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*