

# Java RMI

---

FELIPE CUNHA

# Java RMI

---

## RMI: Invocação Remota de Métodos

- Solução da Sun para desenvolvimento de aplicações distribuídas em Java

Permite que programas executando em uma JVM façam referências a objetos que estejam localizados em outra JVM.

- Chamadas remotas possuem a mesma sintaxe de chamadas locais

O objetivo principal é encapsular a comunicação Socket com os parâmetros de entrada de um método e a resposta com o resultado da execução do método.

# Comparação entre Objetos Remotos e Objetos Locais

	Objeto Local	Objeto Remoto
Definição	classe java (class)	Interface com métodos que serão acessados remotamente herda da interface Remote
Implementação	local interna à classe	classe remota que implementa a interface acima.
Construção	operador <b>new</b>	Servidor constrói com <b>new</b> , cliente referencia objeto pela rede. Pacote Java 2 Remote Object Activation permite utilização de <b>new</b> remoto.

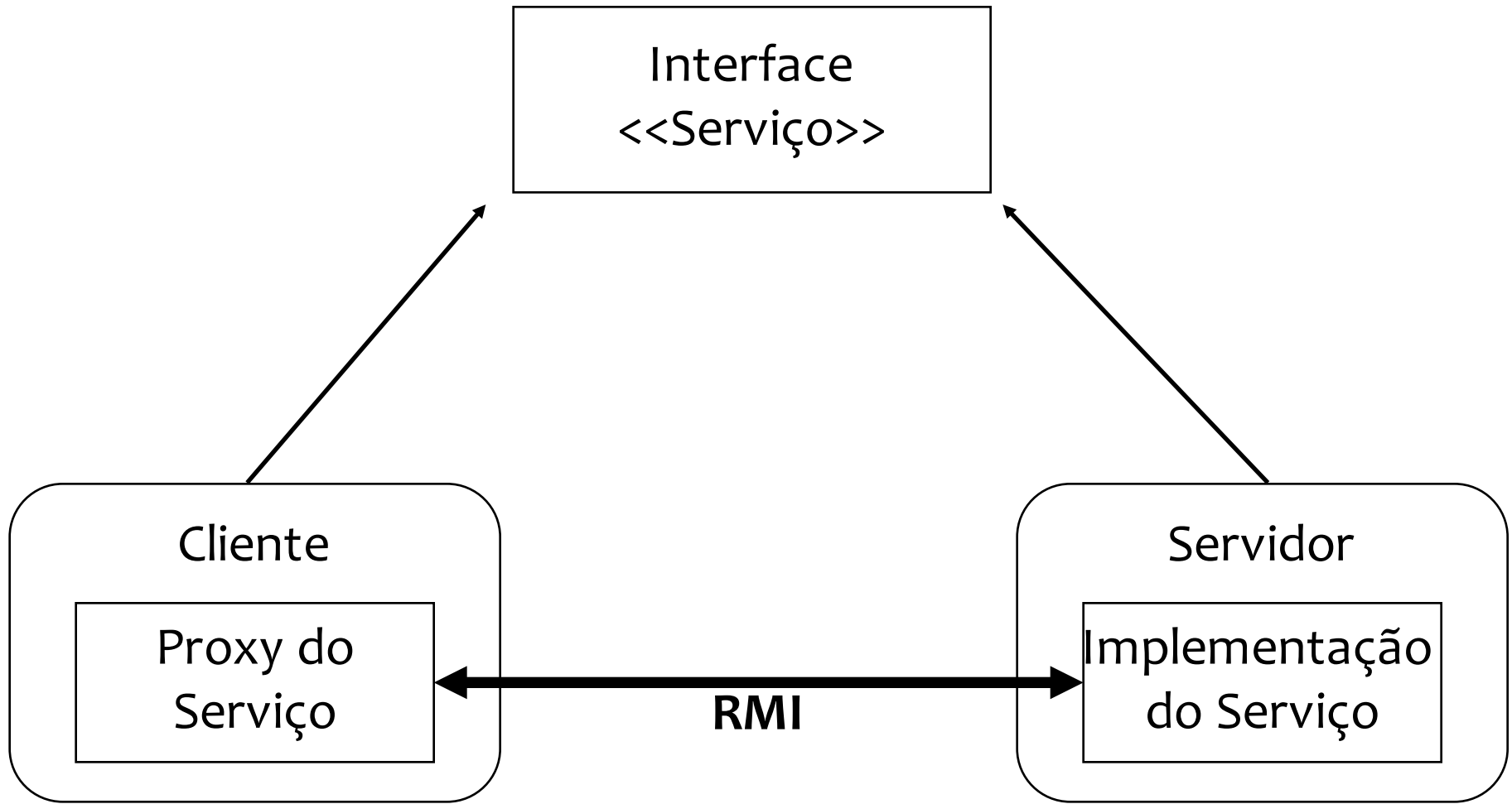
# Comparação entre Objetos Remotos e Objetos Locais

---

	Objeto Local	Objeto Remoto
Acesso	variável é uma referência local ao objeto	Objeto remoto acessado por uma referência Proxy (“stub”).
Finalização	Método finalize()	interface unreferenced
Exceções	classes que herdam de RuntimeException ou Exception	classes que herdam de RemoteException

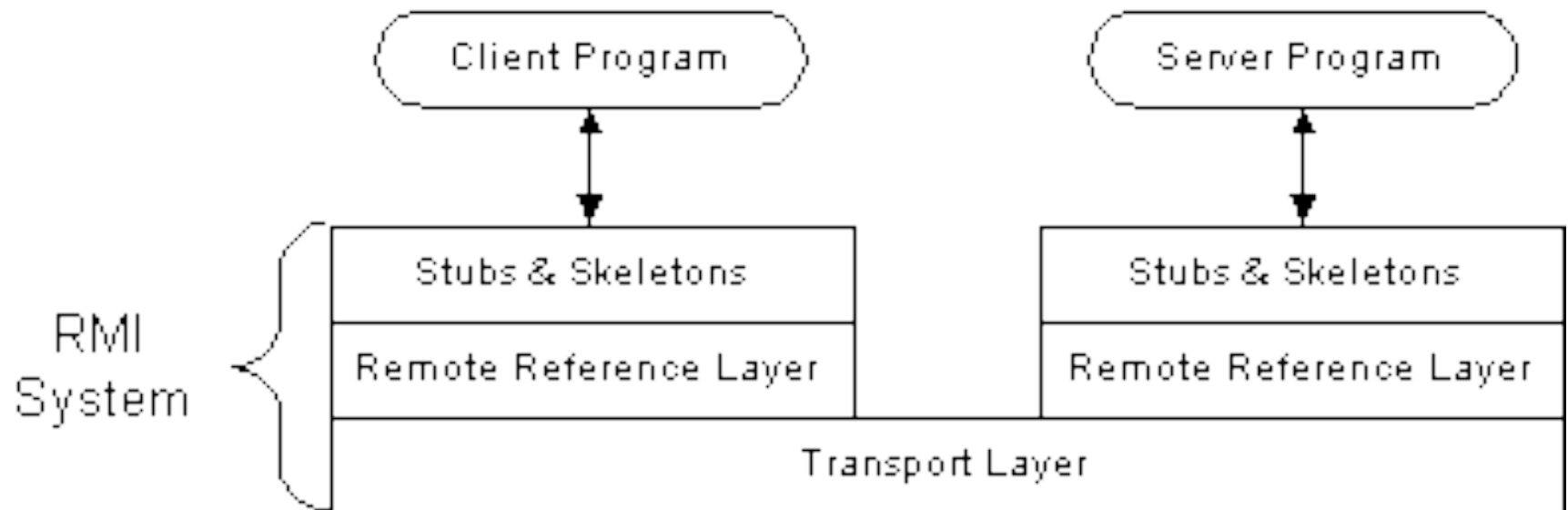
# Arquitetura RMI

---



# Arquitetura RMI

---



# Implementação de um sistema RMI

---

Definição da Interface para os serviços remotos

Implementação dos Serviços remotos

Geração dos arquivos Stub e Skeleton (proxy)

Um servidor irá hospedar o serviço remoto

Um servidor de Nomes RMI permite clientes encontrarem um servidor remoto.

Disponibilização das classes através de um servidor HTTP ou FTP

Um cliente que utilize os serviços remotos.

# Registry

---

Registry: objeto remoto que mapeia nomes em referências para objetos

- rmiregistry: String → referência para objeto

Principais funções:

- rebind: associa um nome a uma referência (substituindo qualquer associação já existente a este nome). Usado pelo servidor.
- lookup: retorna uma referência para um objeto remoto. Usado pelo cliente.

Todo servidor deve possuir seu registry



# Stub e Skeleton

---

Stub: parte que executa no cliente

Skeleton: parte que executa no servidor

rmic: compilador usado para gerar stub e skeleton a partir da interface remota.

Stub é instalado em um diretório especial do servidor e é buscado automaticamente pelos clientes, via HTTP ou sistema de arquivos

# Definição da Interface Remota

---

Interface Remota: relaciona os métodos que serão invocados remotamente

```
public interface Calculator extends java.rmi.Remote {  
  
    public long add(long a, long b) throws  
        java.rmi.RemoteException;  
    public long sub(long a, long b) throws  
        java.rmi.RemoteException;  
    public long mul(long a, long b) throws  
        java.rmi.RemoteException;  
    public long div(long a, long b) throws  
        java.rmi.RemoteException;  
}
```

# Implementação da Interface Remota

---

Uma interface remota é implementada por uma classe

```
public class CalculatorImpl extends
    java.rmi.server.UnicastRemoteObject
    implements Calculator {

    public CalculatorImpl() throws
        java.rmi.RemoteException {
        super();    }

    public long add(long a, long b) throws
        java.rmi.RemoteException {
        return a + b;    }

    ...

    public long div(long a, long b) throws
        java.rmi.RemoteException {
        return a / b;    }

}
```

# Gerando Stub e Skeleton

---

## Compilador rmic

- Uso: `rmic <opções> <nomes das classes>`
- Onde <opções> incluem:
  - `-keep` Manter código fonte intermediário
  - `-keepgenerated` (mesmo que `"-keep"`)
  - `-g` Informações de Debug
  - `-verbose` Mensagens do compilador
  - `-classpath <path>` Onde encontrar fontes eventualmente necessários
  - `-d <directory>` onde colocar arquivos compilados
  - etc.....

# Registrando Servidor

---

Para registrarmos o servidor no Registry usamos o pacote de serviço de nomes: `import java.rmi.Naming`

```
import java.rmi.Naming;

public class CalculatorServer {
    public CalculatorServer() {
        try {
            Calculator c = new CalculatorImpl();
            Naming.rebind("rmi://localhost:1099/CalculatorService", c);
        } catch (Exception e) {
            System.out.println("Trouble: " + e);
        }
    }

    public static void main(String args[]) {
        new CalculatorServer();
    }
}
```

# Implementação do Cliente

---

```
import java.rmi.Naming;
import java.rmi.RemoteException;
public class CalculatorClient {
    public static void main(String[] args) {
        try {
            Calculator c = (Calculator)
                Naming.lookup("rmi://remotehost/CalculatorService");
            System.out.println( c.sub(4, 3) );
            System.out.println( c.div(9, 3) );
        } catch (RemoteException re) {
            System.out.println(re);
        }
    }
}
```

# Executando aplicações RMI

---

Antes que uma aplicação RMI possa funcionar, devemos disparar o serviço de nomes numa máquina servidora conhecida:

```
start /min rmiregistry
```

Após a execução do Registry (que deve estar permanentemente rodando), poderemos então executar o servidor.

O `rmiregistry` é um *daemon* com endereço e porta bem conhecidos.

# Exemplo de Aplicação Distribuída Usando Java RMI

---

Codificar um servidor com um objeto da seguinte classe

```
public class ContaImpl .... { .....  
    public float obterSaldo ();  
    .....  
}
```

Codificar um cliente que chame remotamente o método obterSaldo() deste objeto



# Passo 1: Definição da Interface Remota

---

Interface Remota: relaciona os métodos que serão invocados remotamente

```
// arquivo Conta.java
```

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

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

```
    public interface Conta extends Remote { .....
```

```
        public float obterSaldo () throws
```

```
            RemoteException; .....
```

```
    }
```

# Passo 2: Implementação da Interface Remota

---

Uma interface remota é implementada por uma classe

```
// arquivo ContaImpl.java
import java.rmi.*
import java.rmi.server.*;
import conta.*;
public class ContaImpl extends UnicastRemoteObject
    implements Conta
{
```

# Passo 2: Implementação da Interface Remota (cont.)

---

```
private int numero;  
private float saldo= 0.0;  
public ContaImpl (int n) throws RemoteException  
{ numero= n; super(); }  
  
public float obterSaldo () throws  
RemoteException  
{ return saldo; }  
  
public static void main (String[] args) { .....  
    Conta c= new ContaImpl (804);  
    Naming.rebind ("Conta804", c);  
    ..... }
```

# Passo 3: Implementação do Cliente

---

```
// arquivo ClientImpl.java
import java.rmi.*
import java.rmi.server.*;
import conta.*;
public class ClientImpl {
    public static void main (String[] args) { .....
        Conta c= (Conta) Naming.lookup ( “//” + args[0] +
        “/Conta804”);
        float s= c.obterSaldo();    // chamada remota
        ..... }
}
```

# Passo 4: Compilação

---

## Compilação:

- `javac Conta.java` // Interface
- `javac ContaImpl.java` // Servidor
- `javac ClienteImpl.java` // Cliente

## Geração de stubs:

`rmic ContaImpl.java`

`rmic` gera os seguintes arquivos:

- `ContaImpl_Stub.class`: stub do cliente
- `ContaImpl_Skel.class`: skeleton (servidor)

# Passo 5: Execução

---

## Execução do Servidor

```
start /min rmiregistry // registry
```

```
start java contaImpl // servidor
```

## Execução do Cliente

```
java clienteImpl dcc_nt01
```

Observação: Alguns parâmetros de linha de comando foram omitidos nestes exemplos. Os comandos completos podem ser obtidos consultando o RMI Tutorial.

# Passagem de Parâmetros

---

Objetos podem ser passados como parâmetros, utilizando passagem por valor

- Basta que implementem a interface Serializable

Carregamento Dinâmico de Código:

- Se o código da classe de um objeto não estiver disponível na JVM para onde o objeto foi enviado, o mesmo é carregado dinamicamente.

# Exemplo de Passagem de Objeto como Parâmetro

---

## Interface Remota:

```
interface Compute extends Remote {  
    Object executeTask (Task t) throws  
        RemoteException; .....  
}
```

## Interface do Objeto Passado como Parâmetro:

```
interface Task extends Serializable {  
    Object execute ();  
}
```



# Exemplo de Passagem de Objeto como Parâmetro

---

Servidor:

```
class computeEngine
    extends UnicastRemoteObject
    implements Compute { ....
    public Object executeTask (Task t)
    { return t.execute(); }
    public static void main (String[] args)
    { // Cria e registra objeto remoto }
    ....
}
```

# Exemplo de Passagem de Objeto como Parâmetro

---

Cliente:

```
class Pi implements Task { .....  
    public Object execute()    // Calcula valor de  $\pi$   
    { ..... } .....  
    public static void main (String[] args) { .....  
        Compute comp= “referência p/ objeto remoto”  
        Pi Task= new Pi();  
        Float x= (Float) comp.executeTask (task);  
    }  
}
```

# RMI e Firewalls

---

RMI utiliza portas dinâmicas:

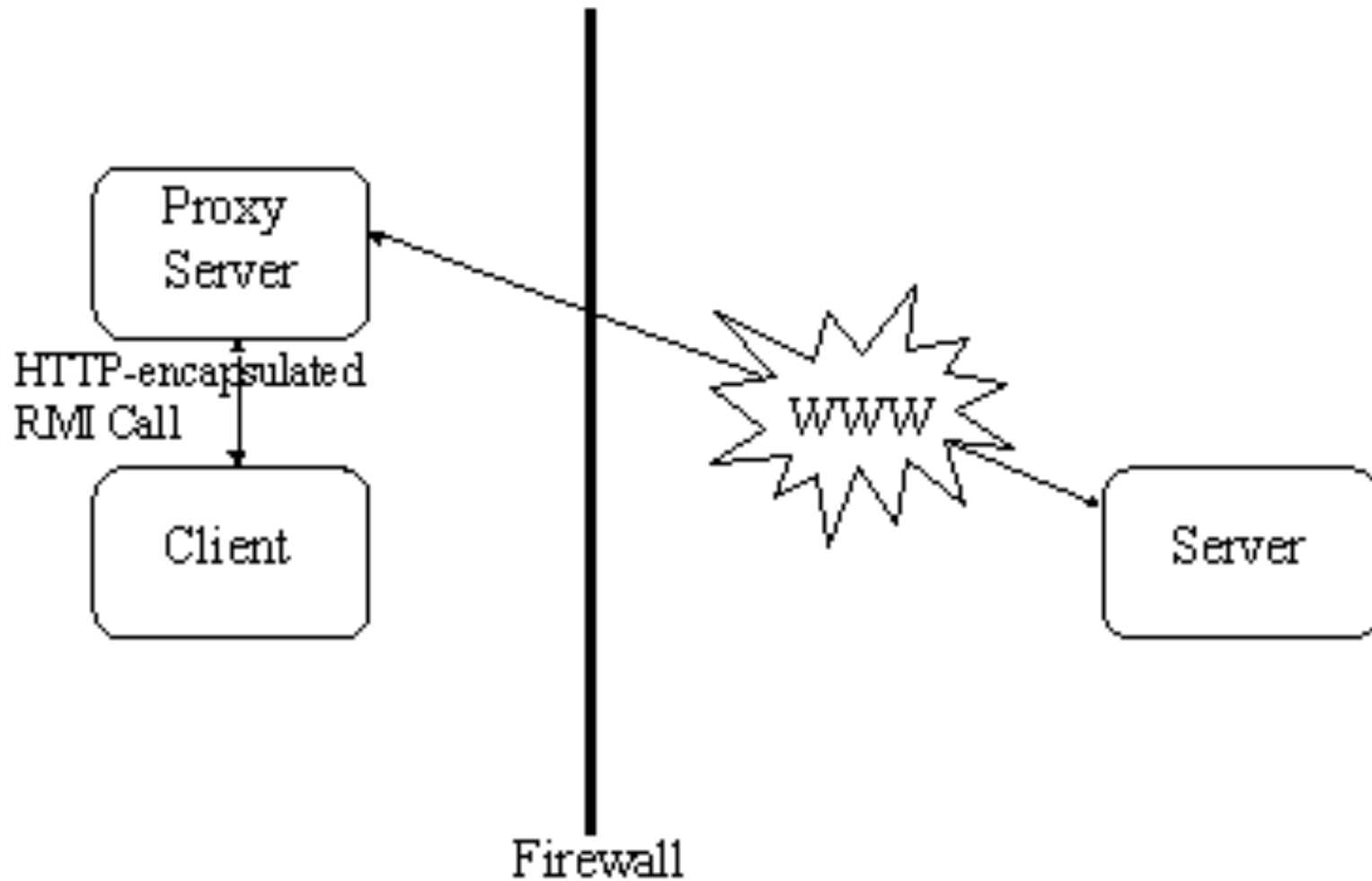
- Apesar do registry possuir porta bem conhecida, para cada conexão do servidor com o cliente, uma conexão socket com alocação dinâmica de porta é utilizada.

Firewalls são programados para bloquear tráfego de portas não conhecidas

- Solução: HTTP tunneling provido pelo pacote RMI
  - “http://hostname:port”
  - Deve-se setar devidamente `http.proxyHost`
  - “http://hostname:80/cgi-bin/  
java-rmi?forward=<port> ”
  - Script `cgi-bin` está incluído na distribuição do `java.rmi`

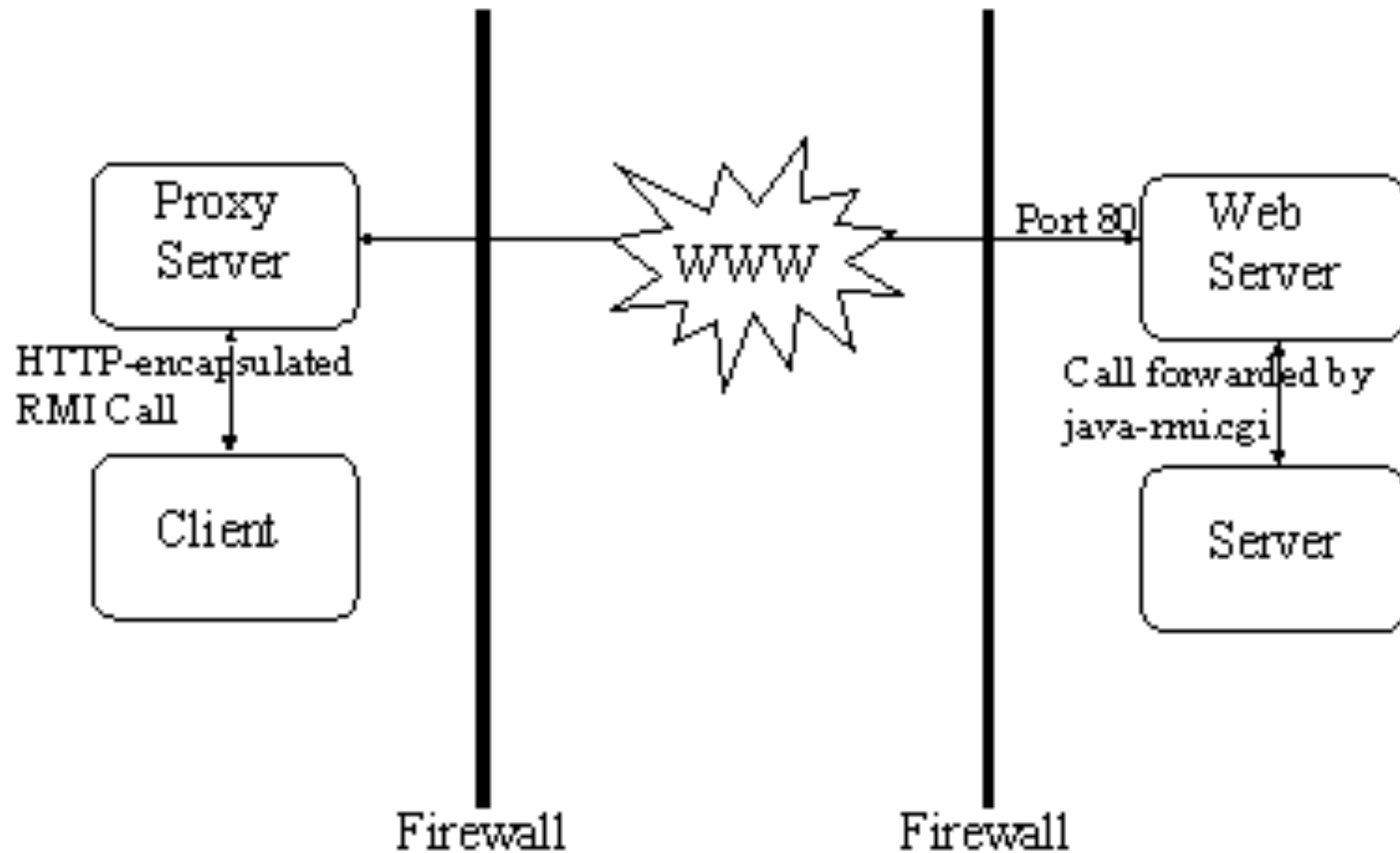
# RMI e Firewalls

---



# RMI e Firewalls

---



# RMI: Considerações Finais

---

Além de permitir tunelamento HTTP, Java RMI permite integração com CORBA através do RMI-IIOP, protocolo de comunicação do RMI com o IIOP (Internet Inter-ORB Protocol) disponível para comunicação de ORBs em sistemas CORBA via internet.

Para clientes que precisam ser também servidores, java.rmi possui Callback, onde usuário se exporta dinamicamente.

RMI é rápido e flexível quando se deseja criar conexão soquetes para tarefas específicas.

- Baseado em na teoria do RPC.

# Java RMI: Considerações Finais

---

## Vantagens:

- Portável entre plataformas para as quais existe uma implementação de Java
- Mobilidade de Objetos (e do código associado)

## Desvantagens:

- Solução proprietária da linguagem Java