas por meio de alguns procedimentos de obtenção e atribuição (*getter* e *setter*), adicionados automaticamente na interface.
- Os mecanismos de passagem de parâmetros usados nas chamadas de procedimento local – por exemplo, chamada por valor e chamada por referência – não são convenientes quando o chamador e o procedimento estão em processos diferentes. Em particular, a chamada por referência não é suportada. Em vez disso, a especificação de um procedimento na interface de um módulo em um programa distribuído descreve os parâmetros como sendo de *entrada* ou de *saída* ou, às vezes, ambos. Os parâmetros de *entrada* são passados para o servidor remoto pelo envio dos valores dos argumentos na mensagem de requisição e, então, repassados como argumentos para a operação a ser executada no servidor. Os parâmetros de *saída* são retornados na mensagem de resposta e são usados como resultado da chamada ou para substi-

```
// Arquivo de entrada Person.idl
struct Person {
        string name;
        string place;
        long year;
};
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p) ;
        void getPerson(in string name, out Person p);
        long number( );
};
```

Figura 5.8 Exemplo de IDL do CORBA.

tuir os valores das variáveis correspondentes no ambiente de chamada. Quando um parâmetro é usado tanto para entrada como para saída, o valor deve ser transmitido nas mensagens de requisição e de resposta.

- Outra diferença entre módulos locais e remotos é que os endereços de um processo não são válidos em outro processo remoto. Portanto, os endereços não podem ser passados como argumentos nem retornados como resultado de chamadas para módulos remotos.

Essas restrições têm um impacto significativo na especificação de linguagens de definição de interface, conforme discutido a seguir.

Linguagens de definição de interface: um mecanismo de RPC pode ser integrado a uma linguagem de programação em particular, caso inclua uma notação adequada para definir interfaces, permitindo que os parâmetros de entrada e saída sejam mapeados no uso normal de parâmetros da linguagem. Essa estratégia é útil quando todas as partes de um aplicativo distribuído podem ser escritas na mesma linguagem. Ela também é conveniente, pois permite que o programador utilize uma única linguagem, por exemplo, Java, para invocação local e remota.

Entretanto, muitos serviços úteis existentes são escritos em C++ e em outras linguagens. Seria vantajoso permitir que programas escritos em uma variedade de linguagens, incluindo Java, acessassem-nos de forma remota. As *linguagens de definição de interface* (ou IDLs) são projetadas para permitir que procedimentos implementados em diferentes linguagens invoquem uns aos outros. Uma IDL fornece uma notação para definir interfaces, na qual cada um dos parâmetros de uma operação pode ser descrito como sendo de *entrada* ou de *saída*, além de ter seu tipo especificado.

A Figura 5.8 mostra um exemplo simples de IDL do CORBA. A estrutura *Person* é a mesma usada para ilustrar o empacotamento na Seção 4.3.1. A interface chamada *PersonList* especifica os métodos disponíveis para RMI em um objeto remoto que implementa essa interface. Por exemplo, o método *addPerson* especifica seu argumento como *in*, significando que se trata de um argumento de *entrada* (*input*); e o método *getPerson*, que recupera uma instância de *Person* pelo nome, especifica seu segundo argumento como *out*, significando que se trata de um argumento de *saída* (*output*).

Medidas de tolerância a falhas			*Semântica de chamada*
Reenvio da mensagem de requisição	*Filtragem de duplicatas*	*Reexecução de procedimento ou retransmissão da resposta*	
Não	Não aplicável	Não aplicável	*Talvez*
Sim	Não	Executa o procedimento novamente	*Pelo menos uma vez*
Sim	Sim	Retransmite a resposta	*No máximo uma vez*

Figura 5.9 Semânticas de chamada.

O conceito de IDL foi desenvolvido inicialmente para sistemas RPC, mas se aplica igualmente à RMI e também a serviços Web. Nossos estudos de caso incluem:

- XDR da Sun como exemplo de IDL para RPC (na Seção 5.3.3);
- IDL do CORBA como exemplo de IDL para RMI (no Capítulo 8 e também incluído anteriormente);
- a linguagem de descrição de serviços Web (WSDL, Web Services Description Language), que é projetada para uma RPC na Internet que suporte serviços Web (veja a Seção 9.3);
- e *buffers* de protocolo utilizados no Google para armazenar e trocar muitos tipos de informações estruturadas (veja a Seção 21.4.1).

Semântica de chamada RPC • Os protocolos de requisição-resposta foram discutidos na Seção 5.2, em que mostramos que a operação *doOperation* pode ser implementada de várias maneiras para fornecer diferentes garantias de entrega. As principais escolhas são:

Retentativa de mensagem de requisição: para retransmitir a mensagem de requisição até que uma resposta seja recebida ou que se presuma que o servidor falhou.

Filtragem de duplicatas: quando são usadas retransmissões para eliminar requisições duplicadas no servidor.

Retransmissão de resultados: para manter um histórico das mensagens de respostas a fim de permitir que os resultados perdidos sejam retransmitidos sem uma nova execução das operações no servidor.

Combinações dessas escolhas levam a uma variedade de semânticas possíveis para a confiabilidade das invocações remotas vistas pelo ativador. A Figura 5.9 mostra as escolhas de interesse, com os nomes correspondentes da semântica que produzem. Note que, para chamadas de procedimentos locais, a semântica é *exatamente uma vez*, significando que todo procedimento é executado exatamente uma vez (exceto no caso de falha de processo). As escolhas de semântica de invocação RPC estão definidas a seguir.

Semântica talvez: com a semântica *talvez*, a chamada de procedimento remoto pode ser executada uma vez ou não ser executada. A semântica *talvez* surge quando nenhuma das medidas de tolerância a falhas é aplicada e pode sofrer os seguintes tipos de falha:

- falhas por omissão, se a mensagem de requisição ou de resultado for perdida;
- falhas por colapso, quando o servidor que contém a operação remota falha.

Se a mensagem de resultado não tiver sido recebida após um tempo limite e não houver novas tentativas, não haverá certeza se o procedimento foi executado. Se a mensagem de requisição foi perdida, então o procedimento não foi executado. Por outro lado, o procedimento pode ter sido executado e a mensagem de resultado, perdida. Uma falha por colapso pode ocorrer antes ou depois do procedimento ser executado. Além disso, em um sistema assíncrono, o resultado da execução do procedimento pode chegar após o tempo limite. A semântica *talvez* é útil apenas para aplicações nas quais são aceitáveis chamadas mal-sucedidas ocasionais.

Semântica pelo menos uma vez: com a semântica *pelo menos uma vez*, o invocador recebe um resultado (no caso em que sabe que o procedimento foi executado pelo menos uma vez) ou recebe uma exceção, informando-o de que nenhum resultado foi recebido. A semântica *pelo menos uma vez* pode ser obtida pela retransmissão das mensagens de requisição, o que mascara as falhas por omissão da mensagem de requisição ou de resultado. A semântica *pelo menos uma vez* pode sofrer dos seguintes tipos de falha:

- falhas por colapso, quando o servidor que contém o procedimento remoto falha;
- falhas arbitrárias – nos casos em que a mensagem de requisição é retransmitida, o servidor remoto pode recebê-la e executar o procedimento mais de uma vez, possivelmente causando o armazenamento ou o retorno de valores errados.

A Seção 5.2 definiu uma *operação idempotente* como aquela que pode ser executada repetidamente, com o mesmo efeito de que se tivesse sido executada exatamente uma vez. As operações não idempotentes podem ter o efeito errado se forem executadas mais de uma vez. Por exemplo, uma operação para aumentar um saldo bancário em $10 deve ser executada apenas uma vez; se ela fosse repetida, o saldo aumentaria e aumentaria! Se as operações em um servidor podem ser projetadas de modo que todos os procedimentos em suas interfaces remotas sejam operações idempotentes, então a semântica *pelo menos uma vez* pode ser aceitável.

Semântica no máximo uma vez: com a semântica *no máximo uma vez*, ou o chamador recebe um resultado – no caso em que o chamador sabe que o procedimento foi executado exatamente uma vez – ou recebe uma exceção informando-o de que nenhum resultado foi recebido – no caso em que o procedimento terá sido executado uma vez ou não terá sido executado. A semântica *no máximo uma vez* pode ser obtida pelo uso de todas as medidas de tolerância a falhas resumidas na Figura 5.9. Como no caso anterior, o emprego de retentativas mascara as falhas por omissão das mensagens de requisição ou de resultado. Esse conjunto de medidas de tolerância a falhas evita falhas arbitrárias garantindo que, para cada RPC, um procedimento nunca seja executado mais de uma vez. A RPC da Sun (discutida na Seção 5.3.3) fornece semântica de chamada *pelo menos uma vez*.

Transparência • Os criadores da RPC, Birrell e Nelson [1984], pretendiam tornar as chamadas de procedimentos remotos o mais parecidas possível com as chamadas de procedimentos locais, sem nenhuma distinção na sintaxe entre uma chamada de procedimento local e um remoto. Todas as chamadas necessárias para procedimentos de empacotamento e trocas de mensagens foram ocultadas do programador que faz a chamada. Embora as mensagens de requisição sejam retransmitidas após um tempo limite, isso é transparente para o chamador, a fim de tornar a semântica das chamadas de procedimento remoto iguais às das chamadas de procedimento local.

Mais precisamente, voltando à terminologia do Capítulo 1, a RPC tenta oferecer pelo menos transparência de localização e de acesso, ocultando o local físico do procedi-

mento (potencialmente remoto) e também acessando procedimentos locais e remotos da mesma maneira. O *middleware* também pode oferecer níveis de transparência adicionais para RPC.

Entretanto, as chamadas de procedimento remoto são mais vulneráveis às falhas do que as locais, pois envolvem uma rede, outro computador e outro processo. Qualquer que seja a semântica escolhida, há sempre a chance de que nenhum resultado seja recebido e, em caso de falha, é impossível distinguir entre falha da rede e do processo servidor remoto. Isso exige que os clientes que estão fazendo chamadas remotas possam se recuperar de tais situações.

A latência de um procedimento remoto é muito maior do que a de um local. Isso sugere que os programas que utilizam chamadas remotas precisam levar esse fator em consideração, talvez minimizando as interações remotas. Os projetistas do Argus [Liskov e Scheifler 1982] sugeriram que deveria ser possível para um chamador cancelar uma chamada de procedimento remoto que esteja demorando demais, de tal maneira que isso não tenha efeito sobre o servidor. Para possibilitar isso, o servidor precisaria restaurar as variáveis para o estado em que estavam antes que o procedimento fosse chamado. Esses problemas serão discutidos no Capítulo 16.

As chamadas de procedimento remoto também exigem um estilo de passagem de parâmetros diferente, conforme discutido anteriormente. Em particular, a RPC não oferece chamada por referência.

Waldo *et al.* [1994] dizem que a diferença entre operações locais e remotas deve ser expressa na interface de serviço, para permitir que os participantes reajam de maneira consistente às possíveis falhas parciais. Outros sistemas foram ainda mais longe, argumentando que a sintaxe de uma chamada remota deve ser diferente da de uma chamada local – no caso do Argus, a linguagem foi ampliada para tornar as operações remotas explícitas para o programador.

A escolha quanto ao fato de a RPC ser transparente também está disponível para os projetistas de IDLs. Por exemplo, em algumas IDLs, uma invocação remota pode lançar uma exceção quando o cliente não consegue se comunicar com um procedimento remoto. Isso exige que o programa cliente trate de tais exceções, permitindo que ele lide com essas falhas. Uma IDL também pode fornecer um recurso para especificar a semântica de chamada de um procedimento. Isso pode ajudar o projetista do serviço – por exemplo, se a semântica *pelo menos uma vez* for escolhida para evitar as sobrecargas da semântica *no máximo uma vez*, as operações deverão ser projetadas para serem idempotentes.

O consenso atual é o de que as chamadas remotas devem se tornar transparentes, no sentido de a sintaxe de uma chamada remota ser a mesma de uma chamada local, mas a diferença entre chamadas locais e remotas deve ser expressa em suas interfaces.

5.3.2 Implementação de RPC

Os componentes de *software* exigidos para implementar RPC estão mostrados na Figura 5.10. O cliente que acessa um serviço inclui um *procedimento stub* para cada procedimento da interface de serviço. O procedimento *stub* se comporta como um procedimento local para o cliente, mas em vez de executar a chamada, ele empacota o identificador de procedimento e os argumentos em uma mensagem de requisição, a qual envia para o servidor por meio de seu módulo de comunicação. Quando a mensagem de resposta chega, ele desempacota os resultados. O processo servidor contém um despachante junto a um procedimento *stub* de servidor e um procedimento de serviço para cada procedimento na interface de serviço. O despachante seleciona um dos procedimentos *stub* de servidor, de

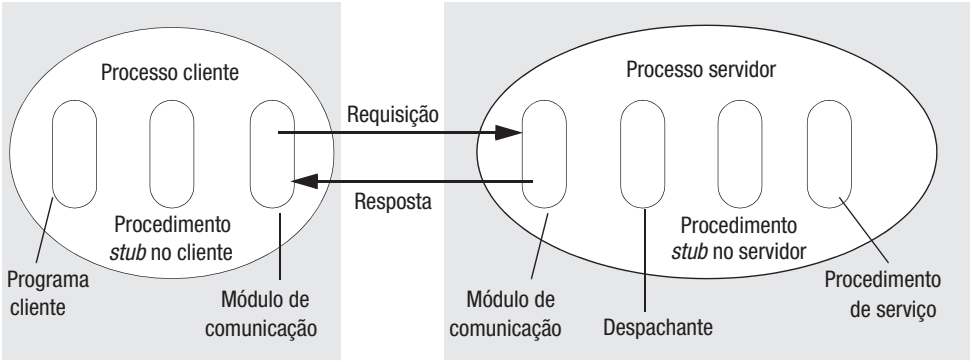

Figura 5.10 Função dos procedimentos *stub* no cliente e no servidor na RPC.

acordo com o identificador de procedimento presente na mensagem de requisição. Então, o procedimento *stub* de servidor desempacota os argumentos presentes na mensagem de requisição, chama o procedimento de serviço correspondente e empacota os valores de retorno para a mensagem de resposta. Os procedimentos de serviço implementam os procedimentos da interface de serviço. Os procedimentos *stub* do cliente e do servidor e o despachante podem ser gerados automaticamente por um compilador de interface, a partir da definição de interface do serviço.

Geralmente, a RPC é implementada sobre um protocolo de requisição-resposta, como o discutido na Seção 5.2. O conteúdo das mensagens de requisição e de resposta é igual ao ilustrado para os protocolos de requisição-resposta na Figura 5.4. A RPC pode ser implementada de modo a ter uma das escolhas de semântica de invocação discutidas na Seção 5.3.1 – *pelo menos uma vez* ou *no máximo uma vez* são geralmente escolhidas. Para conseguir isso, o módulo de comunicação implementará as escolhas de projeto desejadas, em termos das retransmissões de requisições, tratando de duplicatas e da retransmissão de resultados, como mostrado na Figura 5.9.

5.3.3 Estudo de caso: RPC da Sun

A RFC 1831 [Srinivasan 1995a] descreve a RPC da Sun, que foi projetada para comunicação cliente-servidor no sistema de arquivos de rede NFS (Network File System). A RPC da Sun é também chamada de RPC ONC (Open Network Computing). Ela é fornecida como parte dos vários sistemas operacionais da Sun e em outros, derivados do UNIX, estando também disponível em instalações de NFS. Os implementadores têm a opção de usar chamadas de procedimento remoto sobre UDP ou TCP. Quando a RPC da Sun é usada com UDP, o comprimento das mensagens de requisição e de resposta é limitado – teoricamente, em 64 quilobytes –, mas, na prática, mais frequentemente, em 8 ou 9 quilobytes. A semântica usada é a *pelo menos uma vez*. Existe a opção de usar *broadcast* de RPC.

O sistema RPC da Sun fornece uma linguagem de interface chamada XDR (External Data Representation) e um compilador de interface chamado *rpcgen*, destinado a ser usado com a linguagem de programação C.

Linguagem de definição de interface • A linguagem XDR da Sun, originalmente projetada para especificar representações externas de dados, foi estendida para se tornar uma linguagem de definição de interface. Ela pode ser usada para definir uma interface de serviço para RPC, por meio da especificação de um conjunto de definições de procedi-

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
        int length;
        char buffer[MAX];
};
struct writeargs {
        FileIdentifier f;
        FilePointer position;
        Data data;
};
struct readargs {
        FileIdentifier f;
        FilePointer position;
        Length length;
};

program FILEREADWRITE {
      version VERSION {
            void WRITE (writeargs) = 1;                    1
            Data READ (readargs) = 2;                      2
      }=2;
} = 9999;
```

Figura 5.11 Interface de arquivos na XDR da Sun.

mentos, junto a suporte para definições de tipo. A notação é bastante primitiva em comparação com a usada pela IDL do CORBA ou Java. Em particular:

- A maioria das linguagens permite a especificação de nomes de interface, mas a RPC da Sun não permite – em vez disso, são fornecidos um número de programa e um número de versão. Os números de programa podem ser obtidos a partir de uma autoridade central, para permitir que cada programa tenha seu próprio número exclusivo. O número de versão muda quando uma assinatura de procedimento é alterada. Tanto o número de programa como o número de versão são passados na mensagem de requisição para que o cliente e o servidor possam verificar se estão usando a mesma versão.
- A definição de um procedimento especifica uma assinatura e um número. O número do procedimento é usado como identificador nas mensagens de requisição.
- É permitido apenas um parâmetro de entrada. Portanto, os procedimentos que necessitam de vários parâmetros devem incluí-los como componentes de uma única estrutura.
- Os parâmetros de saída de um procedimento são retornados por meio de um único resultado.
- A assinatura de um procedimento consiste no tipo do resultado, no nome do procedimento e no tipo do parâmetro de entrada. O tipo do resultado e do parâmetro de entrada pode especificar um único valor ou uma estrutura contendo vários valores.

Por exemplo, veja a definição de XDR, na Figura 5.11, de uma interface com dois procedimentos para escrever e ler arquivos. O número do programa é 9999 e o número da versão é 2. O procedimento *READ* (linha 2) recebe como parâmetro de entrada uma estrutura com três componentes, especificando um identificador de arquivo, uma posição no arquivo e o número de bytes a serem lidos. Seu resultado é uma estrutura contendo o número de bytes retornados e os dados do arquivo. O procedimento *WRITE* (linha 1) não tem nenhum resultado. Os procedimentos *WRITE* e *READ* recebem os números 1 e 2. O número zero é reservado para um procedimento nulo, que é gerado automaticamente e se destina a testar se um servidor está disponível.

Essa linguagem de definição de interface fornece uma notação para definir constantes, *typedefs*, estruturas, tipos enumerados, uniões e programas. *Typedefs*, estruturas e tipos enumerados utilizam a sintaxe da linguagem C. A partir de uma definição de interface, o compilador de interface *rpcgen* pode ser usado para gerar o seguinte:

- procedimentos *stub* no cliente;
- procedimento *main*, despachante e procedimentos *stub* no servidor;
- procedimentos de empacotamento e desempacotamento da XDR, a serem usados pelo despachante e pelos procedimentos *stub* do cliente e do servidor.

Vinculação (binding) • A RPC da Sun executa em cada computador em um número de porta bem conhecido um serviço de vinculação (*binding*) local chamado *mapeador de porta* (*port mapper*). Cada instância de um mapeador de porta grava o número do programa, o número da versão e o número da porta em uso para cada um dos serviços em execução local. Quando um servidor é iniciado, ele registra seu número de programa, número de versão e número de porta no mapeador de porta. Quando um cliente é iniciado, ele descobre a porta do serviço fazendo uma requisição remota para o mapeador de porta no computador onde o servidor executa, especificando o número de programa e o número de versão.

Quando um serviço tem várias instâncias sendo executadas em diferentes computadores, elas podem usar diferentes números de porta para receber as requisições dos clientes. Se um cliente precisa enviar uma requisição para todas as instâncias de um serviço que usam diferentes números de porta, ele não pode usar *multicast* IP direto para esse propósito. A solução é os clientes fazerem chamadas de procedimento remoto por *multicast* para todos os mapeadores de porta, especificando o número do programa e da versão. Cada mapeador de porta encaminha todas essas chamadas para o programa de serviço local apropriado, se houver um.

Autenticação • As mensagens de requisição-resposta da RPC da Sun fornecem campos adicionais que permitem autenticar as informações a serem passadas entre cliente e servidor. A mensagem de requisição contém as credenciais do usuário que está executando o programa cliente. Por exemplo, na autenticação UNIX, as credenciais incluem o *uid* e o *gid* do usuário. Mecanismos de controle de acesso podem ser construídos sobre informações de autenticação repassadas aos procedimentos do servidor por intermédio de um segundo argumento. O programa servidor é responsável por impor o controle de acesso, decidindo se vai ou não executar cada chamada de procedimento, de acordo com as informações de autenticação. Por exemplo, ao se tratar de um servidor de arquivos NFS, ele pode verificar se o usuário tem direitos suficientes para executar a operação solicitada em determinado arquivo. Vários protocolos de autenticação diferentes podem ser suportados. Isso inclui:

- nenhum;
- estilo UNIX, como descrito anteriormente;

- baseado em uma chave compartilhada para assinar as mensagens RPC;
- Kerberos (veja o Capítulo 11).

Um campo no cabeçalho RPC indica qual estilo está sendo usado.

Uma estratégia de segurança mais genérica está descrita na RFC 2203 [Eisler *et al.* 1997]. Ela proporciona sigilo e integridade das mensagens RPC, assim como autenticação. A estratégia permite que cliente e servidor negociem um contexto de segurança em que escolhem não usar segurança alguma ou, no caso de ser exigida segurança, garantir integridade das mensagens, garantir privacidade das mensagens ou garantir ambas.

Programas clientes e servidores • Mais material sobre a RPC da Sun está disponível no endereço [www.cdk5.net/rmi] (em inglês). Ele inclui exemplos de programas clientes e servidores correspondentes à interface definida na Figura 5.11.

5.4 Invocação a método remoto

A RMI (Remote Method Invocation – invocação a método remoto) está intimamente relacionada à RPC, mas é estendida para o mundo dos objetos distribuídos. Na RMI, um objeto chamador pode invocar um método em um objeto potencialmente remoto. Assim como na RPC, geralmente os detalhes subjacentes ficam ocultos para o usuário. As características comuns entre a RMI e a RPC são:

- Ambas suportam programação com interfaces, com as vantagens resultantes advindas dessa estratégia (veja a Seção 5.3.1).
- Normalmente, ambas são construídas sobre protocolos de requisição-resposta e podem oferecer diversas semânticas de chamada, como *pelo menos uma vez* e *no máximo uma vez*.
- Ambas oferecem um nível de transparência semelhante – isto é, as chamadas locais e remotas empregam a mesma sintaxe, mas as interfaces remotas normalmente expõem a natureza distribuída da chamada subjacente, suportando exceções remotas, por exemplo.

As diferenças a seguir levam a uma maior expressividade na programação de aplicações e serviços distribuídos complexos.

- O programador pode usar todo o poder expressivo da programação orientada a objetos no desenvolvimento de *software* de sistemas distribuídos, incluindo o uso de objetos, classes e herança, e também pode empregar metodologias de projeto orientado a objetos relacionadas e ferramentas associadas.
- Complementando o conceito de identidade de objeto dos sistemas orientados a objetos, em um sistema baseado em RMI, todos os objetos têm referências exclusivas (sejam locais ou remotos) e tais referências também podem ser passadas como parâmetros, oferecendo, assim, uma semântica de passagem de parâmetros significativamente mais rica do que na RPC.

A questão da passagem de parâmetros é particularmente importante nos sistemas distribuídos. A RMI permite ao programador passar parâmetros não apenas por valor, como parâmetros de entrada ou saída, mas também por referência de objeto. Passar referências é particularmente atraente se o parâmetro for grande ou complexo. Então, o lado remoto, ao receber uma referência de objeto, pode acessar esse objeto usando invocação a método remoto, em vez de transmitir o valor do objeto pela rede.

O restante desta seção examinará o conceito de invocação a método remoto com mais detalhes, considerando inicialmente os principais problemas envolvidos nos modelos de objeto distribuído, antes de observar os problemas de implementação relacionados à RMI, incluindo a coleta de lixo distribuída.

5.4.1 Questões de projeto para RMI

Conforme mencionado anteriormente, a RMI compartilha os mesmos problemas de projeto da RPC em termos de programação com interfaces, semântica de chamada e nível de transparência. O leitor deve consultar a Seção 5.3.1 para uma discussão sobre esses itens.

O principal problema de projeto se relaciona com o modelo de objeto e, em particular, com a transição de objetos para objetos distribuídos. Descreveremos primeiro o modelo de objeto de imagem única convencional e, depois, consideraremos o modelo de objeto distribuído.

O modelo de objeto • Um programa orientado a objetos, por exemplo em Java ou C++, consiste em um conjunto de objetos interagindo, cada um dos quais composto de um conjunto de dados e um conjunto de métodos. Um objeto se comunica com outros objetos invocando seus métodos, geralmente passando argumentos e recebendo resultados. Os objetos podem encapsular seus dados e o código de seus métodos. Algumas linguagens, como Java e C++, permitem que os programadores definam objetos cujas variáveis de instância podem ser acessadas diretamente. No entanto, para uso em um sistema de objeto distribuído, os dados de um objeto devem ser acessíveis somente por intermédio de seus métodos.

Referências de objeto: os objetos podem ser acessados por meio de referências de objeto. Por exemplo, em Java, uma variável que parece conter um objeto, na verdade, contém uma referência para esse objeto. Para invocar um método em um objeto, são fornecidos a referência do objeto e o nome do método, junto aos argumentos necessários. O objeto cujo método é invocado é chamado de *alvo*, de *destino* e, às vezes, de *receptor*. As referências de objeto são valores de primeira classe, significando que eles podem, por exemplo, ser atribuídos a variáveis, passados como argumentos e retornados como resultados de métodos.

Interfaces: uma interface fornece a definição das assinaturas de um conjunto de métodos (isto é, os tipos de seus argumentos, valores de retorno e exceções), sem especificar sua implementação. Um objeto fornecerá uma interface em particular, caso sua classe contenha código que implemente os métodos dessa interface. Em Java, uma classe pode implementar várias interfaces, e os métodos de uma interface podem ser implementados por qualquer classe. Uma interface também define os tipos que podem ser usados para declarar o tipo de variáveis ou dos parâmetros e valores de retorno dos métodos. Note que as interfaces não têm construtores.

Ações: em um programa orientado a objetos, a ação é iniciada por um objeto invocando um método em outro objeto. Uma invocação pode incluir informações adicionais (argumentos) necessárias para executar o método. O receptor executa o método apropriado e depois retorna o controle para o objeto que fez a invocação, algumas vezes fornecendo um resultado. A ativação de um método pode ter três efeitos:

1. O estado do receptor pode ser alterado.
2. Um novo objeto pode ser instanciado; por exemplo, pelo uso de um construtor em Java ou C++.
3. Outras invocações podem ocorrer nos métodos de outros objetos.

Como uma invocação pode levar a mais invocações a métodos em outros objetos, uma ação é um encadeamento de invocações relacionadas de métodos, cada uma com seu respectivo retorno.

Exceções: os programas podem encontrar muitos tipos de erros e condições inesperadas, de diversos graus de gravidade. Durante a execução de um método, muitos problemas diferentes podem ser descobertos: por exemplo, valores inconsistentes nas variáveis do objeto ou falhas nas tentativas de ler ou gravar arquivos ou soquetes de rede. Quando os programadores precisam inserir testes em seu código para tratar de todos os casos incomuns ou errôneos possíveis, isso diminui a clareza do caso normal. As exceções proporcionam uma maneira clara de tratar com condições de erro, sem complicar o código. Além disso, cada cabeçalho de método lista explicitamente como exceções as condições de erro que pode encontrar, permitindo que os usuários desse método as tratem adequadamente. Pode ser definido um bloco de código para *disparar* uma exceção, quando surgirem condições inesperadas ou erros em particular. Isso significa que o controle passa para outro bloco de código, que *captura* a exceção. O controle não retorna para o lugar onde a exceção foi disparada.

Coleta de lixo (garbage collection): é necessário fornecer uma maneira de liberar o espaço em memória ocupado pelos objetos quando eles não são mais necessários. Uma linguagem (por exemplo, Java) que pode detectar automaticamente quando um objeto não está mais acessível, recupera o espaço em memória e o torna disponível para alocação por outros objetos. Esse processo é chamado de *coleta de lixo* (*garbage collection*). Quando uma linguagem não suporta coleta de lixo (por exemplo, C++), o programador tem de se preocupar com a liberação do espaço alocado para os objetos. Isso pode ser uma fonte de erros significativa.

Objetos distribuídos • O estado de um objeto consiste nos valores de suas variáveis de instância. No paradigma baseado em objetos, o estado de um programa é dividido em partes separadas, cada uma associada a um objeto. Como os programas baseados em objetos são logicamente particionados, a distribuição física dos objetos em diferentes processos ou computadores de um sistema distribuído é uma extensão natural (o problema do posicionamento foi discutido na Seção 2.3.1).

Os sistemas de objetos distribuídos podem adotar a arquitetura cliente-servidor. Nesse caso, os objetos são gerenciados pelos servidores, e seus clientes invocam seus métodos usando invocação a método remoto, RMI. Na RMI, a requisição de um cliente para invocar um método de um objeto é enviada em uma mensagem para o servidor que gerencia o objeto. A invocação é realizada pela execução de um método do objeto no servidor e o resultado é retornado para o cliente em outra mensagem. Para permitir encadeamentos de invocações relacionadas, os objetos nos servidores podem se tornar clientes de objetos em outros servidores.

Os objetos distribuídos podem adotar outros modelos arquiteturais. Por exemplo, os objetos podem ser replicados para usufruir as vantagens normais da tolerância a falhas e do melhor desempenho, e os objetos podem ser migrados com o intuito de melhorar seu desempenho e sua disponibilidade.

O fato de ter objetos clientes e servidores em diferentes processos impõe o encapsulamento. Isso significa que o estado de um objeto pode ser acessado somente pelos métodos do objeto, implicando que não é possível a métodos não autorizados agirem livremente sobre o estado. Por exemplo, a possibilidade de RMIs concorrentes, a partir de objetos em diferentes computadores, significa que um objeto pode ser acessado de forma concorrente. Portanto, surge a possibilidade de conflito de acessos. Entretanto, o fato de os dados de um objeto serem acessados somente pelos próprios métodos permite que os objetos forneçam métodos para protegerem-se contra acessos incorretos. Neste caso, por exemplo, eles podem usar primitivas de sincronização, como variáveis de condição, para proteger o acesso a suas variáveis de instância.

Outra vantagem de tratar o estado compartilhado de um programa distribuído como um conjunto de objetos é que um objeto pode ser acessado via RMI ou ser copiado em

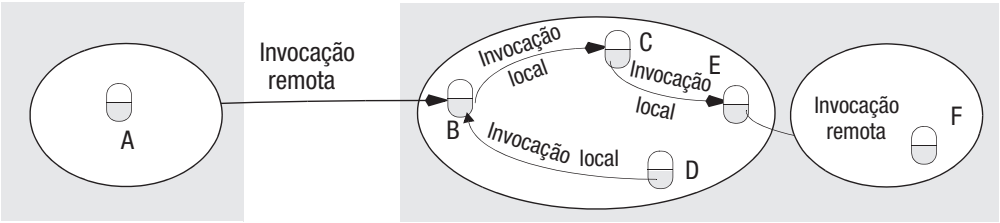

Figura 5.12 Invocação a métodos locais e remotos.

uma cache local e acessado diretamente, desde que a implementação da classe esteja disponível de forma local.

O fato de os objetos serem acessados somente por meio de seus métodos oferece outra vantagem para sistemas heterogêneos, pois diferentes formatos de dados podem ser usados em diferentes instalações – esses formatos não serão notados pelos clientes que utilizam RMI para acessar os métodos dos objetos.

O modelo de objeto distribuído • Esta seção discute as extensões feitas no modelo de objeto para torná-lo aplicável aos objetos distribuídos. Cada processo contém um conjunto de objetos, alguns dos quais podem receber invocações locais e remotas, enquanto os outros objetos podem receber somente invocações locais, como mostrado na Figura 5.12. As invocações a métodos entre objetos em diferentes processos, sejam no mesmo computador ou não, são conhecidas como *invocações a métodos remotos*. As invocações a métodos entre objetos no mesmo processo são invocações a métodos locais.

Referimo-nos aos objetos que podem receber invocações remotas como *objetos remotos*. Na Figura 5.12, B e F são objetos remotos. Todos os objetos podem receber invocações locais de outros objetos que contenham referências a eles. Por exemplo, o objeto C deve ter uma referência ao objeto E para poder invocar um de seus métodos. Os dois conceitos fundamentais a seguir estão no centro do modelo de objeto distribuído:

> *Referência de objeto remoto:* outros objetos podem invocar os métodos de um objeto remoto se tiverem acesso a sua *referência de objeto remoto*. Por exemplo, na Figura 5.12, uma referência de objeto remoto de B deve estar disponível para A.
>
> *Interface remota:* todo objeto remoto tem uma interface remota especificando qual de seus métodos pode ser invocado de forma remota. Por exemplo, os objetos B e F devem ter interfaces remotas.

Os parágrafos a seguir discutem as referências de objeto remoto, as interfaces remotas e outros aspectos do modelo de objeto distribuído.

Referências de objeto remoto: a noção de referência de objeto é estendida para permitir que qualquer objeto que possa receber uma RMI tenha uma referência de objeto remoto. Uma referência de objeto remoto é um identificador que pode ser usado por todo um sistema distribuído para se referir a um objeto remoto único em particular. Sua representação, que geralmente é diferente da representação de referências de objeto local, foi discutida na Seção 4.3.4. As referências de objeto remoto são análogas às locais, pois:

1. O objeto remoto que vai receber uma invocação a método remoto é especificado pelo *invocador* como uma referência de objeto remoto.
2. As referências de objeto remoto podem ser passadas como argumentos e resultados de invocações a métodos remotos.

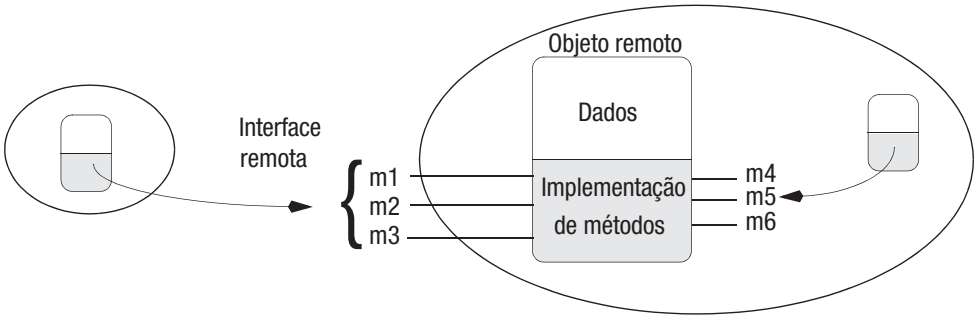

Figura 5.13 Um objeto remoto e sua interface remota.

Interfaces remotas: a classe de um objeto remoto implementa os métodos de sua interface remota; por exemplo, como métodos de instância públicos em Java. Objetos em outros processos só podem invocar os métodos pertencentes à interface remota, como mostrado na Figura 5.13. Os objetos locais podem invocar os métodos da interface remota, assim como outros métodos implementados por um objeto remoto. Note que as interfaces remotas, assim como todas as interfaces, não têm construtores.

O sistema CORBA fornece uma linguagem de definição de interface (IDL) que é usada para definir interfaces remotas. Veja, na Figura 5.8, um exemplo de interface remota definida na IDL do CORBA. As classes de objetos remotos e os programas clientes podem ser implementados em qualquer linguagem, como C++, Java ou Python, para a qual esteja disponível um compilador de IDL. Os clientes CORBA não precisam usar a mesma linguagem que o objeto remoto para invocar seus métodos de forma remota.

Na RMI Java, as interfaces remotas são definidas da mesma forma que qualquer outra interface Java. Elas adquirem a capacidade de ser interfaces remotas estendendo uma interface como *Remote*. Tanto a IDL do CORBA (Capítulo 8) como a linguagem Java suportam herança múltipla de interfaces, isto é, uma interface pode estender uma ou mais interfaces.

Ações em um sistema de objeto distribuído • Assim como no caso não distribuído, uma ação é iniciada por uma invocação a método, a qual pode resultar em mais invocações a métodos em outros objetos. Porém, no caso distribuído, os objetos envolvidos em um encadeamento de invocações relacionadas podem estar localizados em diferentes processos ou em diferentes computadores. Quando uma invocação cruza o limite de um processo ou computador, a RMI é usada e a referência remota do objeto deve estar disponível para o invocador. Na Figura 5.12, o objeto A precisa conter uma referência de objeto remoto para o objeto B. As referências de objeto remoto podem ser obtidas como resultado de invocações a métodos remotos. Por exemplo, na Figura 5.12, o objeto A poderia obter uma referência remota para o objeto F a partir do objeto B.

Quando uma ação levar à instanciação de um novo objeto, esse objeto normalmente ficará dentro do processo em que a instanciação é feita; por exemplo, onde o construtor foi usado. Se o objeto recentemente instanciado tiver uma interface remota, ele será um objeto remoto com uma referência de objeto remoto.

Os aplicativos distribuídos podem fornecer objetos remotos com métodos para instanciar objetos que podem ser acessados por uma RMI, fornecendo assim, efetivamente, o efeito da instanciação remota de objetos. Suponha, por exemplo, que o objeto L da Figura 5.14 contivesse um método para criar objetos remotos e que as invocações remotas de C e K pudessem levar à instanciação dos objetos M e N, respectivamente.

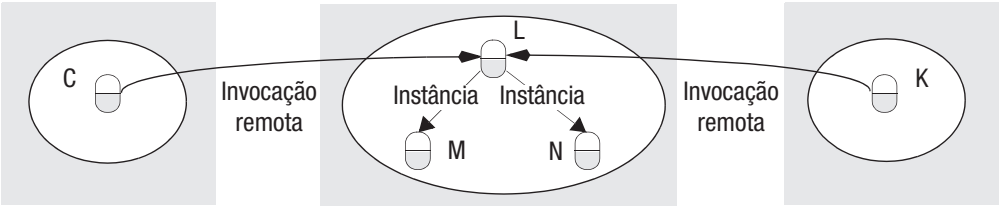

Figura 5.14 Instanciação de objetos remotos.

Coleta de lixo em um sistema de objeto distribuído: se uma linguagem, por exemplo Java, suporta coleta de lixo, então qualquer sistema RMI associado deve permitir a coleta de lixo de objetos remotos. A coleta de lixo distribuída geralmente é obtida pela cooperação entre o coletor de lixo local existente e um módulo adicionado que realiza uma forma de coleta de lixo distribuída, normalmente baseada em contagem de referência. A Seção 5.4.3 descreverá esse esquema em detalhes. Se a coleta de lixo não estiver disponível, os objetos remotos que não são mais necessários deverão ser excluídos.

Exceções: uma invocação remota pode falhar por motivos relacionados ao fato de o objeto invocado estar em um processo ou computador diferente do invocador. Por exemplo, o processo que contém o objeto remoto pode ter falhado ou estar ocupado demais para responder, ou a mensagem de invocação ou de resultado pode ter se perdido. Portanto, a invocação a método remoto deve ser capaz de lançar exceções, como expiração de tempo limite (*timeout),* causados pelo envio de mensagens, assim como as ocorridas durante a execução do método invocado. Exemplos destas últimas são uma tentativa de ler além do final de um arquivo ou de acessar um arquivo sem as permissões corretas.

A IDL do CORBA fornece uma notação para especificar exceções em nível de aplicativo e o sistema subjacente gera exceções padrão quando ocorrem erros devido à distribuição de mensagens. Os programas clientes CORBA precisam ser capazes de tratar as exceções. Por exemplo, um programa cliente em C++ usará os mecanismos de exceção do C++.

5.4.2 Implementação de RMI

Vários objetos e módulos separados estão envolvidos na realização de uma invocação a método remoto. Eles aparecem na Figura 5.15, na qual um objeto em nível de aplicativo A invoca um método em um objeto remoto B, para o qual mantém uma referência de objeto remoto. Esta seção discute as funções de cada um dos componentes mostrados nessa figura, tratando primeiro da comunicação e dos módulos de referência remota e, depois, do *software* RMI neles executado.

Em seguida, exploraremos os seguintes tópicos relacionados: a geração de *proxies*, a associação de nomes às suas referências de objeto remoto, a invocação e a passividade de objetos e a localização de objetos a partir de suas referências de objeto remoto.

Módulo de comunicação • Dois módulos de comunicação cooperam para executar o protocolo de requisição-resposta, o qual transmite mensagens de *requisição*-re*sposta* entre o cliente e o servidor. O conteúdo das mensagens de *requisição*-re*sposta* aparece na Figura 5.4. O módulo de comunicação usa apenas os três primeiros elementos, os quais especificam o tipo de mensagem, sua *requestId* e a referência remota do objeto a ser invocado. O identificador da operação (*OperationId*) e todo o empacotamento e desempacotamento são de responsabilidade do *software* RMI, como discutido a seguir. Juntos, os módulos

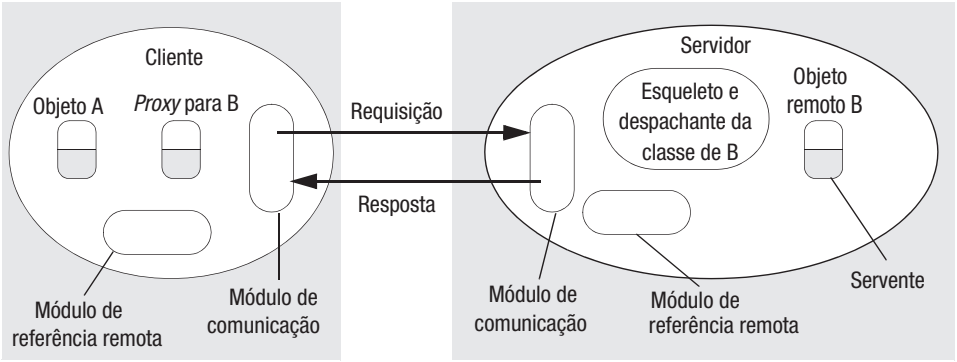

Figura 5.15 A função do *proxy* e do esqueleto na invocação a método remoto.

de comunicação são responsáveis por fornecer uma semântica de invocação específica, por exemplo, *no máximo uma vez*.

O módulo de comunicação no servidor seleciona o despachante para a classe do objeto a ser invocado, passando sua referência local, obtida a partir do módulo de referência remota, através do identificador do objeto remoto dado pela mensagem de requisição. A função do despachante será discutida no *software* RMI, a seguir.

Módulo de referência remota • O módulo de referência remota é responsável pela transformação entre referências de objeto local e remoto e pela criação de referências de objeto remoto. Para executar sua funcionalidade, o módulo de referência remota de cada processo contém uma *tabela de objetos remotos* para registrar a correspondência entre as referências de objeto local desse processo e as referências de objeto remoto (que abrangem todo o sistema). A tabela inclui:

- Uma entrada para todos os objetos remotos mantidos pelo processo. Por exemplo, na Figura 5.15, o objeto remoto B será registrado na tabela do servidor.
- Uma entrada para cada *proxy* local. Por exemplo, na Figura 5.15, o *proxy* de B será registrado na tabela do cliente.

A função de um *proxy* é discutida na subseção sobre *software* RMI. As ações do módulo de referência remota são as seguintes:

- Quando um objeto remoto precisa ser passado como argumento ou resultado pela primeira vez, o módulo de referência remota cria uma referência de objeto remoto, a qual adiciona em sua tabela.
- Quando uma referência de objeto remoto chega em uma mensagem de requisição ou resposta, é solicitada ao módulo de referência remota a correspondente referência de objeto local, a qual pode se referir a um *proxy* ou a um objeto remoto. No caso da referência de objeto remoto não estar na tabela, o *software* RMI cria um novo *proxy* e pede ao módulo de referência remota para adicioná-lo na tabela.

Esse módulo é chamado pelos componentes do *software* RMI quando estão empacotando e desempacotando referências de objeto remoto. Por exemplo, quando chega uma mensagem de requisição, a tabela é usada para descobrir qual objeto local deve ser invocado.

Serventes • Um *servente* é uma instância de uma classe que fornece o corpo de um objeto remoto. É o servente que trata as requisições remotas repassadas pelo esqueleto cor-

respondente. Os serventes são partes integrantes do processo servidor. Eles são criados quando os objetos remotos são instanciados e permanecem em uso até não serem mais necessários, sendo finalmente excluídos.

O software RMI • É uma camada de *software* – *middleware* – entre os objetos do aplicativo e os módulos de comunicação e de referência remota. As funções dos seus componentes, mostrados na Figura 5.15, são as seguintes:

Proxy: a função de um *proxy* é tornar a invocação a método remoto transparente para os clientes, comportando-se como um objeto local para o invocador; mas, em vez de executar uma invocação local, ele a encaminha em uma mensagem para um objeto remoto. Ele oculta os detalhes da referência de objeto remoto, do empacotamento de argumentos, do desempacotamento dos resultados e do envio e recepção de mensagens do cliente. Existe um *proxy* para cada objeto remoto a que um processo faz uma referência de objeto remoto. A classe de um *proxy* implementa os métodos da interface remota do objeto remoto que ele representa. Isso garante que as invocações a métodos remotos sejam apropriadas para o objeto remoto em questão. Entretanto, o *proxy* implementa os métodos de uma forma diferente. Cada método do *proxy* empacota uma referência para o objeto alvo, o *OperationId* e seus argumentos em uma mensagem de *requisição* e a envia para o objeto alvo. A seguir, espera pela mensagem de *resposta*; quando a recebe, desempacota e retorna os resultados para o invocador.

Despachante: um servidor tem um despachante e um esqueleto para cada classe que representa um objeto remoto. Em nosso exemplo, o servidor tem um despachante e um esqueleto para a classe do objeto remoto B. O despachante recebe a mensagem de *requisição* do módulo de comunicação e utiliza o *OperationId* para selecionar o método apropriado no esqueleto, repassando a mensagem de *requisição*. O despachante e o *proxy* usam o mesmo *OperationId* para os métodos da interface remota.

Esqueleto: a classe de um objeto remoto tem um *esqueleto*, que implementa os métodos da interface remota, mas de uma forma diferente dos métodos implementados no servente que personifica o objeto remoto. Um método de esqueleto desempacota os argumentos na mensagem de *requisição* e invoca o método correspondente no servente. Ele espera que a invocação termine e, em seguida, empacota o resultado, junto às exceções, em uma mensagem de *resposta* que é enviada para o método do *proxy* que fez a requisição.

As referências de objeto remoto são empacotadas na forma mostrada na Figura 4.13, que inclui informações sobre a interface remota do objeto remoto; por exemplo, o nome da interface remota ou da classe do objeto remoto. Essas informações permitem que a classe do *proxy* seja determinada para que, quando necessário, um novo *proxy* possa ser criado. Por exemplo, o nome da classe do *proxy* pode ser gerado anexando"*_proxy*" ao nome da interface remota.

Geração das classes para proxies, despachantes e esqueletos • As classes para *proxy*, despachante e esqueleto usados na RMI são geradas automaticamente por um compilador de interface. Por exemplo, na implementação Orbix do CORBA, as interfaces dos objetos remotos são definidas na IDL do CORBA e o compilador de interface pode ser usado para gerar as classes para *proxies*, despachantes e esqueletos em C++ ou em Java [www.iona.com]. Para a RMI Java, o conjunto de métodos oferecidos por um objeto

remoto é definido como uma interface Java implementada dentro da classe do objeto remoto. O compilador da RMI Java gera as classes de *proxy*, despachante e esqueleto a partir da classe do objeto remoto.

Ativação dinâmica: uma alternativa aos proxies • O *proxy* que acabamos de descrever é estático, no sentido de que sua classe é gerada a partir de uma definição de interface e, depois, compilada no código do cliente. Às vezes, isso não é prático: suponha que um programa cliente receba uma referência remota para um objeto cuja interface remota não estava disponível no momento da compilação. Neste caso, ele precisa de outra maneira para invocar o objeto remoto: isso é chamado de *invocação dinâmica* e dá ao cliente acesso a uma representação genérica de uma invocação remota, como o método *DoOperation* usado no Exercício 5.18, que está disponível como parte da infraestrutura de RMI (veja a Seção 5.4.1). O cliente fornecerá a referência de objeto remoto, o nome do método e os argumentos para *DoOperation* e depois esperará para receber os resultados.

Note que, embora a referência de objeto remoto inclua informações sobre a interface do objeto remoto, como seu nome, isso não é suficiente – os nomes dos métodos e os tipos dos argumentos são necessários para se fazer uma invocação dinâmica. O CORBA fornece suas informações por meio de um componente chamado *Interface Repository* (repositório de interfaces), que será descrito no Capítulo 8.

A interface de invocação dinâmica não é tão conveniente para usar como *proxy*, mas é útil em aplicações em que algumas das interfaces dos objetos remotos não podem ser previstas no momento do projeto. Um exemplo de tal aplicação é o quadro branco compartilhado usado para ilustrar a RMI Java (Seção 5.5), o CORBA (Capítulo 8) e os serviços Web (Seção 9.2.3). Resumindo: o aplicativo de quadro branco compartilhado exibe muitos tipos diferentes de figuras, como círculos, retângulos e linhas, mas precisa mostrar novas figuras que não foram previstas quando o cliente foi compilado. Um cliente que utilize invocação dinâmica é capaz de resolver esse problema. Na Seção 5.5, veremos que o *download* dinâmico de classes em clientes é uma alternativa à invocação dinâmica. Isso está disponível na RMI Java.

Esqueletos dinâmicos: a partir do exemplo anterior, fica claro que um servidor também precisará conter objetos remotos cujas interfaces não eram conhecidas no momento da compilação. Por exemplo, um cliente pode fornecer um novo tipo de figura para que o servidor de quadro branco compartilhado armazene. Um servidor com esqueletos dinâmicos resolveria essa situação. Vamos deixar para descrever os esqueletos dinâmicos no capítulo sobre CORBA (Capítulo 8). Entretanto, conforme veremos na Seção 5.5, a RMI Java resolve esse problema usando um despachante genérico e o *download* dinâmico de classes no servidor.

Programas clientes e servidores • O programa servidor contém as classes para os despachantes e esqueletos, junto às implementações das classes de todos os serventes que suporta. Além disso, o programa servidor contém uma seção de *inicialização* (por exemplo, em um método *main* em Java ou C++). A seção de *inicialização* é responsável por criar e inicializar pelo menos um dos serventes que fazem parte do servidor. Mais serventes podem ser criados em resposta às requisições dos clientes. A seção de *inicialização* também pode registrar alguns de seus serventes com um vinculador (*binder*) (veja o próximo parágrafo); porém, geralmente, é registrado apenas um servente, o qual pode ser usado para acessar os restantes.

O programa cliente conterá as classes dos *proxies* de todos os objetos remotos que ativará. Ele pode usar um vinculador para pesquisar referências de objeto remoto.

Métodos de fábrica: mencionamos anteriormente que as interfaces de objeto remoto não incluem construtores. Isso significa que os serventes não podem ser criados por invocação remota em construtores. Os serventes são criados pela *inicialização* ou por métodos de uma interface remota destinados a esse propósito. O termo *método de fábrica* é também usado para se referir a um método que cria serventes, e um *objeto de fábrica* é um objeto com métodos de fábrica. Todo objeto remoto que precise criar novos objetos remotos a pedido dos clientes deve fornecer métodos em sua interface remota para esse propósito. Tais métodos são chamados de métodos de fábrica, embora na verdade sejam apenas métodos normais.

O vinculador (binder) • Os programas clientes geralmente exigem uma maneira de obter uma referência de objeto remoto para pelo menos um dos objetos remotos mantidos por um servidor. Por exemplo, na Figura 5.12, o objeto A exige uma referência de objeto remoto para o objeto B. Em um sistema distribuído, um *vinculador* (*brinder*) é um serviço separado que mantém uma tabela contendo mapeamentos dos nomes textuais para referências de objeto remoto. Ele é usado pelos servidores para registrar seus objetos remotos pelo nome e pelos clientes para pesquisá-los. O Capítulo 8 traz uma discussão sobre o serviço de atribuição de nomes (Naming Service) do CORBA. O vinculador Java, RMIregistry, será discutido brevemente no estudo de caso sobre a RMI Java, na Seção 5.5.

Threads no servidor • Quando um objeto executa uma invocação remota, ele pode acarretar execução em mais invocações a métodos em outros objetos remotos, os quais podem levar algum tempo para retornar. Para evitar que a execução de uma invocação remota atrase a execução de outra, geralmente os servidores criam uma *thread* para cada invocação remota. Quando isso acontece, o projetista da implementação de um objeto remoto deve levar em conta os efeitos das execuções concorrentes sobre seu estado.

Invocação de objetos remotos • Algumas aplicações exigem que as informações sejam válidas por longos períodos de tempo. Entretanto, não é prático que os objetos que representam tais informações sejam mantidos por períodos ilimitados em processos que estejam em execução, particularmente porque eles não estão necessariamente em uso o tempo todo. Para evitar o potencial desperdício de recursos, devido à execução de todos os servidores que gerenciam objetos remotos, os servidores podem ser iniciados somente quando forem necessários para os clientes. Isso é análogo ao que acontece no conjunto padrão de serviços TCP, como o FTP, que são iniciados de acordo com a demanda, por um serviço chamado *Inetd*. Os processos que iniciam processos servidores para conter objetos remotos são chamados de *ativadores* pelos motivos a seguir.

Um objeto remoto é dito *ativo* quando está disponível para invocação dentro de um processo em execução, e é chamado de *passivo* se não estiver ativo no momento da invocação, mas puder se tornar. Um objeto passivo consiste em duas partes:

1. na implementação de seus métodos;
2. em seu estado na forma empacotada.

A *ativação* consiste na criação de um objeto ativo a partir do objeto passivo correspondente pela criação de uma nova instância de sua classe e pela inicialização de suas variáveis de instância a partir do estado armazenado. Os objetos passivos podem ser ativados por demanda; por exemplo, quando eles precisam ser invocados por outros objetos.

Um *ativador* é responsável por:

- Registrar os objetos passivos que estão disponíveis para ativação, o que envolve registrar os nomes dos servidores com os URLs ou nomes de arquivo dos objetos passivos correspondentes.

- Iniciar processos servidores, identificá-los e ativar objetos remotos neles.
- Controlar a localização dos servidores de objetos remotos que já tenha ativado.

A RMI Java fornece a capacidade de tornar objetos remotos *passíveis de ativação* [java.sun.com IX]. Quando um objeto passível de ativação é invocado, se já não estiver correntemente ativo, ele se tornará ativo a partir de seu estado empacotado e, depois, será invocado. A RMI Java emprega um ativador para cada computador servidor.

O estudo de caso do CORBA, no Capítulo 8, descreverá o repositório de implementação – uma forma simplificada de ativador para disparar serviços contendo objetos em um estado inicial.

Repositório de objetos persistentes • Um objeto que mantém seu estado entre invocações é chamado de *objeto persistente*. Geralmente, os objetos persistentes são gerenciados por repositórios de objetos persistentes, que guardam seus estados em uma forma empacotada no disco. Exemplos incluem o serviço de estado persistente do CORBA (veja o Capítulo 8), *Java Data Objects* [java.sun.com VIII] e *Persistent Java* [Jordan 1996, java.sun.com IV].

Em geral, um repositório de objetos persistentes gerencia uma grande quantidade de objetos, os quais são armazenados em disco ou em um banco de dados até serem necessários. Eles serão ativados quando seus métodos forem invocados por outros objetos. A ativação geralmente é projetada para ser transparente – isto é, o invocador não deve saber se um objeto já está na memória principal ou se precisa ser ativado antes de seu método ser invocado. Os objetos persistentes que não são mais necessários na memória principal podem ser postos em estado passivo. Na maioria dos casos, os objetos são salvos no repositório de objetos persistentes sempre que atingirem um estado consistente, para fornecer um grau de tolerância a falhas. O repositório de objetos persistentes precisa de uma estratégia para decidir quando deve tornar passivo os objetos. Por exemplo, ele pode fazer isso em resposta a um pedido feito pelo programa que ativou os objetos, no final de uma transação ou quando o programa termina. Geralmente, os repositórios de objetos persistentes tentam otimizar esse procedimento salvando apenas os objetos que foram modificados desde a última vez em que foram salvos.

De modo geral, os repositórios de objetos persistentes permitem que conjuntos de objetos persistentes relacionados tenham nomes legíveis por seres humanos, como nomes de caminho ou URLs. Na prática, cada um desses nomes é associado à raiz de um conjunto de objetos persistentes.

Existem duas abordagens para decidir se um objeto é persistente ou não:

- O repositório de objetos persistentes mantém raízes persistentes e todo objeto atingido a partir de uma raiz persistente é definido como persistente. Esta estratégia é usada por *Persistent Java*, *Java Data Objects* e por *PerDiS* [Ferreira *et al.* 2000]. Elas utilizam um coletor de lixo para descartar os objetos que não podem mais ser atingidos a partir das raízes persistentes.
- O repositório de objetos persistentes fornece algumas classes bases para a obtenção de persistência – os objetos persistentes pertencem a suas subclasses. Por exemplo, em *Arjuna* [Parrington *et al.* 1995], os objetos persistentes são baseados nas classes C++ que fornecem transações e recuperação. Os objetos que não são mais necessários devem ser explicitamente excluídos.

Alguns repositórios de objetos persistentes, por exemplo *PerDiS* e *Khazana* [Carter *et al.* 1998] permitem que os objetos sejam ativados em caches locais aos usuários, em vez de serem ativados nos servidores. Neste caso, um protocolo de consistência de cache é exigido. Mais detalhes sobre os modelos de consistência podem ser encontrados no *site*

que acompanha o livro, no capítulo da 4ª edição sobre memória compartilhada distribuída [www.cdk5.net/dsm] (em inglês).

Localização de objetos • A Seção 4.3.4 descreveu uma forma de referência de objeto remoto que continha o endereço IP e o número de porta do processo que criou o objeto remoto como uma maneira de garantir a exclusividade. Essa forma de referência de objeto remoto também pode ser usada como endereço de um objeto remoto, desde que ele permaneça no mesmo processo enquanto existir. Contudo, eventualmente, alguns objetos remotos poderão fazer parte de processos diferentes, possivelmente em computadores distintos. Neste caso, uma referência de objeto remoto não pode atuar como endereço. Os clientes que fazem invocações precisam tanto de uma referência de objeto remoto como de endereço para enviar as invocações.

Um *serviço de localização* ajuda os clientes a encontrarem objetos remotos a partir de suas referências de objeto remoto. Ele utiliza um banco de dados que faz o mapeamento das referências de objeto remoto para suas prováveis localizações correntes – as localizações são prováveis porque um objeto pode ter migrado novamente, desde a última vez que foi encontrado. Por exemplo, o sistema *Clouds* [Dasgupta *et al.* 1991] e o sistema *Emerald* [Jul *et al.* 1988] usavam um esquema de cache/*broadcast* no qual um membro de um serviço de localização em cada computador continha uma pequena cache de mapeamentos de referência de objeto remoto para localização. Se uma referência de objeto remoto estivesse na cache, a invocação era feita nesse endereço e falhava se o objeto tivesse mudado de lugar. Para localizar um objeto que tinha mudado de lugar, ou cuja localização não fosse a cache, o sistema fazia uma requisição em *broadcast*. Esse esquema pode ser aprimorado pelo uso de ponteiros de previsão de localização, os quais contêm sugestões sobre a nova localização de um objeto. Outro exemplo é o serviço de resolução de nomes, mencionado na Seção 9.1, usado para transformar o URN de um recurso em seu URL corrente.

5.4.3 Coleta de lixo distribuída

O objetivo de um coletor de lixo distribuído é garantir que, se uma referência local ou remota para um objeto ainda for mantida em algum lugar, em um conjunto de objetos distribuídos, então o próprio objeto continuará a existir; assim que não houver mais referência para ele, esse objeto será coletado e a memória utilizada por ele será recuperada.

Descrevemos o algoritmo de coleta de lixo distribuída Java, que é semelhante ao descrito por Birrell *et al.* [1995]. Ele é baseado na contagem de referência. Quando uma referência de objeto remoto for feita em um processo, um *proxy* será criado e existirá enquanto for necessário. O processo onde o objeto reside (seu servidor) deve ser informado desse *proxy* cliente. Então, posteriormente, quando o *proxy* deixar de existir no cliente, o servidor deverá ser novamente informado. O coletor de lixo distribuído trabalha em cooperação com os coletores de lixo locais, como segue:

- Cada processo servidor mantém um conjunto de nomes dos processos que contêm referências de objeto remoto para cada um de seus objetos remotos; por exemplo, *B.holders* é o conjunto de processos clientes (máquinas virtuais) que têm *proxies* para o objeto *B*. (Na Figura 5.15, esse conjunto incluirá o processo cliente ilustrado.) Esse conjunto pode ser mantido em uma coluna adicional na tabela de objetos remotos.
- Quando um cliente *C* recebe, pela primeira vez, uma referência remota para um objeto remoto particular *B*, ele faz uma invocação à *addRef(B)* no servidor desse objeto remoto e depois cria um *proxy*; o servidor adiciona *C* a *B.holders*.

- Quando o coletor de lixo de um cliente *C* percebe que um *proxy* para o objeto remoto *B* não é mais necessário, ele faz uma invocação à *removeRef(B)* no servidor correspondente e exclui o *proxy*; o servidor remove *C* de *B.holders*.
- Quando *B.holders* estiver vazio, o coletor de lixo local do servidor recuperará o espaço ocupado por *B*, a não ser que existam clientes (*holders*) locais.

Esse algoritmo é feito para ser executado por meio de comunicação do tipo requisição-resposta, com uma semântica de invocação *no máximo uma vez* entre os módulos de referência remota presentes nos processos cliente e servidor – ele não exige nenhum sincronismo global. Note que invocações normais à coleta de lixo não afetam cada RMI; elas ocorrem quando os *proxie*s são criados e excluídos.

Existe a possibilidade de que um cliente possa fazer uma invocação *removeRef(B)* praticamente ao mesmo tempo em que outro cliente faz uma invocação *addRef(B)*. Se a invocação *removeRef* chegar primeiro e *B.holders* estiver vazio, o objeto remoto *B* poderá ser excluído antes da chegada da invocação *addRef*. Para evitar essa situação, se o conjunto *B.holders* estiver vazio no momento em que uma referência de objeto remoto for transmitida, uma entrada temporária é adicionada até a chegada de *addRef*.

O algoritmo de coleta de lixo distribuída Java tolera falhas de comunicação usando a seguinte estratégia: as operações *addRef* e *removeRef* são idempotentes. No caso de uma chamada *addRef(B)* retornar uma exceção (significando que o método foi executado uma vez ou não foi executado), o cliente não criará o *proxy*, mas fará uma chamada *removeRef(B)*. O efeito de *removeRef* estará correto, tenha *addRef* obtido êxito ou não. O caso em que *removeRef* falha é tratado por *leasing*, conforme descrito a seguir.

O algoritmo de coleta de lixo distribuída Java pode tolerar a falha de processos clientes. Para conseguir isso, os servidores *cedem* seus objetos para os clientes por um tempo limitado. O período de arrendamento (*leasing*) começa quando o cliente faz uma invocação a *addRef* no servidor e termina quando o tempo tiver expirado ou quando o cliente fizer uma invocação a *removeRef* no servidor. As informações armazenadas pelo servidor, relativas a cada *leasing*, contêm o identificador da máquina virtual do cliente e o período de *leasing*. Os clientes são responsáveis por pedir ao servidor para que renove seus *leasings* antes que expirem.

Arrendamento (leasing) no Jini • O sistema distribuído Jini inclui uma especificação para *arrendamento* [Arnold *et al.* 1999] que pode ser usada nas situações em que um objeto oferece um recurso para outro objeto como, por exemplo, quando objetos remotos oferecem referências para outros objetos. Os objetos que oferecem tais recursos correm o risco de terem que mantê-los mesmo quando os usuários não estiverem mais interessados ou quando seus programas tiverem terminado. Para evitar a necessidade de protocolos complicados para descobrir se os usuários ainda estão interessados no recurso, eles são oferecidos por tempo limitado. A concessão do uso de um recurso por um período de tempo é chamada de *arrendamento* (*leasing*). O objeto que está oferecendo o recurso o manterá até o momento em que o arrendamento expirar. Os usuários do recurso são responsáveis, quando necessário, por solicitar a renovação do período de arrendamento.

O período de um arrendamento pode ser negociado entre o cedente e o receptor, embora isso não aconteça com os arrendamentos usados na RMI Java. Um objeto que representa um arrendamento implementa a interface *Lease*. Ela contém informações sobre o período de arrendamento e os métodos que permitem sua renovação ou seu cancelamento. O cedente retorna uma instância de *Lease* quando fornece um recurso para outro objeto.

5.5 Estudo de caso: RMI Java

A RMI Java estende o modelo de objeto Java para dar suporte para objetos distribuídos na linguagem Java. Em particular, ela permite que os objetos invoquem métodos em objetos remotos usando a mesma sintaxe das invocações locais. Além disso, a verificação de tipo se aplica igualmente às invocações remotas e às locais. Entretanto, um objeto que faz uma invocação remota sabe que seu destino é remoto, pois precisa lidar com exceções *RemoteException*; e o desenvolvedor de um objeto remoto sabe que ele é remoto porque precisa implementar a interface *Remote*. Embora o modelo de objeto distribuído seja integrado na linguagem Java de maneira natural, a semântica da passagem de parâmetros difere, pois o invocador e o alvo (destino) são remotos entre si.

A programação de aplicativos distribuídos na RMI Java é relativamente simples, pois se trata de um sistema desenvolvido com base em apenas uma linguagem – as interfaces remotas são definidas na linguagem Java. Se for usado um sistema que emprega várias linguagens em seu desenvolvimento, como o CORBA, o programador precisará aprender uma IDL e entender como ela faz o mapeamento em cada linguagem de implementação. Entretanto, mesmo quando é usada apenas uma linguagem de programação, o programador de um objeto remoto deve considerar seu comportamento em um ambiente concorrente.

No restante desta introdução, daremos um exemplo de interface remota e, com base nele, discutiremos a semântica da passagem de parâmetros. Finalmente, discutiremos o *download* de classes e o vinculador. A segunda seção deste estudo de caso discutirá como se faz a construção de programas cliente e servidor com base no exemplo de interface remota. A terceira seção abordará o projeto e a implementação da RMI Java. Para detalhes completos sobre a RMI Java, consulte o exercício dirigido sobre invocação remota [java.sun.com I].

Neste estudo de caso e no estudo de caso do CORBA, no Capítulo 8, assim como na discussão sobre serviços Web, no Capítulo 9, usamos um *quadro branco compartilhado* como exemplo. Trata-se de um programa distribuído que permite a um grupo de usuários compartilhar uma vista comum de uma superfície de desenho contendo objetos gráficos, como retângulos, linhas e círculos, cada um dos quais desenhado por um dos usuários. O servidor mantém o estado corrente de um desenho, fornecendo uma operação para os clientes informarem-no sobre a figura mais recente que um de seus usuários desenhou e mantendo um registro de todas as figuras que tiver recebido. O servidor também fornece operações que permitem aos clientes recuperarem as figuras mais recentes desenhadas por outros usuários, fazendo uma consulta sequencial no servidor. O servidor tem um número de versão (um valor inteiro), que é incrementado sempre que uma nova figura chega. O servidor fornece operações que permitem aos clientes perguntarem seu número de versão e o número de versão de cada figura, para que eles possam evitar a busca de figuras que já possuem.

Interfaces remotas na RMI Java • As interfaces remotas são definidas pela ampliação de uma interface chamada *Remote*, fornecida no pacote *java.rmi*. Os métodos devem disparar a exceção *RemoteException*, mas exceções específicas do aplicativo também podem ser disparadas. A Figura 5.16 mostra um exemplo de duas interfaces remotas chamadas *Shape* e *ShapeList*. Neste exemplo, *GraphicalObject* é uma classe que contém o estado de um objeto gráfico – por exemplo, seu tipo, sua posição, retângulo envoltório, cor da linha e cor de preenchimento – e fornece operações para acessar e atualizar seu estado. A classe *GraphicalObject* deve implementar a interface *Serializable*. Considere primeiramente a interface *Shape*: o método *getVersion* retorna um valor inteiro, enquanto

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
      int getVersion( ) throws RemoteException;
      GraphicalObject getAllState( ) throws RemoteException;                    1
}
public interface ShapeList extends Remote {
      Shape newShape(GraphicalObject g) throws RemoteException;                 2
      Vector allShapes( ) throws RemoteException;
      int getVersion( ) throws RemoteException;
}
```

Figura 5.16 Interfaces remotas *Shape* e *ShapeList*.

o método *getAllState* retorna uma instância da classe *GraphicalObject*. Agora, considere a interface *ShapeList*: seu método *newShape* passa como argumento uma instância de *GraphicalObject*, mas retorna como resultado um objeto com uma interface remota (isto é, um objeto remoto). Um ponto importante a se notar é que tanto objetos locais como remotos podem aparecer como argumentos e resultados em uma interface remota. Estes últimos são sempre denotados pelo nome de suas interfaces remotas. No próximo parágrafo, discutiremos como os objetos locais e os objetos remotos são passados como argumentos e resultados.

Passagem de parâmetros e resultados • Na RMI Java, supõe-se que os parâmetros de um método são parâmetros de *entrada* e o resultado de um método é um único parâmetro de *saída*. A Seção 4.3.2 descreveu a serialização Java, que é usada para empacotar argumentos e resultados na RMI Java. Qualquer objeto que seja serializável – isto é, que implemente a interface *Serializable* – pode ser passado como argumento ou ser resultado na RMI Java. Todos os tipos primitivos e objetos remotos são serializáveis. As classes de argumentos e valores de resultado são carregadas por *download* no destino pelo sistema RMI, quando necessário.

Passagem de objetos remotos: quando o tipo de um parâmetro ou valor de resultado é definido como uma interface remota, o argumento ou resultado correspondente é sempre passado como uma referência de objeto remoto. Por exemplo, na Figura 5.16, linha 2, o valor de retorno do método *newShape* é definido como *Shape* – uma interface remota. Quando uma referência de objeto remoto é recebida, ela pode ser usada para fazer chamadas RMI no objeto remoto a que se refere.

Passagem de objetos não remotos: todos os objetos não remotos serializáveis são copiados e passados por valor. Por exemplo, na Figura 5.16 (linhas 2 e 1), o argumento de *newShape* e o valor de retorno de *getAllState* são de tipo *GraphicalObject*, que é serializável e passado por valor. Quando um objeto é passado por valor, um novo objeto é criado no processo destino. Os métodos desse novo objeto podem ser invocados de forma local, possivelmente fazendo seu estado ser diferente do estado do objeto original no processo do remetente.

Assim, em nosso exemplo, o cliente usa o método *newShape* para passar uma instância de *GraphicalObject* para o servidor; o servidor cria um objeto remoto de tipo *Shape,* contendo o estado de *GraphicalObject,* e retorna uma referência de objeto remoto para ele.

void rebind (String name, Remote obj)
Este método é usado por um servidor para registrar o identificador de um objeto remoto pelo nome, conforme mostrado na Figura 5.18, linha 3.

void bind (String name, Remote obj)
Este método pode ser usado como alternativa por um servidor para registrar um objeto remoto pelo nome, mas se o nome já estiver vinculado a uma referência de objeto remoto, será disparada uma exceção.

void unbind (String name, Remote obj)
Este método remove vínculos.

Remote lookup(String name)
Este método é usado pelos clientes para procurar um objeto remoto pelo nome, conforme mostrado na Figura 5.20, linha 1. Retorna uma referência de objeto remoto.

String [ ] list( )
Este método retorna um vetor de objetos *String* contendo os nomes vinculados no registro.

Figura 5.17 A classe *Naming* de *RMIregistry* Java.

Os argumentos e valores de retorno em uma invocação remota são serializados em um fluxo de bytes, usando o método descrito na Seção 4.3.2, com as seguintes modificações:

1. Quando um objeto que implementa a interface *Remote* é serializado, ele é substituído por sua referência de objeto remoto, a qual contém o nome de sua classe (do objeto remoto).
2. Quando um objeto é serializado, suas informações de classe são anotadas com a localização da classe (como um URL), permitindo que a classe seja carregada por *download* pelo destino.

***Download* de classes** • A linguagem Java é projetada para permitir que as classes sejam carregadas, por meio de *download,* de uma máquina virtual para outra. Isso é particularmente relevante para objetos distribuídos que se comunicam por meio de invocação remota. Vimos que os objetos não remotos são passados por valor e os objetos remotos são passados por referência, como argumentos e resultados das RMIs. Se o destino ainda não possuir a classe de um objeto passado por valor, seu código será carregado por *download* automaticamente. Analogamente, se o destino de uma referência de objeto remoto ainda não possuir a classe de um *proxy*, seu código será carregado por *download* automaticamente. Isso tem duas vantagens:

1. Não há necessidade de cada usuário manter o mesmo conjunto de classes em seu ambiente de trabalho.
2. Os programas clientes e servidores podem fazer uso transparente de instâncias de novas classes quando elas forem adicionadas.

Como exemplo, considere o programa de quadro branco e suponha que sua implementação inicial de *GraphicalObject* não admita texto. Então, um cliente com um objeto textual pode implementar uma subclasse de *GraphicalObject*, que lida com texto, e passar uma instância para o servidor como argumento do método *newShape*. Depois disso, outros clientes poderão recuperar a instância usando o método *getAllState*. O código da nova classe será carregado por *download* automaticamente, do primeiro cliente para o servidor e depois, conforme for necessário, para outros clientes.

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ShapeListServer{
    public static void main(String args[ ]){
        System.setSecurityManager(new RMISecurityManager( ));
        try{
            ShapeList aShapeList = new ShapeListServant( );                    1
            ShapeList stub =                                                   2
                (ShapeList) UnicastRemoteObject.exportObject(aShapeList,0);    3
            Naming.rebind("//bruno.ShapeList", stub );                         4
            System.out.println("ShapeList server ready");
        }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage( ));}
    }
}
```

Figura 5.18 Classe *ShapeListServer* Java com o método *main*.

RMIregistry • O RMIregistry é o vinculador da RMI Java. Uma instância de RMIregistry normalmente deve ser executada em cada computador servidor que contenha objetos remotos. Ele mantém uma tabela mapeando nomes textuais no estilo dos URLs, em referências para objetos remotos contidos nesse computador. Ele é acessado por métodos da classe *Naming*, cujos métodos recebem como argumento um *string* formatado como um URL, da forma:

//nomeComputador:porta/nomeObjeto

onde *nomeComputador* e *porta* se referem à localização do RMIregistry. Se eles forem omitidos, serão presumidos como sendo o computador local e a porta padrão. Sua interface oferece os métodos mostrados na Figura 5.17, na qual as exceções não estão listadas – todos os métodos podem disparar a exceção *RemoteException*.

Usado dessa maneira, os clientes devem direcionar suas consultas de *pesquisa* para computadores específicos. Como alternativa, é possível configurar um serviço de vinculação em nível de sistema. Para se conseguir isso, é necessário executar uma instância do RMIregistry no ambiente de rede e, então, usar a classe *LocateRegistry*, que está em *java.rmi.registry*, para descobrir esse registro. Mais especificamente, essa classe contém um método *getRegistry,* que retorna um objeto de tipo *Registry* representando o serviço de vinculação remoto:

public static Registry getRegistry( ) throws RemoteException

Depois disso, é necessário fazer uma chamada para *rebind* nesse objeto *Registry* retornado, para estabelecer uma conexão com o registro RMI remoto.

5.5.1 Construção de programas cliente e servidor

Esta seção esboça as etapas necessárias para produzir programas cliente e servidor que utilizam as interfaces *Remote Shape* e *ShapeList*, mostradas na Figura 5.16. O programa servidor é uma versão simplificada do servidor de quadro branco que implementa as duas interfaces *Shape* e *ShapeList*. Descreveremos um programa cliente de consulta sequencial simples e, depois, apresentaremos a técnica de *callback*, que pode ser usada para evi-

```
import java.util.Vector;
public class ShapeListServant implements ShapeList {
      private Vector theList; // contém a lista de elementos Shapes
      private int version;
      public ShapeListServant( ){...}
      public Shape newShape(GraphicalObject g) {                    1
            version++;
            Shape s = new ShapeServant( g, version);                2
            theList.addElement(s);
            return s;
      }
      public  Vector allShapes( ){...}
      public int getVersion( ) { ... }
}
```

Figura 5.19 A classe *ShapeListServant* Java que implementa a interface *ShapeList*.

tar a necessidade de fazer a consulta sequencial no servidor. Versões completas das classes ilustradas nesta seção estão disponíveis na página do livro em [www.grupoa.com.br] ou em [www.cdk5.net/rmi].

Programa servidor • O programa é um servidor de quadro branco: ele representa cada figura como um objeto remoto instanciado por um servente que implementa a interface *Shape* e contém o estado de um objeto gráfico, assim como seu número de versão; ele representa sua coleção de figuras por meio de outro servente, que implementa a interface *ShapeList* e contém uma coleção de figuras em um *Vector*.

O servidor consiste em um método *main* e uma classe servente para implementar cada uma de suas interfaces remotas. O método *main* da classe servidora está mostrado na Figura 5.18, com as principais etapas contidas nas linhas marcadas de 1 a 4:

- Na linha 1, o servidor cria uma instância de *ShapeListServant*.
- As linhas 2 e 3 utilizam o método *exportObject* (definido em *UnicastRemoteObject*) para tornar esse objeto disponível para o *runtime* da RMI, tornando-o, com isso, disponível para receber invocações. O segundo parâmetro de *exportObject* especifica a porta TCP a ser usada para receber invocações. É uma prática normal configurar isso como zero, significando que uma porta anônima será usada (uma gerada pelo *runtime* da RMI). Usar *UnicastRemoteObject* garante que o objeto resultante durará somente enquanto o processo no qual é criado existir (uma alternativa é torná-lo um objeto *Activable*; isto é, que dura além da instância do servidor).
- Finalmente, a linha 4 vincula o objeto remoto a um nome no RMIregistry. Note que o valor vinculado ao nome é uma referência de objeto remoto e seu tipo é o tipo de sua interface remota – *ShapeList*.

As duas classes serventes são *ShapeListServant*, que implementa a interface *ShapeList*, e *ShapeServant*, que implementa a interface *Shape*. A Figura 5.19 apresenta um esboço da classe *ShapeListServant.*

A implementação dos métodos da interface remota em uma classe servente é extremamente simples, pois ela pode ser feita sem nenhuma preocupação com os detalhes da comunicação. Considere o método de *newShape* na Figura 5.19 (linha 1), que poderia

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
      public static void main(String args[ ]){
            System.setSecurityManager(new RMISecurityManager( ));
            ShapeList aShapeList = null;
            try{
                  aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList");          1
                  Vector sList = aShapeList.allShapes( );                                2
            } catch(RemoteException e) {System.out.println(e.getMessage( ));
            }catch(Exception e) {System.out.println("Client: " + e.getMessage( ));}
      }
}
```

Figura 5.20 Cliente Java para *ShapeList*.

ser chamado de método de fábrica, pois ele permite que o cliente solicite a criação de um servente. Ele usa o construtor de *ShapeServant*, o qual cria um novo servente contendo o objeto *GraphicalObject* e o número de versão passados como argumentos. O tipo do valor de retorno de *newShape* é *Shape* – a interface implementada pelo novo servente. Antes de retornar, o método *newShape* adiciona a nova figura em seu vetor, que contém a lista de figuras (linha 2).

O método *main* de um servidor precisa criar um gerenciador de segurança para permitir que a linguagem Java aplique a proteção apropriada ao servidor RMI. É fornecido um gerenciador de segurança padrão, chamado *RMISecurityManager*. Ele protege os recursos locais para garantir que as classes carregadas a partir de *sites* remotos não possam ter qualquer efeito sobre recursos como, por exemplo, arquivos; contudo, ele difere do gerenciador de segurança Java padrão, pois permite que o programa forneça seu próprio carregador de classe e use reflexão. Se um servidor RMI não configurar nenhum gerenciador de segurança, os *proxies* e as classes só poderão ser carregados a partir do caminho de classe local. O objetivo é proteger o programa do código que é carregado por *download* como resultado das invocações a métodos remotos.

Programa cliente • Um cliente simplificado para o servidor *ShapeList* está ilustrado na Figura 5.20. Todo programa cliente precisa ser iniciado usando um vinculador para pesquisar uma referência de objeto remoto. Nosso cliente configura um gerenciador de segurança e, depois, pesquisa uma referência de objeto remoto usando a operação *lookup* do RMIregistry (linha 1). Tendo obtido uma referência de objeto remoto inicial, o cliente continua, enviando RMIs para esse objeto remoto ou para outros objetos descobertos durante sua execução, de acordo com as necessidades de seu aplicativo. Em nosso exemplo, o cliente invoca o método *allShapes* no objeto remoto (linha 2) e recebe um vetor de referências de objeto remoto para todas as figuras correntemente armazenadas no servidor. Se o cliente implementasse uma tela para o quadro branco, ele usaria o método *getAllState* do servidor na interface *Shape* para recuperar cada um dos objetos gráficos do vetor e os exibir na janela. Sempre que o usuário terminar o desenho de um objeto gráfico, ele invocará o método *newShape* no servidor, passando o novo objeto gráfico como argumento. O cliente manterá um registro do número de versão mais recente no servidor e, de tempos em tempos, invocará *getVersion* no servidor para descobrir se figuras novas foram adicionadas por outros usuários. Se assim for, ele as recuperará e exibirá.

Callbacks • A ideia geral por trás das *callbacks* é que, em vez de os clientes fazerem consultas no servidor para descobrir se ocorreu algum evento, o servidor informa os clientes quando o evento ocorrer. O termo *callback* é usado para referenciar a ação de um servidor ao notificar os clientes sobre um evento. As *callbacks* podem ser implementadas na RMI como segue:

- O cliente cria um objeto remoto que implementa uma interface que contém um método para o servidor chamar. Nos referimos a isso como *objeto de callback*.
- O servidor fornece uma operação que permite aos clientes lhe informar as referências de objeto remoto de seus objetos de *callback*. O servidor as registra em uma lista.
- Quando ocorre um evento, o servidor chama os clientes interessados. Por exemplo, o servidor de quadro branco chamaria seus clientes quando um objeto gráfico fosse adicionado.

O uso de *callbacks* evita a necessidade de um cliente fazer consultas sistemáticas ao servidor para verificar seus objetos de interesse e suas consequentes desvantagens:

- O desempenho do servidor pode ser degradado pelas constantes consultas.
- Os clientes podem não notificar os usuários sobre as atualizações.

Entretanto, as *callbacks* têm seus próprios problemas: primeiro, o servidor precisa ter listas atualizadas dos objetos de *callback* dos clientes. No entanto, os clientes podem não informar o servidor antes de terminarem, deixando o servidor com listas incorretas. A técnica de arrendamento, discutida na Seção 5.4.3, pode ser usada para superar esse problema. O segundo problema associado a *callbacks* é que o servidor precisa fazer uma série de RMIs síncronas nos objetos de *callback* da lista. Consulte o Capítulo 6 para algumas ideias sobre como resolver o segundo problema.

Ilustramos o uso de *callbacks* no contexto do aplicativo de quadro branco. A interface *WhiteboardCallback* poderia ser definida como:

```
public interface WhiteboardCallback implements Remote {
    void callback(int version) throws RemoteException;
};
```

Essa interface é implementada como um objeto remoto pelo cliente, permitindo que o servidor envie ao cliente um número de versão quando um novo objeto é adicionado. Contudo, para o servidor fazer isso, é necessário que o cliente informe o seu objeto de *callback*. Para tornar isso possível, a interface *ShapeList* exige métodos adicionais, como *register* e *deregister*, definidos como segue:

```
int register(WhiteboardCallback callback) throws RemoteException;
void deregister(int callbackId) throws RemoteException;
```

Após o cliente ter obtido uma referência para o objeto remoto com a interface *ShapeList* (por exemplo, na Figura 5.20, linha 1) e criado uma instância de objeto de *callback*, ele utiliza o método *register* de *ShapeList* para registrar no servidor o seu interesse em receber *callbacks*. O método *register* retorna um valor inteiro (uma *callbackId*) que serve como identificador desse registro. Quando o cliente tiver terminado, ele deve chamar *deregister* para informar o servidor que não quer mais as *callbacks*. O servidor é responsável por manter uma lista dos clientes interessados e por notificar todos eles sempre que seu número de versão aumentar.

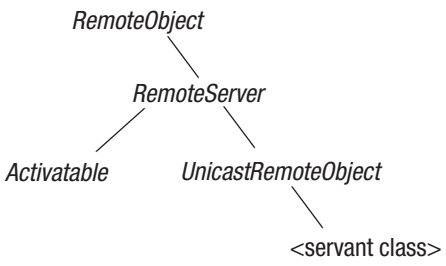

Figura 5.21 Classes que suportam a RMI Java.

5.5.2 Projeto e implementação da RMI Java

O sistema RMI Java original usava todos os componentes mostrados na Figura 5.15. No entanto, no Java 1.2, os recursos de reflexão foram usados para fazer um despachante genérico e para evitar a necessidade de esqueletos. Antes do J2SE 5.0, os *proxies* clientes eram gerados por um compilador, chamado de *rmic*, a partir das classes de servidor compiladas (e não das definições das interfaces remotas). Contudo, essa etapa não é mais necessária, com as versões recentes do J2SE, que contêm suporte para a geração dinâmica de classes *stub* em tempo de execução.

Uso de reflexão • A reflexão é usada para passar informações, nas mensagens de requisição, sobre o método a ser invocado. Isso é obtido por meio da classe *Method* no pacote de reflexão. Cada instância de *Method* representa as características de um método em particular, incluindo sua classe, os tipos de seus argumentos, o valor de retorno e as exceções. A característica mais interessante dessa classe é que uma instância de *Method* pode ser invocada em um objeto de uma classe, por intermédio de seu método *invoke*. O método *invoke* exige dois argumentos: o primeiro especifica o objeto que vai receber a invocação e o segundo é um vetor de *Object* contendo os argumentos. O resultado é retornado como tipo *Object*.

Voltando ao uso da classe *Method* na RMI: o *proxy* precisa empacotar informações sobre um método e seus argumentos na mensagem de requisição. Para o método, ele empacota um objeto da classe *Method*. Ele coloca os argumentos em um vetor de elementos *Object* e, depois, empacota esse vetor. O despachante desempacota o objeto *Method* e seus argumentos no vetor de elementos *Object* da mensagem de *pedido*. Como sempre, a referência de objeto remoto do destino terá sido desempacotada e a referência de objeto local correspondente, obtida do módulo de referência remota. Então, o despachante chama o método *invoke* do objeto *Method*, fornecendo o destino e o vetor de valores de argumento. Quando o método tiver sido executado, o despachante empacotará o resultado ou as exceções na mensagem de *resposta*. Assim, o despachante é genérico – isto é, o mesmo despachante pode ser usado por todas as classes de objeto remoto e nenhum esqueleto é exigido.

Classes Java que suportam RMI • A Figura 5.21 mostra a estrutura de herança das classes que suportam servidores Java RMI. A única classe que o programador necessita conhecer é *UnicastRemoteObject*, que toda classe servente precisa estender. A classe *UnicastRemoteObject* estende uma classe abstrata chamada *RemoteServer*, a qual fornece versões abstratas dos métodos exigidos pelos servidores remotos. *UnicastRemoteObject* foi o primeiro exemplo de *RemoteServer* a ser fornecido. Outro método interessante, chamado *Activatable*, está disponível para fornecer objetos passíveis de ativação. Há também classes para fornecer suporte à replicação de objetos. A classe *RemoteServer* é uma subclasse de *RemoteObject* que tem uma variável de instância contendo a referência de objeto remoto e fornece os seguintes métodos:

equals: este método compara referências de objeto remoto;

toString: este método fornece o conteúdo da referência de objeto remoto como um *String*;

readObject, writeObject: estes métodos desserializam/serializam objetos remotos.

Além disso, o operador *instanceOf* pode ser usado para testar objetos remotos.

5.6 Resumo

Este capítulo discutiu três paradigmas da programação distribuída – protocolos de requisição-resposta, chamadas de procedimento remoto e invocação a método remoto. Todos esses paradigmas fornecem mecanismos para entidades distribuídas independentes (processos, objetos, componentes ou serviços) se comunicarem diretamente.

Os programas de requisição-resposta fornecem suporte leve e mínimo para a computação cliente-servidor. Tais protocolos são frequentemente usados em ambientes onde as sobrecargas de comunicação devem ser minimizadas – por exemplo, em sistemas incorporados. Sua função mais comum é dar suporte para RPC ou RMI, conforme discutido a seguir.

A estratégia de chamada de procedimento remoto foi um avanço significativo nos sistemas distribuídos, fornecendo suporte de nível mais alto para os programadores, por estender o conceito de chamada de procedimento para operar em um ambiente de rede. Isso oferece importantes níveis de transparência nos sistemas distribuídos. Contudo, devido a suas diferentes características de falha e de desempenho e à possibilidade de acesso concorrente aos servidores, não é necessariamente uma boa ideia fazer as chamadas de procedimento remoto serem exatamente iguais às chamadas locais. As chamadas de procedimento remoto fornecem diversas semânticas de invocação, desde invocações *talvez* até a semântica *no máximo uma vez*.

O modelo de objeto distribuído é uma ampliação do modelo de objeto local usado nas linguagens de programação baseadas em objetos. Os objetos encapsulados formam componentes úteis em um sistema distribuído, pois o encapsulamento os tornam inteiramente responsáveis por gerenciar seus próprios estados, e as invocações a métodos locais podem ser estendidas para invocações remotas. Cada objeto em um sistema distribuído tem uma referência de objeto remoto (um identificador globalmente exclusivo) e uma interface remota que especifica quais de suas operações podem ser invocadas de forma remota.

As implementações de *middleware* da RMI fornecem componentes (incluindo *proxies*, esqueletos e despachantes) que ocultam aos programadores do cliente e do servidor os detalhes do empacotamento, da passagem de mensagem e da localização de objetos remotos. Esses componentes podem ser gerados por um compilador de interface. A RMI Java estende a invocação local em remota usando a mesma sintaxe, mas as interfaces remotas devem ser especificadas estendendo uma interface chamada *Remote* e fazendo cada método disparar uma exceção *RemoteException*. Isso garante que os programadores saibam quando fazem invocações remotas ou implementam objetos remotos, permitindo a eles tratar de erros ou projetar objetos convenientes para acesso concorrente.

Exercícios

5.1 Defina uma classe cujas instâncias representem as mensagens de requisição-resposta, conforme ilustrado na Figura 5.4. A classe deve fornecer dois construtores, um para mensagens de requisição e o outro para mensagens de resposta, mostrando como o identificador de requi-

sição é atribuído. Ela também deve fornecer um método para empacotar a si mesma em um vetor de bytes e desempacotar um vetor de bytes em uma instância. *página 188*

5.2 Programe cada uma das três operações do protocolo de requisição-resposta da Figura 5.3 usando comunicação UDP, mas sem adicionar quaisquer medidas de tolerância a falhas. Você deve usar as classes que definiu no capítulo anterior para referências de objeto remoto (Exercício 4.13) e acima para mensagens de requisição-resposta (Exercício 5.1). *página 187*

5.3 Forneça um esboço da implementação de servidor, mostrando como as operações *getRequest* e *sendReply* são usadas por um servidor que cria uma nova *thread* para executar cada requisição do cliente. Indique como o servidor copiará o *requestId* da mensagem de requisição na mensagem de resposta e como obterá o endereço IP e a porta do cliente. *página 187*

5.4 Defina uma nova versão do método *doOperation* que configure um tempo limite para a espera da mensagem de resposta. Após a expiração do tempo limite, ele retransmite a mensagem de requisição *n* vezes. Se ainda não houver nenhuma resposta, ele informará o chamador. *página 188*

5.5 Descreva um cenário no qual um cliente poderia receber uma resposta de uma chamada anterior. *página 187*

5.6 Descreva as maneiras pelas quais o protocolo de requisição-resposta mascara a heterogeneidade dos sistemas operacionais e das redes de computador. *página 187*

5.7 Verifique se as seguintes operações são *idempotentes*:

i) pressionar o botão "subir" (elevador);

ii) escrever dados em um arquivo;

iii) anexar dados em um arquivo.

É uma condição necessária para a idempotência o fato de a operação não estar associada a nenhum estado? *página 190*

5.8 Explique as escolhas de projeto relevantes para minimizar o volume de dados de resposta mantidos em um servidor. Compare os requisitos de armazenamento quando os protocolos RR e RRA são usados. *página 191*

5.9 Suponha que o protocolo RRA esteja em uso. Por quanto tempo os servidores devem manter dados de resposta não confirmados? Os servidores devem enviar a resposta repetidamente, em uma tentativa de receber uma confirmação? *página 191*

5.10 Por que o número de mensagens trocadas em um protocolo poderia ser mais significativo para o desempenho do que o volume total de dados enviados? Projete uma variante do protocolo RRA na qual a confirmação vá "de carona" (*piggyback*) – isto é, seja transmitida na mesma mensagem – na próxima requisição, onde apropriado e, caso contrário, seja enviada como uma mensagem separada. (Dica: use um temporizador extra no cliente.) *página 191*

5.11 Uma interface *Election* fornece dois métodos remotos:

> *vote:* este método possui dois parâmetros por meio dos quais o cliente fornece o nome de um candidato (um *string*) e o "número do votante" (um valor inteiro usado para garantir que cada usuário vote apenas uma vez). Os números dos votantes são alocados esparsamente a partir do intervalo de inteiros para torná-los difíceis de adivinhar.
>
> *result*: este método possui dois parâmetros com os quais o servidor fornece para o cliente o nome de um candidato e o número de votos desse candidato.

Quais dos parâmetros desses dois métodos são de *entrada* e quais são parâmetros de *saída*? *página 195*

5.12 Discuta a semântica de invocação que pode ser obtida quando o protocolo de requisição-resposta é implementado sobre uma conexão TCP/IP, a qual garante que os dados são distribuídos na ordem enviada, sem perda nem duplicação. Leve em conta todas as condições que causam a perda da conexão. *Seção 4.2.4 e página 198*

5.13 Defina a interface do serviço *Election* na IDL CORBA e na RMI Java. Note que a IDL CORBA fornece o tipo *long* para inteiros de 32 bits. Compare os métodos nas duas linguagens, para especificar argumentos de *entrada* e *saída*. *Figuras 5.8 e 5.16*

5.14 O serviço *Election* deve garantir que um voto seja registrado quando o usuário achar que depositou o voto.

Discuta o efeito da semântica *talvez* no serviço *Election*.

A semântica *pelo menos uma vez* seria aceitável para o serviço *Election* ou você recomendaria a semântica *no máximo uma vez*? *página 199*

5.15 Um protocolo de requisição-resposta é implementado em um serviço de comunicação com falhas por omissão para fornecer semântica de invocação *pelo menos uma vez*. No primeiro caso, o desenvolvedor presume um sistema assíncrono distribuído. No segundo caso, o desenvolvedor presume que o tempo máximo para a comunicação e a execução de um método remoto é *T*. De que maneira esta última suposição simplifica a implementação? *página 198*

5.16 Esboce uma implementação para o serviço *Election* que garanta que seus registros permaneçam consistentes quando ele é acessado simultaneamente por vários clientes. *página 199*

5.17 Suponha que o serviço *Election* seja implementado em RMI e deva garantir que todos os votos sejam armazenados com segurança, mesmo quando o processo servidor falha. Explique como isso pode ser conseguido no esboço de implementação de sua resposta para o Exercício 5.16. *páginas 213, 214*

5.18 Mostre como se usa reflexão Java para construir a classe *proxy* cliente para a interface *Election*. Forneça os detalhes da implementação de um dos métodos dessa classe, o qual deve chamar o método *doOperation* com a seguinte assinatura:

byte *[] doOperation (RemoteObjectRef o, Method m,* byte*[] arguments);*

Dica: uma variável de instância da classe *proxy* deve conter uma referência de objeto remoto (veja o Exercício 4.13). *Figura 5.3, página 224*

5.19 Mostre como se gera uma classe *proxy* cliente usando uma linguagem como C++, que não suporta reflexão, por exemplo, a partir da definição de interface CORBA dada em sua resposta para o Exercício 5.13. Forneça os detalhes da implementação de um dos métodos dessa classe, o qual deve chamar o método *doOperation* definido na Figura 5.3. *página 211*

5.20 Explique como se faz para usar reflexão Java para construir um despachante genérico. Forneça o código Java de um despachante cuja assinatura seja:

public void dispatch(Object target, Method aMethod, byte*[] args)*

Os argumentos fornecem o objeto de destino, o método a ser invocado e os argumentos desse método, em um vetor de bytes. *página 224*

5.21 O Exercício 5.18 exigia que o cliente convertesse argumentos *Object* em um vetor de bytes antes de ativar *doOperation*, e o Exercício 5.20 exigia que o despachante convertesse um vetor de bytes em um vetor de elementos *Object*, antes de invocar o método. Discuta a implementação de uma nova versão de *doOperation* com a seguinte assinatura:

Object [] doOperation (RemoteObjectRef o, Method m, Object[] arguments);

que usa as classes *ObjectOutputStream* e *ObjectInputStream* para comunicar as mensagens de requisição-resposta entre cliente e servidor por meio de uma conexão TCP. Como essas alterações afetariam o projeto do despachante? *Seção 4.3.2 e página 224*

5.22 Um cliente faz invocações a método remoto a um servidor. O cliente demora 5 milissegundos para computar os argumentos de cada requisição, e o servidor demora 10 milissegundos para processar cada requisição. O tempo de processamento do sistema operacional local para cada operação de envio ou recepção é de 0,5 milissegundos, e o tempo que a rede leva para transmitir cada mensagem de requisição ou resposta é de 3 milissegundos. O empacotamento ou desempacotamento demora 0,5 milissegundos por mensagem.

Calcule o tempo que leva para o cliente gerar e retornar duas requisições:

(i) se ele tiver só uma *thread*;

(ii) se ele tiver duas *threads* que podem fazer requisições concorrentes em um único processador.

Você pode ignorar os tempos de troca de contexto. Há necessidade de invocação assíncrona se os processos cliente e servidor forem programados com múltiplas *threads*? *página 213*

5.23 Projete uma tabela de objetos remotos que possa suportar coleta de lixo distribuída, assim como fazer a transformação entre referências de objeto local e remota. Dê um exemplo envolvendo vários objetos remotos e *proxies* em diversos *sites* para ilustrar o uso da tabela. Mostre as alterações na tabela quando uma invocação faz um novo *proxy* ser criado. Em seguida, mostre as alterações na tabela quando um dos *proxies* se torna inatingível. *página 215*

5.24 Uma versão mais simples do algoritmo de coleta de lixo distribuída, descrito na Seção 5.4.3, apenas invoca *addRef* no *site* onde está um objeto remoto, quando um *proxy* é criado, e *removeRef*, quando um *proxy* é excluído. Esboce todos os efeitos possíveis das falhas de comunicação e de processos no algoritmo. Sugira como superar todos esses efeitos, mas sem usar arrendamentos. *página 215*