Objetos Remotos (RMI)

Periodicamente, a comunidade de programação começa a pensar em “objetos por toda parte” como a solução para todos os seus problemas. A idéia é ter uma família feliz de objetos colaborativos, que possa ser localizada em qualquer lugar. Esses objetos devem, é claro, se comunicar através de protocolos padrão através de uma rede. Por exemplo, você terá um objeto no cliente, onde o usuário pode preencher um pedido de dados. O objeto cliente envia uma mensagem para um objeto no servidor, contendo os detalhes do pedido. O objeto servidor reúne as informações solicitadas, talvez acessando um banco de dados ou se comunicando com objetos adicionais. Uma vez que o objeto servidor tenha a resposta para o pedido do cliente, ele a envia de volta para o mesmo. Como acontece em quase tudo na programação, esse plano contém muita propaganda, que pode obscurecer a utilidade do conceito.

Introdução aos Objetos Remotos: os Papéis do Cliente e do Servidor

Vamos voltar à idéia da reunião local de informações em um computador cliente e do envio das informações pela Internet para um servidor. Estamos supondo que um usuário em uma máquina local vai preencher um formulário de pedido de informações. Os dados do formulário são enviados para o servidor do fornecedor e o servidor processa o pedido e por sua vez, desejará enviar de volta informações de produto que o cliente possa ver.

No modelo cliente/servidor tradicional, o pedido é transformado em um formato intermediario (como pares nome/valor ou dados XML). O servidor analisa o formato do pedido, calcula a resposta e formata para transmissão ao cliente. O cliente então analisa a resposta e a apresenta para o usuário.

Mas se seus dados são estruturados, assim há um problema de codificação significativo: você precisa apresentar meios apropriados de transformação dos dados “para” e “do” formato de transmissão.

Qual seria uma solução possível? Bem, lembrando que os objetos enviando pedidos uns para os outros é o princípio da POO (Programação Orientada a Objetos), poderíamos colocar objetos em máquinas diferentes e fazê-los enviar mensagens diretamente uns para os outros. Vamos supor que o objeto cliente tenha sido escrito na linguagem de programação Java, de modo que, teoricamente, ele pode ser executado em qualquer lugar. Para os objetos servidores, existem duas possibilidades óbvias:

- O objeto servidor não foi escrito na linguagem de programação Java (porque se trata de um objeto legado).
- O objeto servidor foi escrito na linguagem de programação Java.

A primeira situação exige de nós termos uma maneira pela qual os objetos falem

entre si independentemente da linguagem em que foram originalmente escritos. Se você pensar sobre isso, concordará conosco que, mesmo a possibilidade teórica disso é uma façanha surpreendente. Como pode, o que é em última alálise, uma seqüência de bytes escrita em uma linguagem arbitrária, da qual podemos não ter nenhum conhecimento, nos dizer quais serviços oferece, a quais mensagens responde? É claro que fazer isso funcionar na prática não é fácil, mas a idéia é elegante. O padrão CORBA, do OMG – Object Management Group – (www.ogm.org) define um mecanismo comum para troca de dados e descoberta de serviços.

A idéia básica é que delegamos a tarefa de descobrir essas informações e ativar todos os serviços solicitados para um agente chamado ORB (Object Request Broker, agente de requisição de objetos). Você pode considerar um ORB como uma espécie de tradutor universal para comunicação entre objetos. Os objetos não falam diretamente uns com os outros. Eles sempre usam um agente de objetos para negociar entre eles. Os ORBs estão localizados ao longo da rede e é importante que eles se comuniquem uns com os outros. A maioria dos ORBs segue a especificação definida pelo OMG para comunicação entre ORBs. Essa especificação é chamada de IIOP (Internet Inter-ORB Protocol, protocolo inter-ORB da Internet).

O CORBA é uma linguagem completamente neutra. Os programas cliente e servidor podem ser escritos em C++, na linguagem de programação Java ou em qualquer outra linguagem vinculada ao CORBA. Você usa uma IDL (Interface Definition Language, linguagem de definição de interface) para especificar as assinaturas das mensagens e os tipos dos dados que seus objetos podem enviar e entender. (As especificações IDL são muito parecidas com as interfaces na linguagem de programação Java; na verdade, você pode considerá-las como interfaces de definição que os objetos que estão em comunicação devem suportar. Um recurso interessante desse modelo é que você pode fornecer uma especificação IDL de um objeto legado existente e depois acessar seus serviços através do ORB, mesmo que ele tenha sido escrito muito antes que o primeiro ORB tivesse aparecido.) Existem muitas pessoas que acreditam que o CORBA se tornará muito importante em breve e que a linguagem de programação Java é uma opção excelente para a implementação de clientes e servidores CORBA. Entretanto, falando francamente, o CORBA tem tido uma reputação - às vezes merecida – de baixo desempenho, implementações complexas e problemas de intercâmbio.

Se os dois objetos que estão se comunicando forem escritos na linguagem de programação Java, a generalidade e complexidade totais do CORBA não são exigidas. A Sun desenvolveu um mecanismo mais simples, chamado RMI (Remote Method Ivocation, chamada de método remoto), especificamente para a comunicação entre aplicativos Java.

Chamada de Métodos Remotos

O mecanismo RMI permite que você faça algo que parece simples. Se você tem

acesso a um objeto em uma máquina diferente, pode chamar métodos do objeto remoto. É claro que os parâmetros do método devem ser enviados para a outra máquina de alguma forma, que o objeto deve ser informado para que execute o método e que o valor de retorno deve ser enviado de volta. O RMI trata de todos esses detalhes.

Por exemplo, o cliente que está buscando informações de produto pode consultar um objeto Warehouse no servidor. Ele chama um método remoto, find, que tem um parâmetro: um objeto Customer. O método find retorna um objeto para o cliente: o objeto informações de produto.

Na terminologia RMI, o objeto cujo método faz a chamada remota é denominado objeto cliente. O objeto remoto é chamado de objeto servidor. É importante lembrar que a terminologia cliente/servidor aplica-se apenas a uma chamada de método simples. O computador que está executando o código na linguagem de programação Java, que chama o método remoto, é o cliente dessa chamada e o computador que contém o objeto que processa a chamada é o servidor dessa chamada. É Totalmente possível que os papéis sejam invertidos em algum momento futuro. O servidor de uma chamada anterior pode se tornar o cliente, quando chamar um método remoto em um objeto residente em outro computador.

“Stubs” e Reunião de Parâmetros

Quando o código cliente quer chamar um método remoto em um objeto remoto, na verdade ele chama um método normal da linguagem de programação Java, que é encapsulado em um objeto substituto chamado stub. O stub reside na máquina cliente e não no servidor. O stub empacota, como um bloco de bytes, os parâmetros usados no método remoto. Esse empacotamento usa uma codificação independente de dispositivo para cada parâmetro. Por exemplo, os números são sempre enviados na ordenação de bytes big-endian. Os objetos são codificados com o mecanismo de serialização. O processo de codificação dos parâmetros é chamado reunião (marshalling) de parâmetros. O objetivo da reunião de parâmetros é converter os parâmetros para um formato conveniente para transporte de uma máquina virtual para outra.

Resumindo: o método de stub na cliente constrói um bloco de informações que consiste em:

- Um identificador do objeto remoto a ser usado;
- Uma descrição do método a ser chamado;
- Os parâmetros reunidos.

Após stub envia essas informações para o servidor. No lado do servidor, um objeto receptor executa as seguintes ações para cada chamada de método remota:

- Ele separa as parâmetros.

- Ele localiza o objeto a ser chamado.
- Ele chama o método desejado.
- Ele captura e combina o valor de retorno ou a execução da chamada.
- Ele envia um pacote composto dos dados de retorno reunidos de volta para o stub no cliente.

O stub cliente separa o valor de retorno ou a execução do servidor. Esse valor se torna o valor de retorno da chamada do stub. Ou então, se o método remoto lançou uma execução, o stub a levanta novamente no espaço de processo do chamador.

Esse processo é obviamente complexo, mas a boa notícia é que ele é completamente automático e até certo ponto, transparente para o programador. Além disso, os desenvolvedores dos objetos Java remotos tentaram de todas as formas obter objetos remotos com a mesma “aparência e comportamento” dos objetos locais.

A sintaxe para uma chamada de método remoto é igual à de uma chamada local. Se centralWarehouse for um objeto stub de um objeto warehouse central em uma máquina remota e getQuantity for o método que você deseja chamar nela, então uma chamada normal é como segue:

```
centralWarehouse.getQuantity(“SuperSucker 100 Vacuun Cleaner”);
```

O código cliente sempre usa variáveis de objeto cujo tipo é uma interface para acessar objetos remotos. Por exemplo, associada a essa chamada haveria uma intercace:

```
interface Warehouse {
  public int getQuantity(String description) throws
                RemoteException;
  ...
}
```

Uma declaração de objeto para uma variável que implementará a interface é:

```
Warehouse centralWarehouse = ...;
```

É claro que as interfaces são entidades abstratas que apenas informam quais métodos podem ser chamados, junto com suas assinaturas. As variáveis cujo tipo é uma interface devem ser sempre ligadas a um objeto real de algum tipo. Ao chamar métodos remotos, a variável objeto se refere a um objeto stub. O programa cliente não sabe realmente o tipo desses objetos. As classes de stub e os objetos associados são criados automaticamente.

Embora os desenvolvedores tenham feito um bom trabalho de ocultar para o programador muitos detalhes da chamada de método remoto, várias técnicas e advertências ainda devem ser dominadas.

Carregamento de Classes Dinâmico

Quando você passa um objeto remoto para outro programa ou como um parâmetro ou como um valor de retorno de um método remoto, esse programa deve tratar do objeto stub associado. Isto é, ele deve ter o código da classe stub. Os métodos stub não realizam muito trabalho interessante. Eles apenas reúnem e separam os parâmetros e depois entram em contato com o servidor para chamadas de método. É claro que eles fazem todo esse trabalho de forma transparente para o programador.

Além disso, as classes de parâmetros, valores de retorno e objetos de exceção talvez também precisem ser carregados. Esse carregamento pode ser mais complexo do que você poderia pensar. Por exemplo, você pode declarar um método remoto com certo tipo de retorno conhecido do cliente, mas o método retorna realmente um objeto de uma subclasse conhecida do cliente. O carregador de classes carregará então essa classe derivada.

Embora não sejam muito atraentes, as classes stub devem estar disponíveis para o programa cliente que está em execução. Uma maneira óbvia de tornar essas classes disponíveis é colocá-las no sistema de arquivos local. Entretanto, se o programa servidor for estendido e novas classes de tipo de retorno e exceções forem carregadas, então será difícil manter a atualização do cliente.

Por isso, os clientes RMI podem carregar classes stub de outro lugar, automaticamente. O processo é semelhante ao processo de carregamento de classes que ocorre em um navegador.

Quando um programa carrega código novo de outro local da rede, há um problema de segurança. Por isso, você precisa usar um gerenciador de segurança em aplicativos clientes RMI. Trata-se de um mecanismo de segurança que protege o programa contra vírus no código do stub. Para aplicativos especializados, os programadores podem implementar seus próprios carregadores de classe e gerenciadores de segurança, mas aqueles fornecidos pelo sistema RMI bastam para utilização normal.

Estabelecendo as chamadas de Métodos Remotos

Executar mesmo o exemplo mais simples de objeto remoto exige muito mais configuração do que executar uma applet ou programa independente. Você deve executar programas nos computadores servidor e cliente. As informações de objeto necessárias devem ser separadas nas interfaces do lado do cliente e nas implementações no lado do servidor. Existe também um mecanismo de pesquisa especial que permite ao cliente localizar objetos no servidor.

Para começar com a codificação real, veremos cada um desses requisitos, usando um exemplo simples. Em nosso primeiro exemplo, geramos dois objetos de tipo Product no computador servidor. Executamos um programa em um computador cliente, que

localiza e consulta esses objetos.

Interfaces e Implementações

Seu programa cliente precisa manipular objetos de servidor, mas ele não tem realmente cópias deles. Os próprios objetos residem no servidor. O código cliente ainda deve saber o que pode fazer com esses objetos. Seus recursos são expressos em uma interface que é compartilhada entre o cliente e o servidor e portanto, residem simultaneamente nas duas máquinas.

```
// compartilhada pelo cliente e pelo servidor
interface Product extends Remote {
  public String getDescription() throws RemoteException;
}
```

Assim como nesse exemplo, todas as interfades de objetos remotos devem estender a interface Remote definida no pacote java.rmi. Todos os métodos nessas interfaces também devem declarar que lançarão uma RemoteException. O motivo da declaração é que as chamadas de método remotas são inerentemente menos confiáveis do que as chamadas locais – é sempre possível que uma chamada remota falhe. Por exemplo, o servidor ou a conexão de rede pode estar temporariamente indisponível ou pode haver um problema de rede. Seu código cliente deve estar preparado para tratar com essas possibilidades. Por isso, a linguagem de programação Java o obriga a capturar a RemoteException com toda a chamada de método remoto e especificar a ação apropriada a ser executada quando a chamada não tiver sucesso.

O cliente acessa o objeto servidor através de um stub que implementa essa interface.

```
Product p = ...;
// veja a seguir como o cliente recebe uma referência
// de stub para um objeto remoto
String d = p.getDescription();
System.out.println(d);
```

Em seguida, no lado do servidor, você deve implementar a classe que efetivamente executa os métodos anunciados na interface remota.

```
// Servidor
public class ProductImpl extends UnicastRemoteObject implements
                  Product {
  private String descr;

  public ProductImpl(String d) throws RemoteException {
    descr = d;
```

```
    }

    public String getDescription() throws RemoteException {
      return “I am a “ + descr + “. Buy me!”;
    }
  }
```

Essa classe tem um único método, getDescription, que pode ser chamado a partir do cliente remoto.

Você pode saber que a classe é um servidor de métodos remotos, porque ela estende UnicastRemoteObject, que é uma classe concreta da plataforma Java que torna os objetos acessíveis remotamente.

Todas as classes de servidor devem estender a classe RemoteServer do pacote java.rmi.server, mas RemoteServer é uma classe abstrata que define apenas os mecanismos básicos para a comunicação entre objetos servidores e seus stubs remotos. A classe UnicastRemoteObject que acompanha o RMI estende a classe abstrata RemoteServer e á concreta portanto, você pode usá-la sem escrever nenhum código.

Um objeto UnicastRemoteObject reside em um servidor. Ele deve estar ativo quando um serviço for solicitado e deve ser acessado através do protocolo TCP/IP. Essa é a única classe de servidor disponível na versão corrente do pacote RMI. A Sun ou outros fornecedores podem, no futuro, criar outras classes para uso pelos servidores de RMI. Por exemplo, a Sun está falando a respeito de uma classe MulticastRemoteObject para objetos que são duplidados em vários servidores. Outras possibilidades são para objetos que são ativados segundo a necessidade e aqueles que podem usar outros protocolos de comunicação, como UDP.

Sem sufixo (por exemplo, Product)	Uma interface remota
Sufixo Impl (por exemplo, ProductImpl)	Uma classe de servidor implementando essa interface
Sufixo Server (por exemplo, ProductServer)	Um programa servidor que ria objetos servidores
Sufixo Client (por exemplo, ProductClient)	Um programa cliente que chama métodos remotos
Sufixo _Stub (por exemplo, ProductImpl_Stub)	Uma classe de stub que é gerada automaticamente pelo programa rmic
Sufixo _Skel (por exemplo, ProductImpl_Skel)	Uma classe esqueleto que é gerada automaticamente pelo programa rmic; necessária para o JDK 1.1

Convenções de atribuição de nomes para classes RMI

Você precisa gerar stubs para a classe ProductImpl. Lembre-se de que os stubs são as classes que reúnem (codificam e enviam) os parâmetros e os resultados das chamadas de método através da rede. O programador nunca usa essas classes diretamente. Além disso, elas não precisam ser escritas à mão. A ferramenta rmic as gera automaticamente, como no exemplo a seguir.

```
rmic -v1.2 ProductImpl
```

Essa chamada à ferramenta rmic gera um arquivo de classe ProductImpl_Stub.class. Se sua classe está em um pacote, você deve chamar rmic com o nome completo do pacote.

Se seu cliente usa JDK 1.1, em vez disso, você deve chamar:

```
rmic ProductImpl
```

Desta forma, dois arquivos são gerados: o arquivo de stub e um segundo arquivo de classe, chamado ProductImpl_Skel.class. Esse arquivo de “esqueleto” não é mais necessário na plataforma Java 2.

Localizando Objetos Servidores

Para acessar um objeto remoto existente no servidor, o cliente precisa de um objeto stub local. Como o cliente solicita tal stub? O método mais comum é chamar um método remoto de outro objeto servidor e obter um objeto stub com valor de retorno. Existe, entretanto, um problema do ovo e da galinha, aqui. O primeiro objeto servidor precisa ser localizado de algum outro modo. A biblioteca RMI da Sun fornece um serviço de registro de partida (bootstarp registry service) para localizar o primeiro objeto servidor.

Um programa servidor registra objetos com o serviço de registro de partida e o cliente recupera stubs para esses objetos. Você registra um objeto servidor fornecendo ao serviço de registro de partida uma referência ao objeto e um nome. O nome é uma string que (espera-se) é única.

```
// Servidor
ProductImpl p1 = new ProductImpl(“Blackwell Toaster”);
Naming.bind(“toaster”, p1);
```

O código cliente obtém um stub para acessar esse objeto servidor, especificando o nome do servidor e o nome do objeto, da seguinte maneira:

```
// Cliente
Product p =new (Product)Naming.lookup(
                  “rmi://seuservidor.com/toaster”);
```

As URLs de RMI começam com rmi:// e são seguidos de um servidor, um número de porta opcional, outra barra e do nome do objeto remoto. Outro exemplo é:

```
rmi://localhost:99/central_warehouse
```

Como padrão, o número de porta é 1099.

Por questões de segurança, um aplicativo pode ligar, desligar ou religar novamente referências de objetos de registro apenas se estiver executando no mesmo host do registro. Isso evita que clientes hostis alterem informações de registro. Entretanto, qualquer cliente pode pesquisar objetos.

O próximo exemplo mostra um programa servidor completo que registra dois objetos Procuct sob os nomes toaster e microwave

Vide os códigos ProductServer.java, ProductImpl.java e Product.java

Iniciando o Servidor

Nosso programa servidor ainda não está pronto para funcionar. Como ele usa o registro de RMI de partida, esse serviço deve estar disponível. Para iniciar o registro de RMI você executa a instrução a seguir em um prompt do DOS ou a partir da caixa de diálogo Executar.

```
start rmiregistry
```

Agora, você está pronto para iniciar o servidor:

```
start java ProductServer
```

Se você executar o programa servidor como:

```
java ProductServer
```

Ele nunca terminará normalmente. Isso parece estranho – afinal, o programa apenas cria dois objetos e os registra. Na verdade, a função main termina imediatamente após o registro, conforme você poderia esperar. Mas quando você cria um objeto de uma classe que estende UnicastRemoteObject, é iniciada uma linha de execução separada, que mantém o programa ativo indefinidamente. Assim, o programa fica funcionando para permitir que os clientes se conectem com ele.

Antes de escrevermos o programa-cliente, vamos verificar o que aconteceu no registro dos objetos remotos. A classe Naming tem um método list que retorna uma lista de todos os nomes correntemente registrados. O Exemplo a seguir trata de um programa que lista os nomes que estão no registro.

Em nosso caso, sua saída é:

```
rmi:/toaster
rmi:/microwave
```

Vide o código ShowBindings.java.

O Lado do Cliente

Agora, podemos escrever o programa-cliente que pede que cada objeto produto

recentemente registrado imprima sua descrição.

Os programas-cliente que usam RMI devem instalar um gerenciados de segurança para controlar as atividades dos stubs carregados dinamicamente. O RMISecurityManager é um gerenciador de segurança assim. Você o instala com a instrução:

```
System.setSecurityManager(new RMISecurityManager());
```

O cliente simplemente obtém referências a dois objetos Product no registro RMI e chama o método getDescription nos dois objetos.

Vide o código ProductClient.java

Executando o Cliente

Por definição, o RMISecurityManager impede todo o código do programa de estabelecer conexões de rede. Mas o programa precisa estabelecer conexões de rede:

- para chegar ao registro RMI
- para contatar os objetos servidores

Para permitir que o cliente se conecte com o registro RMI e com o objeto servidor, você precisa fornecer um arquivo de política. Aqui está um arquivo de política que permite a um aplicativo estabelecer qualquer conexão de rede com uma porta com número mínimo 1024. (A porta RMI é 1099, como padrão e os objetos servidores também usam portas >= 1024.)

```
grant {
  permission java.netSocketPermission “*:1024-65535”, “connect”;
};
```

Quando você executar o programa cliente, deverá especificar o arquivo de política:

```
java -Djava.security.policy=cliente.policy ProductClient
```

Se o registro RMI e o servidor ainda estiverem em execução, você poderá prosseguir com a execução do cliente. Ou então, se você quiser começar desde o início, interrompa o registro RMI e o servidor. Em seguida, siga estes passos:

1. Compile os arquivos-fonte das classes interface, implementação, cliente e servidor.

```
javac Product*.java
```

2. Execute rmic na classe de implementação.

```
rmic –v1.2 ProductImpl
```

3. Inicie o registro RMI.

```
start rmiregistry
```

4. Inicie o servidor.

```
start java ProductServer
```

5. Execute o cliente.

```
java -Djava.security.policy=cliente.policy
ProductClient
```

O programa simplemente imprime

```
I am a Blackwell Toaster. Buy me!
I am a ZapXpress Microwave Oven. Buy me!
```

Essa saída não parece causar muito impacto, mas considere o que ocorre nos bastidores, quando a linguagem Java executa a chamada ao método getDescription. O programa cliente tem uma referência a um stub que é obtido do método lookup. Ele chama o método getDescription, o qual envia uma mensagem de rede para um objeto receptor no lado do servidor. O objeto receptor chama o método getDescription no objeto ProductImpl localizado no servidor. Esse método gera uma string. O receptor envia essa string pela rede. O stub a recebe e a retorna como resultado.

Preparando a Distribuição

Distribuir um aplicativo que usa RMI pode ser complicado, pois muitas coisas podem dar errado e porque as mensagens de erro que você obtém quando algo dá errado são muito deficientes. Verificamos que a distribuição local vale a pena. Nesta etapa preparatória, separe os arquivos de classe em três subdiretórios:

```
server
download
client
```

O diretório server contém todos os arquivos que são necessários para executar o servidor. Posteriormente, você moverá esses arquivos para a máquina que está executando o processo servidor. Em nosso exemplo, o diretório server contém os seguintes arquivos:

```
server:
  ProductServer.class
  ProductImpl.class
  Product.class
  ProductImpl_Stub.class
```

O diretório download contém os arquivos de classe que serão carregados no cliente, assim como as classes das quais dependem. Em nosso exemplo, você deve incluir as seguintes classes no diretório download:

```
download:
```

```
ProductImpl_Stub.class
Product.class
```

Se seu programa foi escrito com JDK 1.1, então você também precisa forcecer as classes de esqueleto (como ProductImpl_Skel.class).

Finalmente, o diretório client contém os arquivos necessários para iniciar o cliente. São eles:

```
client:
  ProductClient.class
  Product.class
  client.policy
```

Você vai distribuir esses arquivos no computador cliente.

Agora, você tem todos os arquivos de classe particionados corretamente e pode testar se todos eles podem ser carregados.

Vamos supor que você pode iniciar um servidor Web em seu computador. Alternativamente, você pode obter um servidor leve a partir do endereço ftp://java.sun.com/pub/jdk1.1/rmi/class-server.zip. Esse servidor não é um servidor Web completo, mas é mais fácil de instalar e tem funcionalidade suficiente para servir arquivos de classe.

Primeiro, mova o diretório download para o diretório de documentos da Web do servidor Web.

Depois, edite o arquivo cliente.policy. Ele deve fornecer ao cliente as seguintes permissões:

- Conectar-se com portas 1024 e acima para atingir o registro RMI e as implementações de servidor
- Conectar-se com a porta HTTP (normalmente 80) para carregar os arquivos de classe de stub

Altere o arquivo para o seguinte:

```
grant {
  permission java.net.SocketPermission “*:1024-6535”, “connect”;
  permission java.net.SocketPermission “*:80”, “connect”;
};
```

Finalmente, você está pronto para testar sua configuração.

1. Inicie o servidor Web.
2. Inicie um novo shell. Certifique-se de que o caminho de classes não esteja configurado para nada. Mude para um diretório que não contenha nenhum arquivo

de classe. Em seguida, inicie o registro RMI.

3. Inicie um novo shell. Mude para o diretório server. Inicie o servidor, fornecendo uma URL para o diretório download, como o valor da propriedade java.rmi.server.codebase:

```
start java -Djava.rmi.server.codebase=
http://localhost/download/ProductServer
```

4. Vá para o diretório client. Inicie o cliente, fornecendo o arquivo de política como o valor da propriedade java.security.policy:

```
java -Djava.security.policy=client.Policy
ProductClient
```

Se o servidor e o cliente foram iniciados sem problemas, então você está pronto para passar para a próxima etapa e distribuir as classes em um cliente e servidor separados. Se não, você precisa fazer alguma otimização.

Distribuindo o Programa

Agora que você já testou a distribuição de seu programa, está pronto para distribuí-lo para os clientes e servidores reais.

Mova as classes do diretório download para o servidor Web. Certifique-se de usar essa URL quando iniciar o servidor. Mova as classes do diretório server para seu servidor e inicie o registro RMI e o servidor.

Sua configuração de servidor está agora finalizada, mas você precisa fazer duas alterações no cliente. Primeiro, edite o arquivo de política e substitua o * pelo nome do seu servidor:

```
grant {
  permission java.net.SocketPermission “seuservidor.com:1024-
                6535”, “connect”;
  permission java.net.SocketPermission “seuservidor.com:80”,
                “connect”;
};
```

Por fim, substitua localhost na URL de RMI do programa-cliente pelo servidor real.

```
String url = “rmi://seuservidor.com/”;
Product c1 = (Product)Naming.lookup(url + “toaster”);
```

Em seguida, recompile o cliente e experimente-o. Se tudo funcionar, então você merece os parabéns. Se não, talvez você ache o quadro a seguir útil. Ele contém uma lista de conferência de vários problemas que podem surgir com freqüência, ao se tentar fazer o RMI funcionar.

Lista de conferência de Distribuição de RMI
A distribuição de RMI é complicada porque ou ela funciona ou falha com uma mensagem de erro muito estranha. A julgar pelo tráfego na lista de discussão de RMI, no endereço http://archives.java.sun.com/archives/rmi-users.html, muitos programadores fracassam inicialmente. Se você também falhar, talvez ache útil verificar os problemas a seguir. Encontramos cada um deles errado pelo menos uma vez durante os testes. • Você colocou os arquivos de classe stub no diretório server? • Você colocou os arquivos de classe de interface nos diretórios download e client? • Ao se iniciar o rmiregistry, CLASSPATH estava desconfigurada? O diretório corrente estava sem arquivos de classe? • Você usa um arquivo de política ao iniciar o cliente? O arquivo de política contém os nomes de servidor corretos (ou * para se conectar com qualquer host)? • Se você usa uma URL file: para teste, especificou o nome de arquivo correto no arquivo de política? Ele termina em /- ou \\-? Você se lembrou de usar \\ para nomes de arquivo do Windows? • Sua URL de codebase termina com uma barra? Finalmente, note que o registro RMI se recorda de todos os arquivos de classe que encontrou. Se você continuar removendo vários arquivos de classe para testar quais arquivos são realmente necessários, certifique-se de reiniciar o rmiregistry a cada vez.

Passagem de Parâmetros em Métodos Remotos

Freqüentemente, você quer passar parâmetros para objetos remotos. Será explicada algumas das técnicas para isso – junto com algumas armadilhas.

Passando Objetos Não-Remotos

Quando um objeto remoto é passado do servidor para o cliente, este recebe um stub. Usando o stub, ele pode manipular o objeto servidor, chamando métodos remotos. O objeto, entretanto, permanece no servidor. Também é possível passar e retornar quaisquer objetos através de uma chamada de método e não apenas aqueles que implementam a interface Remote. Por exemplo, o método getDescription da seção anterior retornava um objeto String. Essa string foi criada no servidor e teve de ser transportada para o cliente. Como String não implementa a interface Remote, o cliente não pode retornar um objeto stub de string. Em vez disso, o cliente recebe uma cópia da string. Assim, após a chamada, o cliente tem o seu próprio objeto String para trabalhar. Isso significa que não há necessidade de mais conexões com qualquer objeto no servidor para tratar dessa string.

Quando um objeto que não é remoto precisa ser transportado de uma máquina virtual Java para outra, a máquina virtual Java faz uma cópia e a envia através da conexão de rede. Essa técnica é muito diferente da passagem de parâmetros em um método local. Quando você passa objetos em um método local ou os retorna como

resultados do método, apenas referências de objeto são passadas. Entretanto, as referências de objeto são endereços de memória de objetos na máquina virtual Java local. Essa informação não tem significado para uma maquina virtual Java diferente.

Não é difícil imaginar como uma cópia de uma string pode ser transportada através de uma rede. O mecanismo RMI também pode fazer cópias de objetos mais complexos, desde que eles possam ser serializáveis. O RMI usa o mesanismo de serialização para enviar objetos através de uma conexão de rede. Isso significa que apenas as informações de todas as classes que implementam a interface Serializable podem ser copiadas. O programa a seguir mostra a cópia de parâmetros e valores de retorno em ação. Esse programa é um aplicativo simples que permite ao usuário procurar um presente. No cliente, o usuário executa um programa que reúne informações sobre o destinatário do presente, neste caso, idade, sexo e hobbies.

Um objeto de tipo Customer é então enviado para o servidor. Como Customer não é um objeto remoto, uma cópia do objeto é feita no servidor. O programa servidor envia de volta um vetor de produtos. O vetor contém os produtos que correspondem ao perfil do cliente e sempre contém um item adicional. Novamente, Vector não é uma classe remota; portanto, o vetor é copiado do servidor para seu cliente. O mecanismo de serialização faz cópias de todos os objetos que são referenciados dentro de um objeto copiado. Em nosso caso, ele faz uma cópia também de todas as entradas vetor. Acrescentamos uma complexidade extra: as entradas são, na verdade objetos Product remotos. Assim, o destinatário recebe uma cópia do vetor, preenchida com objetos stub, para os produtos do servidor.

Resumindo, os objetos remotos são passados através a rede como stubs. Os objetos não-remotos são copiados. Tudo isso é automático e não exige intervenção do programador.

Quando o código chama um método remoto, o stub faz um pacote que contém cópias de todos os valores de parâmetros e o envia para o servidor, usando o mecanismo de serialização de objetos para reunir os parâmetros. O servidor os separa. Naturalmente, o processo pode ser muito lento – especialmente quando os objetos de parâmetros são grandes.

Vamos ver o programa completo. Primeiro, temos as interfaces dos serviços de produto e armazém (Warehouse), como se vê nos códigos Product.java e Warehouse.java.

O código ProductImpl.java mostra a implementação do serviço de produtos. Os produtos armazenam uma descrição, uma faixa de idade, o sexo a quem se destina o produto (masculino, feminino ou ambos) e o hobby correspondente. Note que essa classe implementa o método getDescription anunciado na interface Product e que também implementa outro método, match, que não faz parte dessa interface. O método match é

um exemplo de método local, um método que pode ser chamado apenas a partir do programa local e não remotamente. Como o método match é local, ele não precisa estar preparado para lançar uma RemoteException.

O código Customer.java contém o código da classe Customer. Note, mais uma vez, que Customer não é uma classe remota – nenhum de seus métodos pode ser executado remotamente. Entretando, a classe é serializável. Portanto, os objetos dessa classe podem ser transportados de uma máquina virtual para outra.

O código WarehouseImpl.java mostra a implementação do serviço de armazém. Assim como a classe ProductImpl, a classe WarehouseImpl tem métodos locais e remotos. O método add é local; ele é usado pelo servidor para incluir produtos no armazém. O método find é remoto; ele é usado para localizar itens no armazém.

Para mostrar que o objeto Customer é realmente copiado, o método find da classe WarehouseImpl limpa o objeto customer que recebe. Quando o método remoto retorna, WarehouseClient apresenta o objeto customer que enviou para o servidor, Conforme você verá, esse objeto não mudou. O servidor limpou apenas sua cópia. Neste caso, a operação clear não tem nenhuma razão útil, a não ser demonstrar que objetos locais são copiados quando são passados como parâmetros.

Em geral, os métodos das classes de servidor, como ProductImpl e WarehouseImpl, dever ser sincronizados. Então, é possível que vários stubs de cliente façam chamadas simultâneas a um objeto servidor, mesmo que alguns dos métodos mudem o estado do servidor. No código WarehouseImpl.java, sincronizamos os métodos da classe WarehouseImpl porque é possível que os métodos add local e find remoto sejam chamados simultaneamente. Não sincronizamos os métodos da classe ProductImpl porque os objetos servidor de produto não mudam seu estado. O código WarehouseServer.java mostra o programa servidor que cria um objeto warehouse e o registra no serviço de registro de partida.

O código WarehouseClient.java mostra o código do cliente. Quando o usuário á um clique no botão “Submit”, um novo objeto customer é gerado e passado para o método find remoto. Depois, o registro de cliente é apresentado na área de texto (para provar que a chamada de clear no servidor não o afetou). Finalmente, as descrições dos produtos retornados no vetor são incluídas na área de texto. Note que cada chamada de getDescription é, novamente, uma chamada de método remota.

Passando Objetos Remotos

A passagem de objetos do servidor para o cliente é simples. O cliente recebe um objeto stub e depois o salva em uma variável objeto cujo tipo é o mesmo da interface remota. O cliente pode agora acessar o objeto real no servidor através da variável. O cliente pode copiar essa variável em sua própria máquina local – todas essas cópias são

simplemente referências ao mesmo stub. É importante notar que apenas as interfaces remotas podem ser acessadas através do stub. Uma interface remota é qualquer interface que estenda Remote. Todos os métodos locais são inacessíveis através do stub. (Um método local é qualquer método que não seja definido em uma interface remota.) Os métodos locais podem ser executados apenas na máquina virtual que contém o objeto real.

Em seguida, são gerados os stubs apenas das classes que implementam uma interface remota e apenas os métodos especificados nas interfaces são fornecidos nas classes de stub. Se uma subclasse não implementa uma interface remota, mas uma superclasse implementa e um objeto da subclasse é passado para um método remoto, apenas os métodos da superclasse são acessíveis. Para entender isso melhor, considere o exemplo a seguir. Derivamos uma classe BookImpl de ProductImpl.

```
class BookImpl extends ProductImpl {
  private String ISBN;

  public BookImpl(Sting title, String theISBN, int sex, int
                  age1, int age2, String hobby) {
    super(title + " Book", sex, age1, age2, hobby);
    ISBN = theSBN;
  }

  public String getStockCode() { return ISBN; }
}
```

Agora, suponha que passemos um objeto book para um método remoto, ou com um parâmetro ou como um valor de retorno. O destinatário obtém um objeto stub. Mas esse stub não é um stub book. Em vez disso, é um stub da superclasse ProductImpl, pois apenas essa classe implementa uma interface remota. Assim, neste caso, o método getStockCode não está disponível remotamente.

Uma classe remota pode implementar várias interfaces. Por exemplo, a classe BookImpl pode implementar uma segunda interface, além de Product. Aqui, definimos uma interface remota StockUnit e fizemos a classe BookImpl implementa-la.

```
interface StockUnit extends Remote {
  public String getStockCode() throws RemoteException;
}

class BookImpl extends ProductImpl implements StockUnit {
  private ISBN;
  public BookImpl(Sting title, String theISBN, int sex, int
```

```
                    age1, int age2, String hobby) throws
                    RemoteException {
      super(title + " Book", sex, age1, age2, hobby);
      ISBN = theSBN;
    }

    public String getStockCode() throws RemoteException {
      return ISBN;
    }

  }
```

Agora, quando um objeto book for passado para um método remoto, o destinatário obterá um stub que tem acesso aos métodos remotos das classes Product e StockUnit. Na verdade, você pode usar o operador instanceof para descobrir se um objeto remoto em particular implementa uma interface. Aqui está uma situação típica onde você usará esse recurso. Suponha que você receba um objeto remoto através de uma variável de tipo Product.

```
Vector result = centralWarehouse.find(c);
for(int i = 0;i < result.size();i++) {
  Product p = (Product)result.elementAt(i);
  ...
}
```

Agora, o objeto remoto pode ou não ser um livro. Gostamos de usar instanceof para descobrir se ele é ou não. Mas não podemos testar:

```
if(p instanceof BookImpl) { // errado
  BookImpl b = (BookImpl)p;
  ...
}
```

O objeto p se refere a um objeto stub e BookImpl é a classe do objeto servidor. Poderíamos fazer a conversão de tipo do objeto para um BookImpl_Stub, mas não adiantaria muito.

```
if(p instanceof BookImpl_Stub) {
  BookImpl_Stub b = (BookImpl_Stub)p;
  ...
}
```

Os stubs são gerados mecanicamente pelo programa rmic, para uso interno do mecanismo RMI e os clientes não devem ter de pensar neles. Em vez disso, fazemos a

conversão de tipo para a segunda interface:

```
if(p instanceof StockUnit) {
  StockUnit s = (StockUnit)p;
  String c = s.getStockCode();
  ...
}
```

Esse código testa se o objeto stub ao qual p se refere implementa a interface StockUnit. Se assim for, ele chama o método remoto getStockCode dessa interface.

Resumindo:

- Se um objeto pertencente a uma classe que implementa uma interface remota é passado para um método remoto, o método remoto recebe o objeto stub.
- Você pode fazer a conversão de tipo desse objeto stub para qualquer uma das interfaces remotas que a classe de implementação implementa.
- Você pode chamar todos os métodos remotos definidos nessas interfaces, mas não pode chamar quaisquer métodos locais através do stub.

Usando Objetos Remotos em Conjunto

Os objetos inseridos em conjuntos devem sobrepor o método equals. No caso de um conjunto ou tabela hash, o método hashCode também deve ser definido. Entretanto, há um problema ao se tentar comparar objetos remotos. Para descobrir se dois objetos têm o mesmo conteúdo, a chamada a equals precisa entrar em contato com os servidores que contêm os objetos e comparar seus conteúdos. E essa chamada falharia. Mas o método equals na classe Object não é declarado de modo a lançar uma RemoteException, ao passo que todos os métodos em uma interfrace remota devem lançar essa exceção. Como um método de subclasse não pode lançar mais exceções do que o médoto de superclasse que substitui, você não pode definir um método equals em uma interface remota. O mesmo vale para hashCode.

Em vez disso, você deve contar com as redefinições dos métodos equals e hashCode na classe RemoteObject, que é a superclasse de todos os objetos stub e servidor. Esses métodos não examinam o conteúdo do objeto, apenas a localização dos objetos servidores. O método equals da classe RemoteObject considera dois stubs iguais caso eles se refiram ao mesmo objeto servidor. Dois stubs que se referem a objetos servidores diferentes numca são iguais, mesmo que esses objetos tenham conteúdo idêntico. Analogamente, o código de hash é calculado apenas a partir do identificador de objeto. Os stubs que se referem a objetos servidores diferentes provavelmente terão códigos de hash diferentes, mesmo que os objetos ervidores tenham conteúdo idêntico.

Essa limitação se refere apenas aos stubs. Você pode redefinir equals ou hashCode

para as classes de objeto servidor. Esses métodos são chamados quando você está comparando ou fazendo hash em stubs.

A classe RemoteObject é a base tanto da classe stub quanto da classe servidora. No lado stub, você não pode sobrepor os métodos equals e hashCode, pois os stubs são gerados mecanicamente. No lado servidor, você pode sobrepor os métodos das classes de implementação, mas eles são usados apenas localmente no servidor. Se você sobrepuser esses métodos, os objetos implementação e stub não serão mais considerados idênticos.

Resumindo: você pode usar objetos stub em tabelas de hash, mas deve se lembrar de que os testes de igualdade e o hash não levam em consideração o conteúdo dos objetos remotos.

Clonando Objetos Remotos

Os stubs não possuem um método clone; portanto, você não pode clonar um objeto remoto, chamando clone no stub. O motivo é novamente um tanto técnico. Se clone fizesse uma chamada remota para dizer ao servidor para que clone o objeto implementação, então o método clone precisaria lançar uma RemoteException. Mas o método clona da superclasse Object prometeu numca lançar uma exceção, exceto CloneNotSupportedException. Essa é a mesma limitação que você encontrou na seção anterior, quando viu equals e hashCode não pesquisam o valor do objeto remoto, mas apenas comparam referências de stub. Mas não faz sentido que clone crie outro clone de um stub – se você quisesse ter outra referência ao objeto remoto, poderia apenas copiar a variável stub. Portanto, clone simplesmente não é definido para stubs.

Se você quiser reproduzir um objeto remoto, então deve escrever outro método, digamos, remoteClone. Coloque-o na interface que define os serviços de objeto remoto. É claro que esse médoto pode lançar uma RemoteException. Na classe de implementação, basta definir remoteClone de modo a chamar clone e retornar o objeto de implementação clonado.

```
interface Product extends Remote {
  public Object remoteClone() throws RemoteException,
                  CloneNotSupportedException;
  ...
}

class ProductImpl extends UnicastRemoteObject implements Product
                  {
  public Object remoteClone() throws RemoteException,
                  CloneNotSupportedException {
    return clone();
```

```
    }
    ...
  }
```

Parâmetros Remotos Inadequados

Suponha que aprimoraremos nosso aplicativo de compras, fazendo-o apresentar uma figura de cada presente. Podemos incluir simplesmente o método remoto na interface Product?

```
void paint(Graphics g) throw RemoteException
```

Infelizmente, esse código não funciona, e é importante entender o motivo. O problema é que a classe Graphics não implementa interfaces remotas. Portanto, precisaria ser passada uma cópia de um objeto de tipo Graphics para o objeto remoto e você não pode fazer isso. Por que? Bem, Graphics é uma classe abstrata e os objetos Graphics são retornados através de uma chamada ao método getGraphics da classe Component. Essa chamada, por sua vez, pode acontecer apenas quando você tem alguma subclasse que implementa um contexto gráfico em uma plataforma específica. Esses objetos, por sua vez, precisam interagir com o código gráfico nativo e para fazer isso, eles devem armazenar ponteiros para os blocos de memória necessários pelos métodos gráficos nativos. A linguagem de programação Java, é claro, não tem ponteiros; portanto, essa informação é armazenada como inteiros no objeto gráfico e passa ela coerção de volta a ponteiros apenas nos métodos pares nativos. Agora, primeiramente, a máquina de destino pode ser uma plataforma diferente. Por exemplo, se o cliente executa Windows e o servidor executa X11(Ambiente gráfico no unix), então o servidor não tem os métodos nativos disponíveis para gerar imagens no Windows. Mas, mesmo que o servidor e o cliente tivessem o mesmo sistema gráfico, os valores de ponteiro não seriam válidos no servidor. Portanto, não faz sentido copiar um objeto gráfico. Por isso, a classe Graphics não é serializável e assim, não pode ser enviada através de RMI.

Em vez disso, se o servidor quiser enviar uma imagem para o cliente, ele terá de empregar outro mecanismo para transportar os dados pela rede. Conforme se verifica, esse transporte de dados é realmente difícil de fazer para imagens. A classe Image é tão dependente de dispositivo quanto a classe Graphics. Poderíamos enviar os dados da imagem como uma seqüência de byte no formato JPEG,mas não existe nenhum método no pacote AWT para transformar um bloco de dados JPEG em uma imagem. (Atualmente, isso pode ser feito apenas através do uso de classes não publicadas do pacote sum.awt.image.) Outra alternativa é enviar um array de inteiros representando os pixels. Na próxima seção, mostraremos como resolver esse problema de um modo mais simples: enviando uma URL para o cliente e usando um método da classe Applet que pode ler uma imafem a partir de uma URL.

Usando RMI com Applets

Existem várias preocupações especiais ao se executar o RMI com applets. Os Applets têm seu próprio gerenciador de segurança, pois eles são executados dentro de um navegador. Assim, não usamos o RMISecurityManager no lado do cliente.

Devemos tomar cuidado onde colocar os arquivos stub e de servidor. Considere um navegador que abre uma página Web com uma tag APPLET. O navegador carrega o arquivo de classe referenciado nessa tag e todos os outros arquivos de classe, conforme forem necessários, durante a execução. Os arquivos de classe são carregados a partir do mesmo host que contém a página da Web. Devido às restrições de segurança de applet, o applet pode fazer conexões de rede apenas com seu host de origem. Portanto, os objetos servidores devem residir no mesmo host que a página da Web. Isto é, o mesmo servidor deve armazenar:

- Páginas Web
- Código do applet
- Classes de stub.classObjetos servidores
- O registro RMI

Aqui está um exemplo de applet que estende ainda mais nosso programa de compras. Assim como o aplicativo anterior, o applet obtém informações do cliente e depois seleciona os produtos correspondentes. Entretanto, esse applet envia imagens dos itens recomendados. Conforme mencionamos anteriormente, não é fácil enviar imagens do servidor para o cliente, pois elas são armazenadas em um formato que depende do sistema gráfico local. Em vez disso, o servidor simplesmente envia ao cliente uma string com o nome do arquivo de imagem e podemos usar o método getImage da classe Applet para obter a imagem.

Aqui está como você deve distribuir o código para esse tipo de situação:

- java.rmi.registry.RegistryImpl – Em qualquer ligar no host; o registro deve estar em execução antes que o applet seja iniciado
- WarehouseServer – Em qualquer lugar no host; deve estar em execução antes que o applet seja iniciado
- WarehouseImpl – Pode estar em qualquer lugar no host, desde que WarehouseServer possa encontrá-lo
- WarehouseApplet – Diretório referenciado na tag APPLET
- Stubs – Devem estar no mesmo diretório que WarehouseApplet

O applet procura o registro RMI no mesmo host que o contém. Para encontrar seu host, ele usa os métodos getCodebase e getHost:

```
String url = getCodebase.getHost();
url = “rmi://” + url;
centralWarehouse = (Warehouse)Naming.lookup(url +
                   “/central_warehouse”);
```

Vide os códigos WarehouseApplet.java WarehouseServer.java. Note que o applet não instala um gerenciador de segurança