Java aplicada a Redes de Computadores

Prof Rogério Santos Pozza e Prof. Henrique Yoshikazu Shishido

Universidade Tecnológica Federal do Paraná
Campus Cornélio Procópio

1 Java RMI

A capacidade de trafegar informações em redes de computadores abre uma nova perspectiva no desenvolvimento de sistemas. Isso pode ser observado em sistemas de controle empresarial cada vez mais integrados e a disponibilização de partes de regras de negócio entre diferentes entidades. Para exemplificar, pode-se imaginar uma aplicação Java de uma "entidade A"que necessita acessar dados disponibilizados por uma aplicação Java de uma "entidade B". Por motivos de segurança, o ideal é não oferecer acesso direto à base de dados. Nesse contexto, a tecnologia Java oferece a API de Invocação Remota de Método denominada por Java RMI.

Java RMI permite desenvolver sistemas distribuídos em que métodos de objetos Java possam ser chamados por outras Java Virtual Machine (JVM), até mesmo em diferentes hosts conectados em rede, de maneira transparente ao desenvolvedor. Sendo assim, uma das principais vantagens da RMI é permitir o uso de objetos distribuídos com a mesma sintaxe e estrutura de um objeto local. Esta API faz uso da serialização para passar objetos completos como parâmetros a outras JVM.

1.1 Arquitetura RMI

A arquitetura básica desta API consiste em três elementos: um *servidor*, um *cliente* e um *servidor de registros*. O *servidor de registros* é responsável por depositar as referências dos objetos remotos no servidor, nos quais o *cliente* pode acessar os métodos implementados por meio de *stubs*. Um *stub* representa o comportamento de um objeto remoto e oculta os detalhes de comunicação para simplificar o mecanismo de chamadas remotas. Assim, a comunicação do *cliente* com o objeto remoto é realizada através de uma interface implementada pelo objeto remoto no *servidor*.

É importante ficar claro que, em primeira instância, o *cliente* consulta o *servidor de registros* para obter a referência do objeto remoto. Após obter a referência, o *cliente* se comunica diretamente com objeto remoto sendo capaz de invocar seus métodos conforme ilustrado na Figura 1.

Java RMI gerencia implicitamente a comunicação em rede e abstrai a passagem de parâmetros entre cliente e servidor. Quando um determinado objeto é

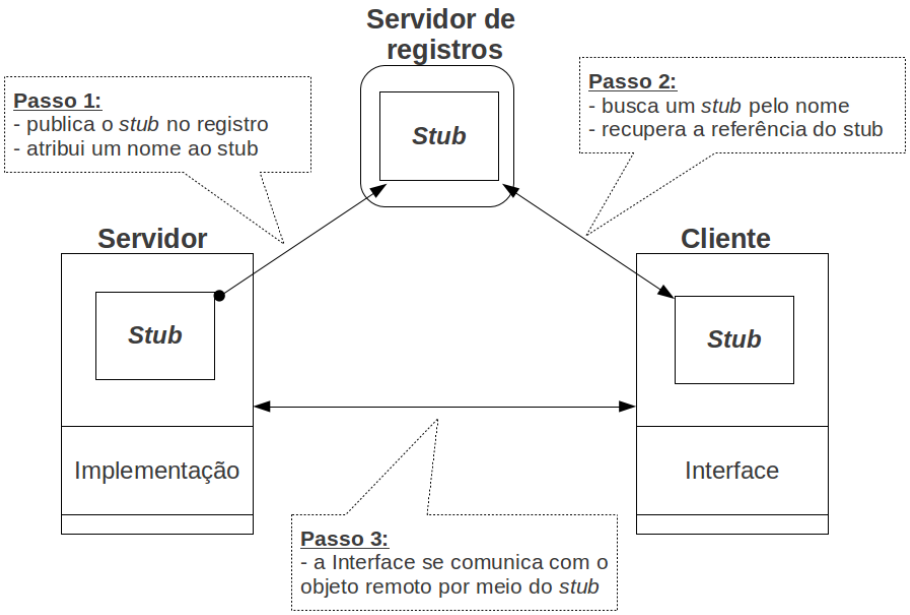

Figura 1: Arquitetura da Java RMI

invocado, seus argumentos são "empacotados"e enviados da JVM local para a remota, onde esses serão "desempacotados". Após o término da execução de um método, os resultados são retornados à JVM local. Se houver algum problema durante a chamada remota, uma exceção é repassada para a JVM.

O servidor RMI deve implementar a interface do objeto remoto a ser disponibilizado e codificar os métodos da *interface*. De fato, os objetos remotos podem possuir vários métodos, porém apenas os declarados na *interface* podem ser acessados remotamente. Para publicar um objeto no *servidor de registros* é necessário criar um *stub* do objeto remoto através da classe *java.rmi.server.UnicastRemoteObject* e registrá-lo por meio da classe *java.rmi.registry.Registry*.

Para invocar os métodos dos objetos remotos, o *cliente* precisa basicamente da classe *java.rmi.registry.Registry* para definir o endereço e porta do *servidor de registros* e buscar a referência do *stub* pelo nome definido no *servidor*. Dessa maneira, é possível invocar métodos de objetos remotos da mesma maneira como de um objeto local.

A API Java RMI é formada por pacotes capazes de tratar exceções, controlar a ativação do servidor, gerenciar o coletor de lixo e criar registros. A seguir são relacionados os pacotes para a invocação de métodos remotos.

- java.rmi
- java.rmi.dgc
- java.rmi.registry
- java.rmi.server

– java.rmi.activation

1.2 Utilizando o RMI

Esta seção apresentará um exemplo de implementação Java RMI baseado em um método remoto de validação de CPF. A Figura 2 ilustra as interações entre as instâncias. No *passo 1*, o *servidor de registros* é inicializado para disponibilizar os *stubs* oferecidos pelo *servidor*. No passo seguinte, é implementado e registrado no *servidor* um método de validação de CPF vinculado a um nome. No *passo 3*, o *cliente* busca pelo *stub* no *servidor de registros* através de seu nome e recebe a referência para, posteriormente, invocar o método remoto (validarCPF).

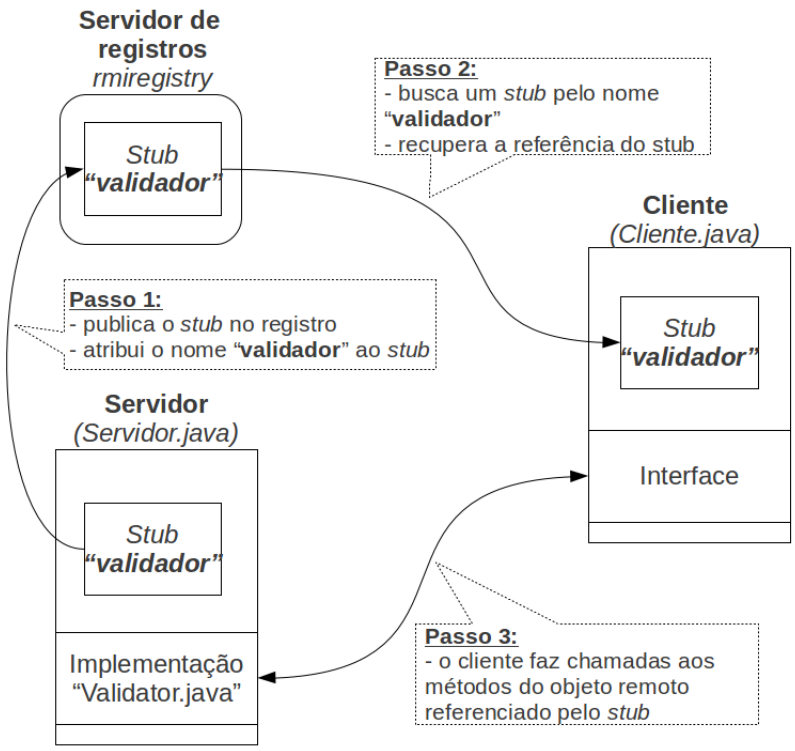

Figura 2: Diagrama de exemplo de uma aplicação RMI

Para facilitar o entendimento da implementação serão descritas as classes e interfaces para a codificação deste exemplo. Inicialmente é necessário codificar a interface (Validator.java) contendo os métodos que serão dispostos pelo *servidor*.

Interface RMI

Um objeto remoto é a instância de uma classe que implementa uma interface remota, e esta, por sua vez, estende a interface *java.rmi.Remote* e declara um conjunto de métodos remotos. Todos os objetos remotos devem declarar a cláusula *throws java.rmi.RemoteException* para qualquer exceção específica.

Segue um exemplo da interface (Interface.java) que declara apenas um método *validarCPF(String cpf)* capaz de verificar se uma sequência numérica é considerada válida:

Listagem 1.1: Interface Validator

```
1 import java.rmi.Remote;
2 import java.rmi.RemoteException;
3
4 public interface Validator extends Remote {
5
6    boolean validarCPF(String cpf) throws RemoteException;
7
8 }
```

A chance de falhas em invocações de métodos remotos é expressivamente superior se comparada às chamadas em objetos locais. Por exemplo, existem possibilidades como problemas no servidor e falha de comunicação de rede. Nesse contexto, é de fundamental importância o tratamento de exceções remotas utilizando *java.rmi.RemoteException* em cada um dos métodos da interface.

Servidor RMI

A classe Servidor (Servidor.java) possui um método *main* responsável por criar uma instância da implementação do objeto remoto, exportá-lo para o *servidor de registros* e vincular essa instância a um nome no *servidor de registros* que servirá como representação do recurso de rede.

O Código 1.2 define a classe Servidor (Servidor.java):

Listagem 1.2: Classe Servidor

```
1 import java.rmi.registry.Registry;
2 import java.rmi.registry.LocateRegistry;
3 import java.rmi.RemoteException;
4 import java.rmi.server.UnicastRemoteObject;
5
6 public class Servidor implements Validator {
7
8   public Servidor() { }
9
10  public boolean validarCPF(String cpf) {
11    /* Codigo para validar CPF */
```

```
12      return true;
13    }
14
15    public static void main(String args[]) {
16
17      try {
18
19        Servidor objeto = new Servidor();
20
21        Validator stub = (Validator) UnicastRemoteObject.
              exportObject(objeto, 0);
22        Registry registro = LocateRegistry.getRegistry("
              localhost", 1099);
23        registro.bind("validador", stub);
24
25        System.err.println("Servidor pronto!");
26
27      } catch (Exception e) {
28
29        System.err.println("Excecao: " + e.toString());
30        e.printStackTrace();
31
32      }
33    }
34  }
```

A codificação da classe Servidor implementa a interface remota *Validator* e oferece um método remoto *validarCPF*. O método *validarCPF* não declara a cláusula *throws*, pois isto já é implementado pela interface.

Criar e publicar um objeto remoto

Dentro do método *main* da classe Servidor é necessário criar o objeto que oferece o recurso. Posteriormente, esse objeto deve ser publicado no *servidor de registros* RMI para que possa receber chamadas remotas. De acordo com a classe Servidor (Servidor.java), isso pode ser feito com as linhas 22 e 24:

O método estático *UnicastRemoteObject.exportObject* exporta o objeto para receber chamadas de métodos remotos em uma porta TCP anônima e retorna um *stub* para o objeto remoto. O método *exportObject* inicia a escuta na porta TCP conforme o segundo parâmetro em um novo socket que passa a receber invocações remotas.

Observação: a partir do Java 5, os *stubs* não precisam ser compilados pelo comando *rmic*.

Registrar o objeto com Java RMI

Para invocar um método remoto, o *cliente* deve obter um *stub* do objeto remoto. Para isso, o Java RMI permite o cliente buscar por *stubs* através de um nome para obtê-los.

O *servidor de registros* no Java RMI é um serviço centralizador de nomes capaz de oferecer a referência (*stub*) de um objeto remoto. Nesse sentido, uma vez que um objeto remoto é registrado no servidor, *clientes* podem buscar um *stub* pelo nome, obter a sua referência e, finalmente, invocar métodos deste objeto.

O segmento de código da classe Servidor atribui o nome "validador"ao objeto *stub* (provido da classe Validator) no *registro*.

```
Registry registro = LocateRegistry.getRegistry("localhost", 1099);
registro.bind("validador", stub);
```

O método estático *LocateRegistry.getRegistry* contém argumentos e retorna um *stub* que implementa a interface remota *java.rmi.registry.Registry* e faz uma solicitação ao *servidor de registros* no servidor local, na porta TCP 1099 (padrão). Em seguida, o método *bind* é invocado pelo *stub registro* para ligar o *stub* do objeto remoto ao nome "validador"no registro.

Observação: a invocação do método LocateRegistry.getRegistry simplesmente retorna um *stub* apropriado para um registro. E, a chamada desta função não verifica se o servidor de registros está em execução. Portanto, se não houver nenhuma instância de servidor de registros sendo executada na porta TCP 1099 em *localhost* (máquina local), ocorrerá uma exceção do tipo *RemoteException* durante a chamada do método *bind*.

Cliente RMI

Nesta seção, a classe do *Cliente* (Cliente.java) apresenta detalhadamente como buscar um objeto remoto e ligar o seu nome a um objeto local.

Listagem 1.3: Classe Cliente

```
1  import java.rmi.registry.Registry;
2  import java.rmi.registry.LocateRegistry;
3
4  public class ClienteRMI {
5
6    private ClienteRMI() {
7    }
8
9    public static void main(String args[]) {
10
11     try {
12
13       Registry registro = LocateRegistry.getRegistry("
             localhost", 1099);
14       Validator stub = (Validator) registro.lookup("validador
             ");
```

```
15
16        boolean cpfValido = stub.validarCPF("04862529942");
17
18        System.out.println("O CPF 04862529942 e: " + cpfValido)
              ;
19
20      } catch (Exception e) {
21
22        System.err.println("Excecao: " + e.toString());
23        e.printStackTrace();
24
25      }
26    }
27 }
```

A classe Cliente inicialmente instancia um objeto *registro* para receber a referência do servidor pelo método estático *LocateRegistry.getRegistry* especificando o endereço e porta TCP do *servidor de registros*. Se não houver nenhum parâmetro especificado, então o registro é buscado no próprio endereço local e porta padrão 1099.

Posteriormente, o *cliente* instancia um *stub* que busca no *servidor de registros* pelo nome *"validador"* através do método *lookup*. É importante observar que o nome *"validador"* foi definido na classe Servidor.

Após esses comandos, o *cliente* é capaz de invocar o método *validarCPF* do objeto remoto, com o seguinte efeito durante a sua execução:

1. a execução *cliente* abre uma conexão com o servidor utilizando os parâmetros porta/endereço e então serializa os dados da chamada para o *stub* do objeto remoto;
2. o servidor aceita a conexão, encaminha a chamada ao objeto remoto e serializa o resultado da validação do CPF ao cliente;
3. por fim, o cliente recebe os dados e remove a serialização para retornar o resultado ao método invocado pelo *stub* local.

Compilando os código-fonte

Após implementados os arquivos Validator.java, Servidor.java e Cliente.java é preciso compilá-los com o comando:

```
javac Validator.java Server.java Cliente.java
```

ou

```
javac Validator.java
javac Server.java
javac Cliente.java
```

A partir do JDK 5.0 não é mais necessário o uso do comando *rmic*, pois há geração dinâmica de classes *stub* na execução utilizados pelo Java RMI para realizar a invocação remota de métodos. Assim, em versões anteriores ao JDK 5.0, ainda é preciso gerar as classes *stub* para a execução de *clientes*. Maiores detalhes podem ser consultados nas notas do Java RMI[1]

Inicializando o servidor de registros Java RMI

Para executar executar esse exemplo, é preciso executar três tarefas em sua devida sequência:

1. iniciar o servidor de registros RMI;
2. executar o Servidor;
3. e, executar o Cliente.

Para inicializar o *servidor de registros* RMI deve-se executar o comando *rmiregistry* na máquina onde será executado o servidor. Por padrão, o comando *rmiregistry* executado sem qualquer parâmetro coloca o serviço na escuta da porta TCP 1099, de acordo com o comando abaixo:

Em sistemas operacionais Linux/Solaris:

```
rmiregistry &
```

Ou na plataforma Windows:

```
start rmiregistry
```

Se por algum motivo, seja necessária a execução do serviço em outra porta de comunicação, pode acrescentar o comando seguido do número da porta. Por exemplo, suponha que queira inicializar o serviço na porta 2222, é necessário colocar a porta logo a frente do comando *rmiregistry* conforme a seguir:

Em sistemas Linux/Solaris:

```
rmiregistry 2222 &
```

Ou na plataforma Windows:

```
start rmiregistry 2222
```

Caso o registro esteja sendo executado em uma porta diferente de 1099, é necessário também especificar a porta nas chamadas da função *LocateRegistry.getRegistry* no servidor e cliente. Por exemplo, com um *servidor de registros* em execução na porta 2222, a chamada da função *getRegistry* no servidor deve ser:

```
Registry registro = LocateRegistry.getRegistry(2222);
```

[1] http://download.oracle.com/javase/1.5.0/docs/guide/rmi/relnotes.html

Inicializar o servidor

Com o servidor de registros em execução, para iniciar o Servidor é precisa executar a classe Servidor usando o comando *java* conforme a seguir:
Em sistemas operacionais Linux/Solaris:

```
java Servidor &
```

Ou em plataforma Windows:

```
start java Servidor
```

De acordo com o código exemplo da classe Servidor (Servidor.java), a saída deve ser semelhante a:

```
Servidor pronto!
```

Devem ser realizadas algumas observações:

- antes de executar o servidor é interessante conferir se a porta TCP do *servidor de registros* é a padrão ou não. Caso a porta do servidor não seja a padrão, deve-se colocar a porta como parâmetro do método *getRegistry* nas classes Servidor e Cliente;
- o diretório onde encontram-se os arquivos compilados deve constar no *CLASSPATH*.

Executando o cliente

Uma vez que o servidor esteja em execução e pronto, o cliente pode ser executado:

```
java Cliente
```

A saída do cliente será a seguinte mensagem:

```
O CPF 04862529942 é: false
```

Entretanto, dentro de todo este contexto, o *rmiregistry* não suporta registros de *stubs* de outra máquina que não seja o *localhost*.