# Invocação de Métodos Remotos

Java RMI

(Remote Method Invocation)

## Tópicos

- Tecnologia RMI
  - Introdução
  - Modelo de camadas do RMI
  - Arquitetura
  - Fluxo de operação do RMI
  - Passos para implementação
  - Estudo de caso

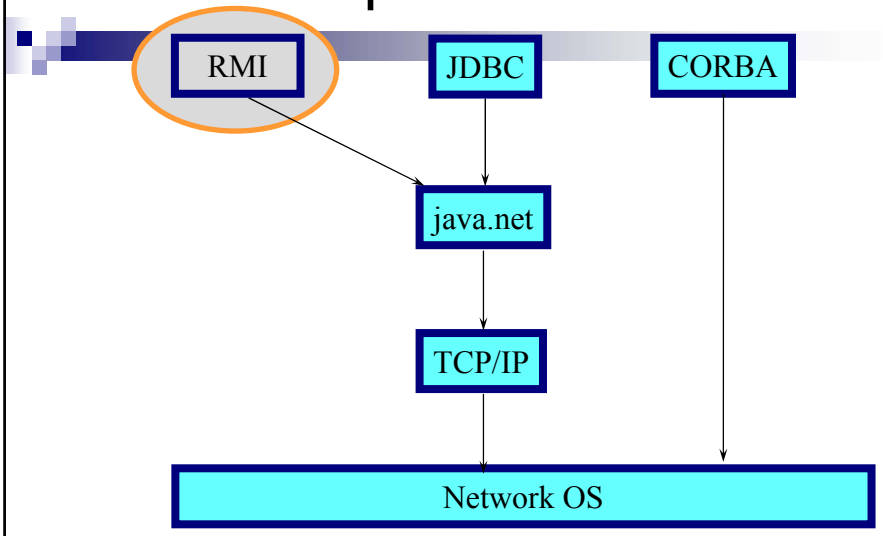
# Java RMI

- Arquitetura de acesso a objetos distribuídos suportada pela linguagem Java.
- Em termos de complexidade de programação e ambiente, é muito simples construir aplicações RMI, comparando-se com RPC e CORBA.
- Em termos de ambiente, exige somente suporte TCP/IP e um serviço de nomes de objetos (rmiregistry), disponibilizado gratuitamente com o JDK/SDK.

# Java RMI: Características

- Arquitetura de *middleware*.
  - Permite invocar objetos residentes em outras JVM.
  - Integra modelo de distribuição à plataforma Java.
  - Possibilita criação de aplicações distribuídas robustas
  - Mantém segurança dos ambientes de execução através do uso de gerenciadores de segurança.

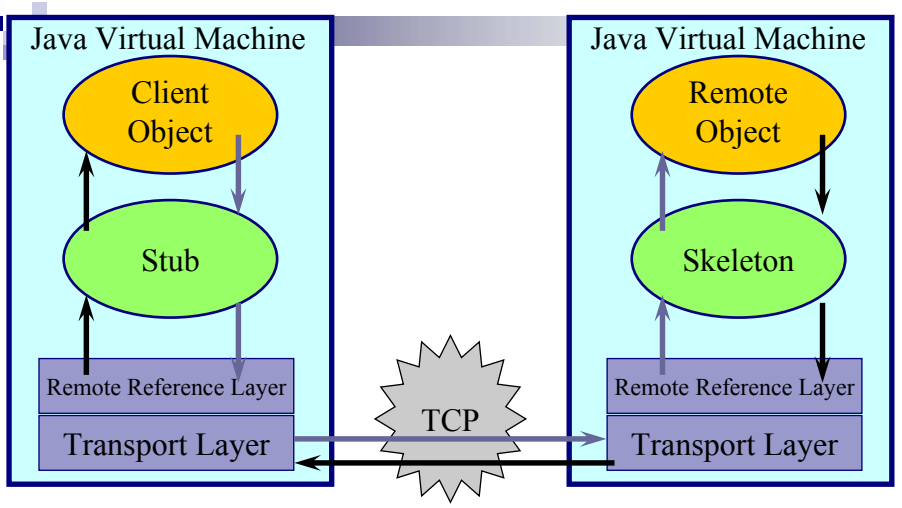
# RMI na arquitetura/API Java




# Arquitetura Java RMI

- Na realidade, o RMI é um mecanismo que permite a intercomunicação entre objetos Java localizados em diferentes hosts. Cada objeto remoto implementa uma interface remota que especifica quais de seus métodos podem ser invocados remotamente pelos clientes.
- Os clientes invocam tais métodos exatamente como invocam métodos locais.

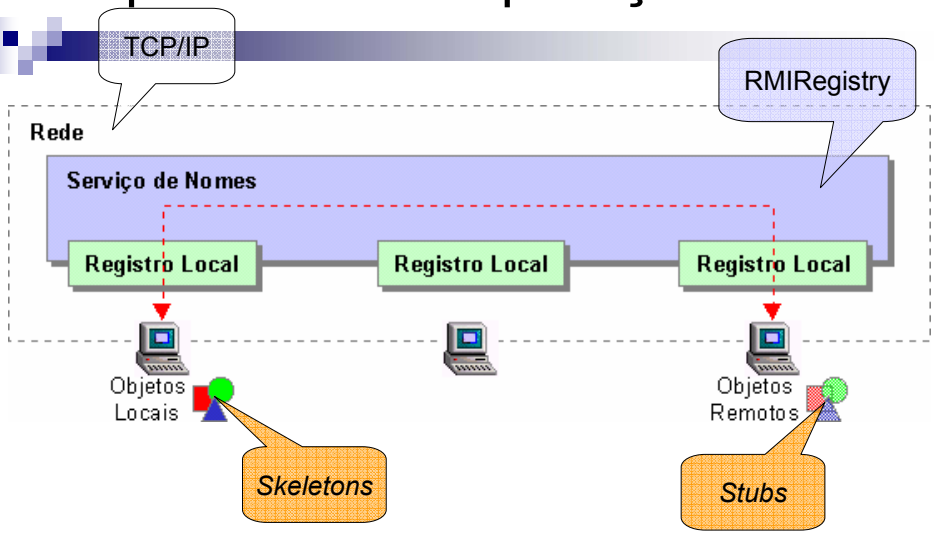
# Modelo de camadas do RMI




# Arquitetura das Aplicações

- Servidor:
  - Cria objetos destinados a uso remoto.
  - Efetua registro num serviço de nomes.
- Cliente:
  - Procura objetos remotos através do serviço de nomes.
  - Obtém referências para objetos remotos.
  - Utiliza objetos remotos.

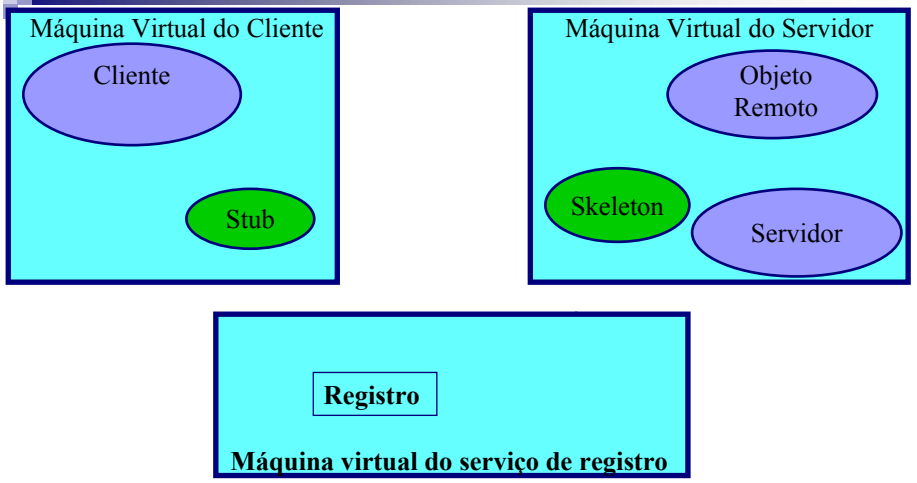
# Arquitetura das Aplicações




# Arquitetura das Aplicações

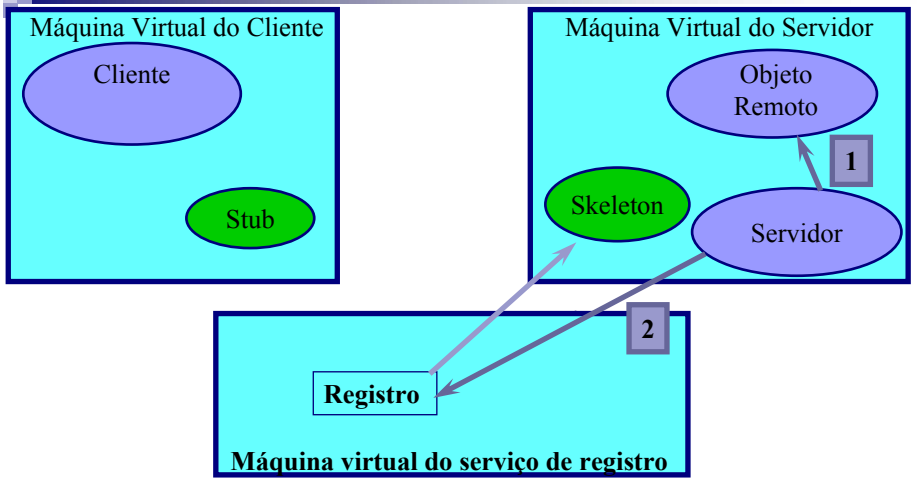
- Objetos remotos só podem ser manipulados através de interface específicas.
- Tais interfaces permitem a geração automática de:
  - *Stubs*: *proxies* locais para manipulação dos objetos.
  - *Skeletons*: *proxies* remotos para recepção das chamadas aos objetos.

# Arquitetura RMI



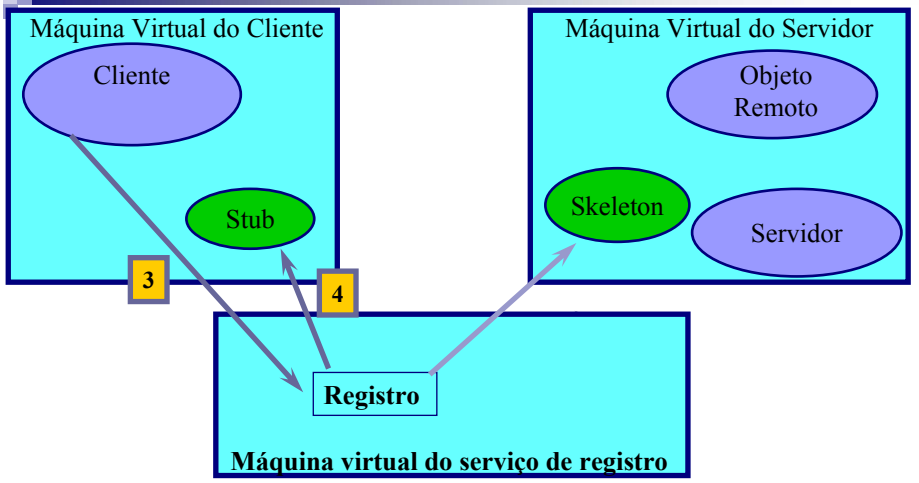

1. Servidor cria o objeto remoto
2. Servidor registra o objeto remoto

# Arquitetura RMI



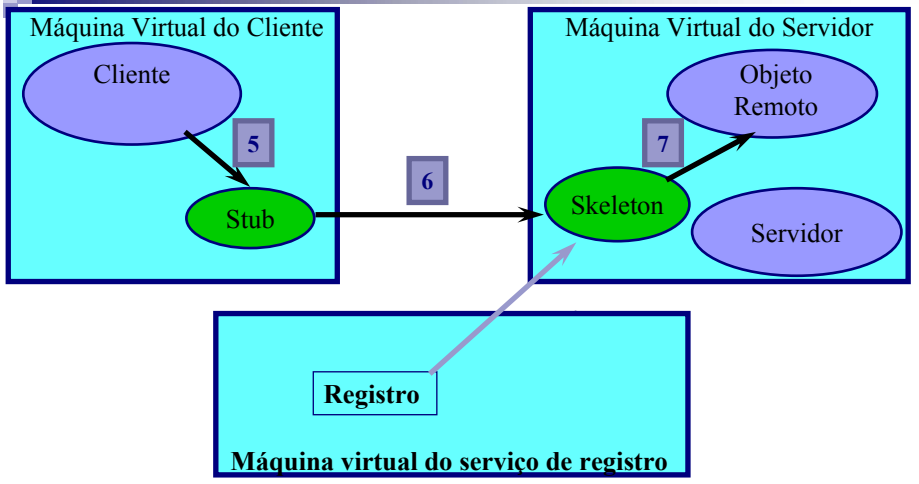
# Arquitetura RMI

**3. Cliente requisita o objeto ao serviço de registro**
**4. O registro retorna a referência remota e os stubs são criados**




# Arquitetura RMI

**5. Cliente invoca o método Stub**
**6. Stub interage com o skeleton**
**7. Skeleton invoca o método no objeto remoto**



# Interfaces para métodos remotos

- O primeiro passo para disponibilizar métodos que possam ser invocados remotamente consiste na preparação de uma interface remota com tais métodos.
- A construção desta interface pode ser feita com base na extensão da interface *Remote* do pacote java.rmi.
- A arquitetura RMI suporta duas classes implementando a mesma interface:
  - Uma, que implementa o serviço e é interpretada no servidor
  - Outra, que age como um mecanismo de proxy e é interpretada no cliente
- Um cliente faz chamadas de métodos ao objeto proxy, RMI envia a requisição à JVM remota, que executa o método.

# Convenções

- Interfaces iniciadas pelo prefixo **I**:
  **I**<*Nome*>
- Implementações finalizadas c/ sufixo **Impl**:
  <*Nome*>**Impl**
- Servidores terminados com sufixo **Server**:
  <*Nome*>**Server**
- Clientes terminados com sufixo **Client**:
  <*Nome*>**Client**

## Passos para a implementação de uma aplicação RMI

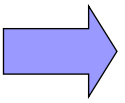
- Definir as funções da classe remota como interfaces escritas em Java.
- Codificar as classes de implementação (objeto remoto) e a classe do servidor.
- Codificar um programa cliente que usa os serviços remotos.

## Definição de Interfaces Remotas

- Devem ser públicas.
- Devem estender java.rmi.Remote.
  - É uma *tag* interface (não exige implementação de métodos especiais).
- Todos os métodos devem lançar a exceção java.rmi.RemoteException.

# Definição de Interfaces Remotas - Exemplo

- Suponha que se queira implementar um método remoto chamado **sayHello**(), que devolve uma String para chamada remota.



**Hello.java**

```
import java.rmi.*;
public interface Hello
        extends Remote {
        public String sayHello()
                throws RemoteException;
        }
```

# Implementação de Objeto Remoto

- Classes devem ser públicas.
- Devem estender java.rmi.server.UnicastRemoteObject.
- Devem implementar:
  - java.rmi.Remote diretamente ou
  - indiretamente via interfaces remotas.
- Todos os construtores devem lançar a exceção java.rmi.RemoteException.

# Implementação de Objeto Remoto

- Ou seja, cada classe que queira disponibilizar o tal método remoto **sayHello**() precisa implementar a interface especificada anteriormente. Além disto, a classe precisa estender a classe ***UnicastRemoteObject***, que é uma especialização de um servidor remoto (classe ***RemoteServer***).

# Implementação de Objeto Remoto - Exemplo

```
import  java.rmi.*;
import java.rmi.server.*;
import java.net.*;

public class  ServidorImpl
    extends UnicastRemoteObject
    implements  Hello {

    public ServidorImpl() throws RemoteException{ // Construtor
       // deve existir mesmo que não faça nada...
       super();
    }
    public String sayHello() throws RemoteException{  // Método remoto
       return("Hello World");
    }
```

# Criação e Registro de Objetos Remotos

- Objetos remotos devem ser instanciados.
- Cada instância deve ser registrada no serviço de nomes.
- Serviço de nomes se encarrega de criar *threads* para executar requisições remotas dirigidas a cada objeto registrado.

# Implementação do Servidor - Exemplo

```
public class Servidor {
  public static void main(String args[ ]) {
    try{
       Hello serv = new ServidorImpl();
       Naming.rebind("ServidorHello", serv);   // Registra nome do servidor
       System.out.println("Servidor remoto pronto.");
    }
       catch(RemoteException e) {
           System.out.println("Exceção remota: " + e);
    }
       catch(MalformedURLException e){ };
     }
} }
```

- A partir deste ponto, o objeto chamado ServidorHello está apto a aceitar chamadas remotas.

## Serviço de Nomes

- Classe java.rmi.Naming provê acesso ao serviço de nomes, i.e., ao RMIRegistry.
  - Naming.bind(String, Remote)
    registra objeto com nome dado.
  - Naming.rebind(String, Remote)
    substitui objeto registrado.
  - Naming.unbind(String, Remote)
    remove registro (nome e objeto).
  - Naming.lookup(String)
    obtém referência para objeto remoto.

## Cliente Remoto

- Obtém referência do objeto remoto através do serviço de nomes.
  - Necessário conhecer:
    - Nome ou IP do servidor de objetos
    - Nome do objeto remoto
- Utiliza o objeto ***exclusivamente*** através de sua interface.

# Implementação do Cliente

- O primeiro passo de implementação de um cliente que quer invocar remotamente método é obter o stub do servidor remoto. A localização deste stub é feita com o método lookup(endereço).
  - Este método devolve uma referência remota do objeto, através do envio do stub.

# Implementação do Cliente

```
import  java.rmi.*;

class Cliente {

public static void main(String args[ ]) {
  try {
     Hello serv= (Hello)
                          Naming.lookup("rmi://localhost/ServidorHello");
     String retorno=serv.sayHello();
     System.out.println(retorno);
  }
     catch(Exception e);
  }
}
```

# Operação do RMI

- Definir interfaces.
- Implementar objetos remotos.
- Implementar servidor de objetos.
- Implementar cliente dos objetos.

- Gerar *stubs*.

# Compilação e execução do servidor

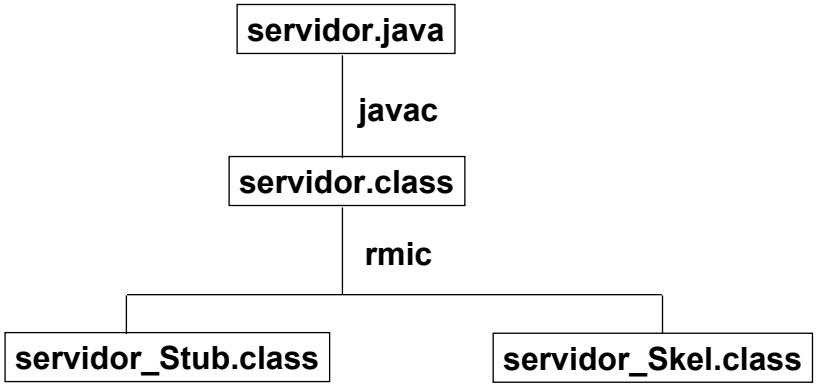
- Não basta apenas compilar e executar o programa anterior. Toda a compilação e execução necessita de um ambiente dado pela seguinte seqüência:

1. Compilar o arquivo .java
2. Chamar o aplicativo rmic para gerar o Stub e Skel
3. Ativar o controlador de registros (rmiregistry)
4. Chamar o interpretador com o servidor compilado

# Stubs e Skels

- O cliente, quando invoca remotamente um método, não conversa diretamente com o objeto remoto, mas com uma implementação da interface remota chamada stub, que é enviada ao cliente. O stub, por sua vez, passa a invocação para a camada de referência remota.
  - Esta invocação é passada para um skel (esqueleto), que se comunica com o programa servidor.

# Compilação do exemplo anterior



# Execução

- O primeiro passo antes de executar o servidor é ativar uma espécie de servidor de nomes de servidores que atendem solicitações de métodos remotos. Isto é feito chamando-se o programa **rmiregistry**.
  - rmiregistry [porta opcional] &
  - start rmiregistry [porta opcional]
- Uma vez que este programa está executando, pode-se chamar o interpretador java para o arquivo Servidor.class.
- Agora é só compilar (javac) e chamar o interpretador java para o arquivo Cliente.class

# Download Dinâmico

- RMI possibilita realizar o *download* dinâmico dos *stubs*, com isso:
  - Modificações nas implementações dos objetos remotos passam ser 100% transparentes nos clientes!
  - Apenas interfaces devem ser mantidas!

# Download Dinâmico

- Exige uso de gerenciadores de segurança.
- Exige definição de arquivo de políticas de segurança.
- Requer uso de um servidor HTTP ou FTP para fornecimento das classes remotas.

# Instalação de Gerenciador de Segurança

```
try {
   // cria e instala um gerenciador de segurança próprio
   RMISecurityManager security = new RMISecurityManager();
   System.out.println("[Instalando SecurityManager ...]");
   System.setSecurityManager(security);
} catch (Exception e) {
   System.err.println(e);
   // se o gerenciador de segurança desejado não pode ser
   // instalado, então o programa deveria ser encerrado
   System.exit(1);
}
```

# Arquivo de Políticas de Segurança

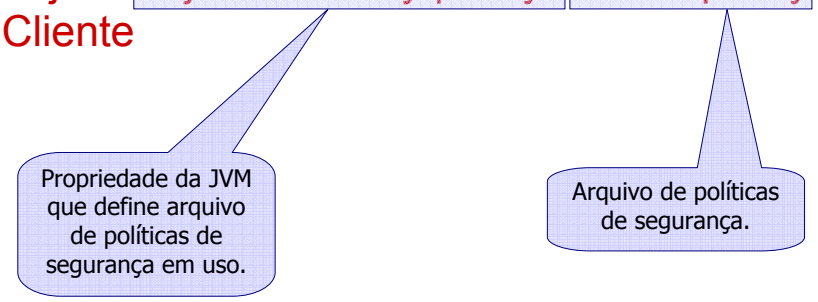
- Arquivo “hellow.policy”:

```
grant {
    permission java.net.SocketPermission
        "*:1024-65535", "connect,resolve,accpet,listen";
    permission java.net.SocketPermission
        "*:666", "connect,resolve,accept,listen";
};
```

# Aplicação do Arquivo de Políticas de Segurança

- Na execução do cliente:

```
>java -Djava.security.policy=hellow.policy Cliente
```



# Servidor de Classes

- Implementação de serviço simples:
  - Servidor de arquivos classes.
  - Pode usar protocolo HTTP.
  - Operação em porta e diretórios customizáveis:

  ><httpd> <porta>

# Operação do Servidor de Classes

- Acionar servidor de classes:

>httpd 8888 (por exemplo!)

- Acionar serviço de nomes:

>rmiregistry &

- Acionar servidor de objetos:

>java Servidor

- Acionar cliente:

>java –Djava.security.policy=hellow.policy
–Djava.rmi.server.codebase=http://localhost:8888 Cliente

# Esquema da chamada