PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Departamento de Ciência da Computação

Disciplina	Curso	Turno	Período
Computação Distribuída	Ciência da Computação	Manhã	8°

Professor
Felipe Cunha (felipe@pucminas.br)

Aula Prática 01

Java RMI

Java RMI (Remote Method Invocation) permite que objetos Java executando no mesmo computador ou em outros computadores comuniquem entre si por meio de chamadas de **métodos remotos**. Essas chamadas de métodos são semelhantes àquelas que ocorrem entre objetos de um mesmo programa. RMI está baseado em uma tecnologia anterior semelhante para programação procedural, chamada de chamada de procedimentos remotos (Remote Procedure Calls, ou RPC), desenvolvida nos anos 80, conforme visto em sala.

Sendo uma extensão de RPC, Java RMI permite comunicação distribuída de um objeto Java com outro. Uma vez que um método (ou serviço) de um objeto Java é registrado em um Servidor de Nomes como sendo remotamente acessível, um cliente pode pesquisar esse serviço e receber uma referência que permita utilizar o mesmo (isto é, chamar seus métodos). Como ocorre em RPC, o empacotamento dos dados é tratado pelo RMI. O programador não precisa se preocupar com a transmissão dos dados sobre a rede. RMI também não exige que o programador domine qualquer linguagem particular para definição de interfaces, porque todo o código de rede é gerado diretamente a partir das classes existentes no programa.

A arquitetura de Java RMI é dividida em três camadas:

- A camada de stub/skeleton oferece as interfaces que os objetos da aplicação usam para interagir entre si.
- A camada de referência remota é responsável por criar e gerenciar referências para objetos remotos;
- A camada de transporte implementa o protocolo que especifica o formato de solicitações enviadas aos objetos remotos pela rede.

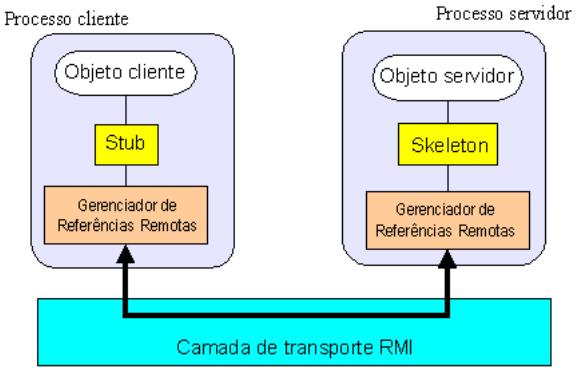

Figura 1: Arquitetura de camadas de RMI.

Interface Cliente

Cliente.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Cliente extends Remote {

    public void Exibir(String mensagem) throws RemoteException;

}
```

Implementação Cliente

ClienteImpl.java

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ClienteImpl extends UnicastRemoteObject implements Cliente {

    private static final long serialVersionUID = 1L;

    protected ClienteImpl() throws RemoteException {
        super();
    }

    public void Exibir(String m) throws RemoteException {
        try {
            System.out.println(m);

        } catch (Exception e) {
            System.out.println("Erro" + e);
        }
    }

    public static void main(String[] args) throws Exception {
        Servidor s = (Servidor) Naming.lookup("Servidor");

        Cliente c = new ClienteImpl();

        s.conectar(c);

        s.enviar("Alo Mundo");

        s.desconectar(c);
    }

}
```

Interface Servidor

Servidor.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Servidor extends Remote {

    public String conectar(Cliente c) throws RemoteException;

    public void enviar(String mensagem) throws RemoteException;

    public void desconectar(Cliente c) throws RemoteException;

}
```

Implementação Servidor

ServidorImpl.java

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class ServidorImpl extends UnicastRemoteObject implements Servidor {

    protected ServidorImpl() throws RemoteException {
        super();
        // TODO Auto-generated constructor stub
    }

    private static final long serialVersionUID = 1L;
    Vector<Cliente> conectados;

    public String conectar(Cliente c) throws RemoteException {
        conectados.addElement(c);
        return ("Usuario Conectado");
    }

    public void desconectar(Cliente c) throws RemoteException {
        conectados.remove(c);
    }

    public void enviar(String mensagem) throws RemoteException {
        int i = 0;
        do {
            Cliente c = (Cliente) conectados.elementAt(i);
            c.Exibir(mensagem);
        } while (i < conectados.size());
    }
```

```
    public static void main(String[] args) {

        try {
            Servidor s = new ServidorImpl();
            Naming.rebind("Servidor", s);
        } catch (Exception e) {
            System.out.println("Erro no servidor: " + e.getMessage());
        }

    }

}
```

Dicas:

- A execução da aplicação cliente-servidor em RMI requer, além da execução da aplicação cliente e da execução da aplicação servidora, a execução do Servidor de Nomes de RMI.
- O aplicativo **rmiregistry** faz parte da distribuição básica de Java e é encarregado de executar o Servidor de Nomes.
- A compilação do codigo fonte usa o compilador normal do Java sem nenhuma alteração de comando JAVAC NOMECLASSE.JAVA.
- Atenção para a configuração do CLASSPATH. Por isso, é ideal que você rode todos os arquivos no mesmo diretório.
- Se nenhum argumento for especificado para as chamadas BIND e LOOKUP, a porta 1099 é usada como padrão.