Java RMI (parte I)

Luís Lopes
DCC-FCUP

Java RMI

- RMI (Remote Method Invocation) é uma API que permite a implementação de aplicações distribuídas baseadas em objectos
- o objectivo principal dos arquitectos da API era o de extender a semântica da linguagem para suportar aplicações cujas componentes executam em diferentes JVMs, mantendo uma sintaxe semelhante à dos programas não distribuídos (single JVM)

Java RMI

- a JVM original não permitia o movimento de código e objectos para fora do seu espaço de endereçamento
- o RMI teve de estender este modelo por forma a descrever qual o comportamento das classes e objectos em ambientes com múltiplas JVMs
- em particular o RMI permite a invocação de métodos em objectos de uma JVM remota

Java RMI

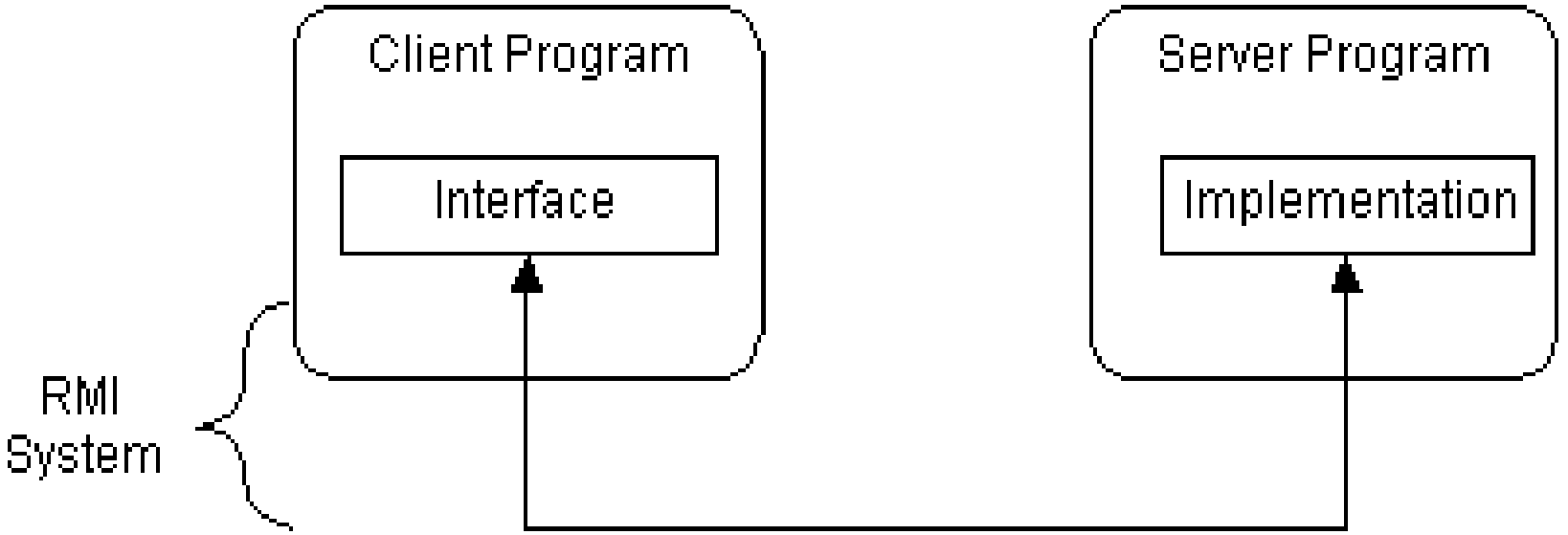
- permite a troca de classes e objectos com complexidade arbitrária entre clientes e servidores, por exemplo, como argumentos ou valores de retorno de chamadas a métodos
- um cliente ou servidor pode mesmo carregar objectos de tipos diferentes dinamicamente (dynamic loading)

Java RMI

- os conceitos fundamentais na arquitectura RMI são os de comportamento e de implementação
- interface é o código que define o comportamento de um componente de software
- uma implementação é o código que implementa o comportamento descrito pela interface
- e.g. o cliente só precisa de conhecer a interface fornecida por um servidor, o qual fornece uma implementação

Java RMI

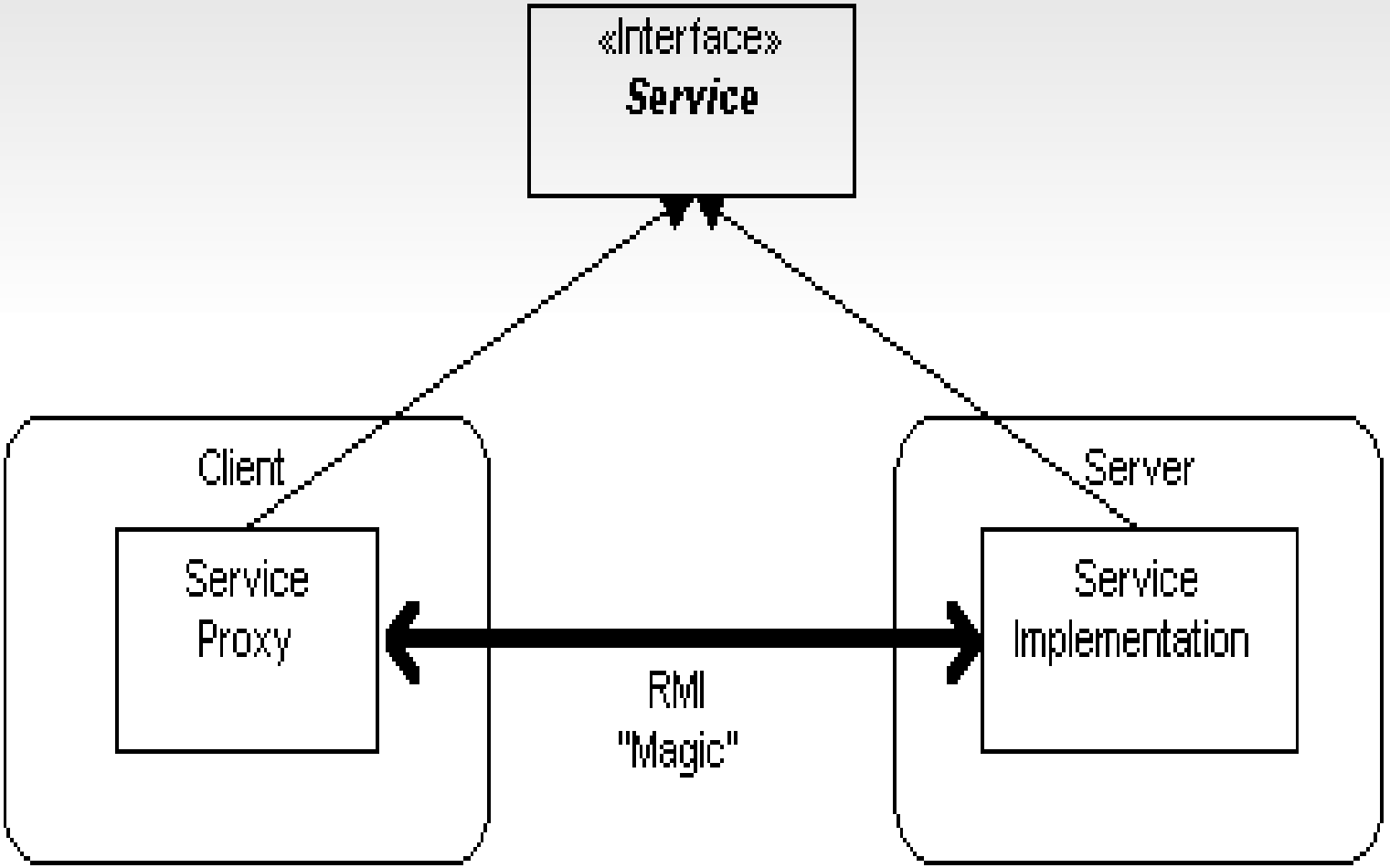
- No RMI a definição de um serviço é feita utilizando uma interface Java que extende java.rmi.Remote
- A implementação do serviço é feita numa classe Java



Java RMI

- uma interface Java não contém nenhum código
- numa aplicação RMI existirão pelo menos duas classes que implementam uma interface: a classe do servidor e uma classe que funciona como um proxy para o servidor e executa no lado do cliente
- a invocação de métodos por um cliente no servidor é na realidade uma invocação local feita num proxy do servidor
- os valores de retorno são passados da mesma forma

Java RMI


«Interface»
Service
Client
Server
Service
Proxy
Service
Implementation
RMI
"Magic"

Java RMI

- o RMI é construído com base em 3 camadas de software
  - Stub e Skeletons
  - Remote Reference
  - Transport
- a camada Stub e Skeletons intercepta as chamadas a métodos remotos efectuadas pelo cliente e re-encaminha-as via o proxy do servidor localizado na JVM do cliente

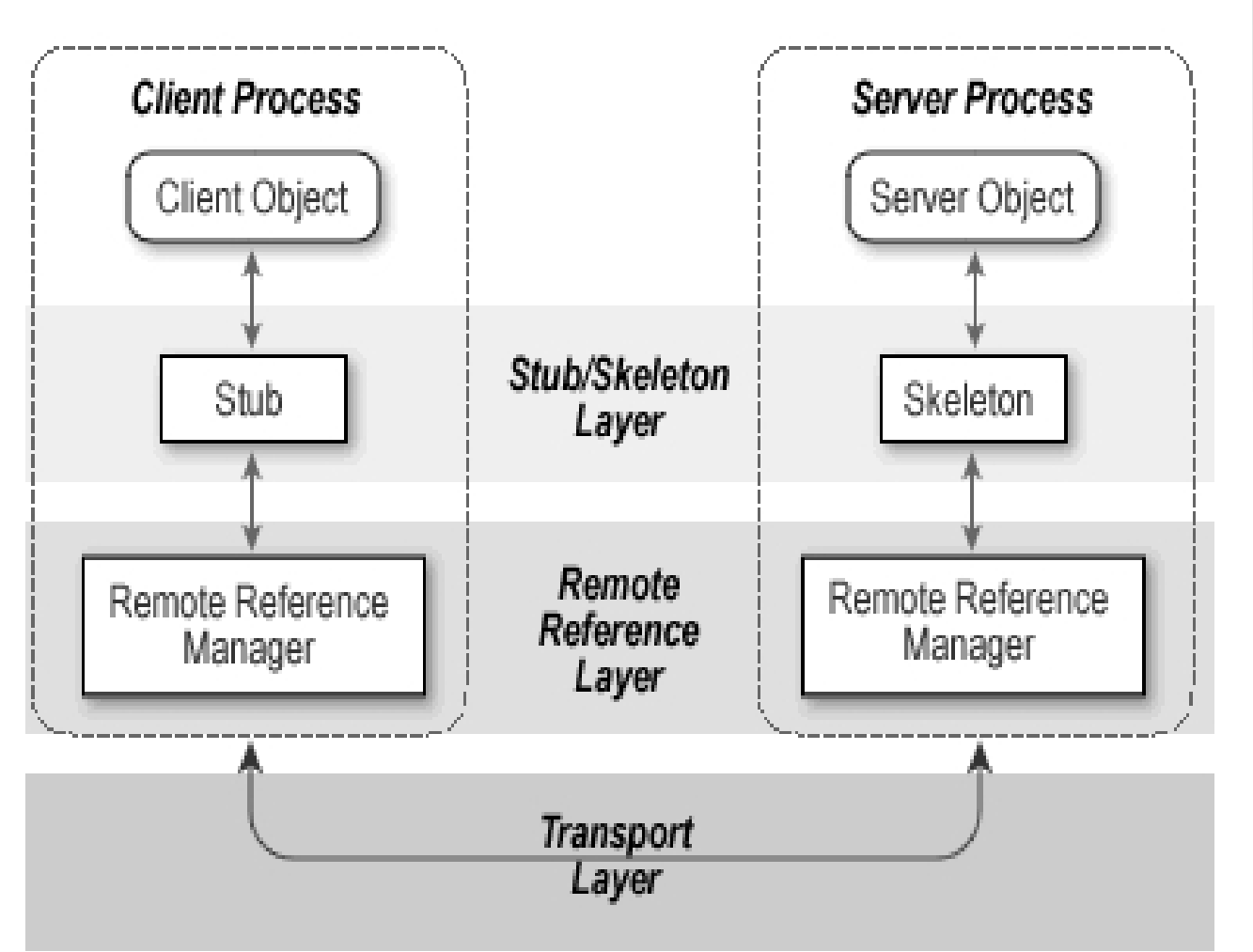
Java RMI

- a camada Remote Reference interpreta e gere as referências feitas pelo cliente a objectos remotos que fornecem um serviço
- esta camada é responsável pela conexão entre clientes e objectos servidores registados na rede
- esta camada também providencia o Remote Object Activation (activação de objectos servidores por instanciação de uma classe)

Java RMI

- a camada Transport é baseada no protocolo TCP/IP e providencia uma conexão estável entre clientes e servidores
- também implementa algum suporte para a penetração de firewalls (e.g. HTTP tunneling)

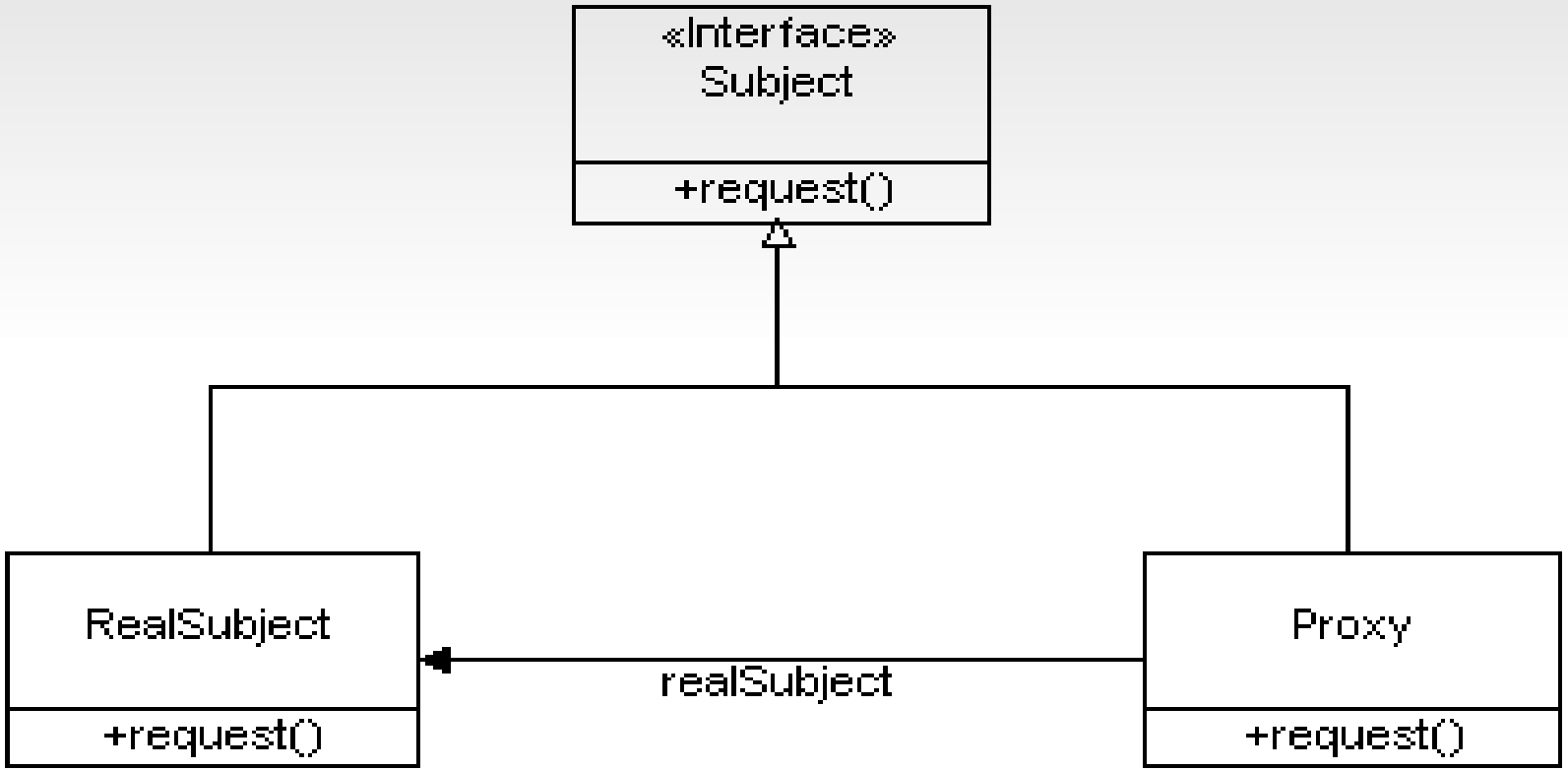
Java RMI


Client Process
Client Object
Stub
Remote Reference Manager
Stub/Skeleton Layer
Remote Reference Layer
Server Process
Server Object
Skeleton
Remote Reference Manager
Transport Layer

Java RMI

- a camada Stub & Skeleton é responsável pela implementação do padrão proxy
- neste padrão, um objecto num contexto (e.g. um servidor uma JVM remota) é representado por um outro (o proxy), num outro contexto (e.g. na JVM do cliente)
- o proxy (implementado no stub) sabe como re-encaminhar chamadas a métodos para o seu objecto original

Java RMI


«Interface»
Subject
+request()
RealSubject
+request()
Proxy
+request()
realSubject

Java RMI

- no RMI, uma classe implementada no stub desempenha o papel de proxy ao passo que a classe que implementa o servidor remoto desempenha o papel de RealSubject
- o Skeleton é uma classe auxiliar que interage com o Stub do lado do servidor
- lê os parâmetros da chamada ao método, faz a chamada ao método, recebe o valor de retorno e envia-o para o Stub

Java RMI

- a partir do Java 2, as classes Skeleton passaram a ser implementadas como parte do protocolo RMI, utilizando Java Reflection

Java RMI

- a camada Remote Reference define e suporta a invocação de métodos numa conexão RMI
- a camada utiliza um tipo de objecto designado por RemoteRef e que representa o link entre o Stub e o objecto servidor
- até ao JDK1.1 o servidor tinha de estar registado no sistema RMI para um cliente poder nele invocar métodos (unicast, point-to-point connection)

Java RMI

- a partir do Java 2, o RMI permite outra semântica para a invocação de métodos
- quando uma chamada a um método é feita num proxy de um objecto activatable, o RMI determina se o objecto servidor está inactivo
- nesse caso o RMI pode instanciar o objecto servidor restaurando o seu estado a partir de um checkpoint prévio, como resposta à invocação do método

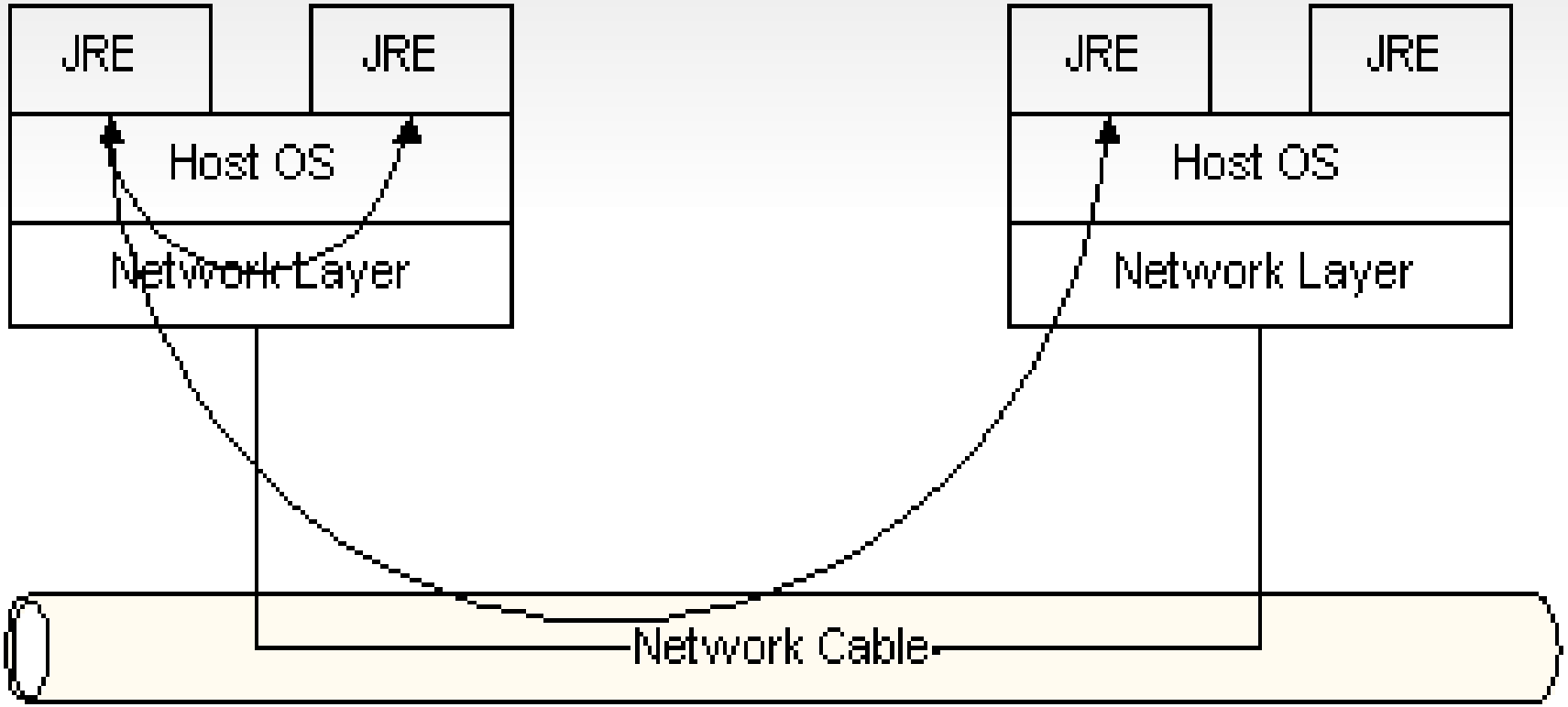
Java RMI

- outras semânticas de chamada a métodos são possíveis, por exemplo, o multicast, em que um cliente, via um único proxy, envia uma invocação simultaneamente a vários servidores na rede (não implementado)
- a arquitectura RMI é extensível e ortogonal a estes aspectos

Java RMI

- a camada Transport faz a conexão entre as JVMs
- as conexões são baseadas em streams utilizando TCP/IP
- mesmo entre JVMs no mesmo computador a comunicação é feita via TCP/IP

Java RMI


JRE
JRE
Host OS
Network Layer
JRE
JRE
Host OS
Network Layer
Network Cable

Java RMI

- o RMI tem um esquema de descoberta de serviços por nome que usa um serviço...
- este serviço mantém um registo de todos os serviços e as suas localizações no sistema RMI
- o serviço de registo está acessível num host e num número de porta bem conhecidos
- o RMI pode utilizar vários destes serviços como o Java Naming & Directory Interface ou o RMI Registry (corre em todos os hosts com objectos servidores)

Java RMI

- para fornecer um serviço remoto um servidor cria um objecto que implementa a interface do serviço e exporta-o para o sistema RMI
- o RMI cria um serviço que escuta por conexões e invocações a métodos do objecto
- o servidor regista o objecto no RMI Registry com um nome público

Java RMI

- no lado do cliente, o RMI Registry é consultado para descobrir a referência ao objecto por nome do serviço
- o nome do serviço é dado por um URL da forma rmi://hostname[:<name_service_port>:]/<service_name>
- o método retorna uma referência para o objecto remoto

Java RMI

- passagem de parâmetros e retorno de resultados na JVM
- tipos primitivos: copy by value
- objectos: são identificados por referências no heap
- quando atribuimos um a uma variável estamos a guardar a referência
- quando passamos a variável como argumento, a referência é copiada e não o objecto

Java RMI

- passagem de argumentos no RMI
- estamos a falar de passar argumentos e retorno de resultados para/de JVMs remotas
- tipos primitivos: copy by value
- objectos: copy by value, o objecto e não a sua referência é copiado
- o envio do objecto é feito depois de serializado, e é deserializado à chegada

Java RMI

- a serialização transforma um grafo de objectos numa stream de bytes que pode ser enviada entre JVMs
- o RMI introduz outro tipo de parâmetro, para além dos anteriores: objectos remotos
- tratam-se de objectos que implementam interfaces remotas (serviços RMI)
- um cliente pode obter uma referência para um destes objectos via RMI Registry ou como resultado de uma chamada a um método

Java RMI

- quando um cliente invoca um método que retorna uma referência para um objecto remoto, o RMI não retorna o objecto mas antes uma referência para um proxy do objecto remoto
- objectos remotos não são passados por valor
- a JVM do cliente recebe uma referência para o proxy do objecto remoto

Java RMI (exemplo)

Java RMI

- componentes de uma aplicação RMI
  - interfaces para o serviço remoto
  - implementação para o serviço remoto (servidor)
  - stubs e skeletons
  - programa de arranque do servidor
  - um serviço de nomes (e.g. RMI Registry)
  - um servidor para classes (FTP ou HTTP, opcional)
  - um programa cliente

Java RMI

- Passos na implementação
  - desenhar as interfaces (javac)
  - implementar as interfaces (javac)
  - gerar stubs para essa implementação (rmic)
  - escrever código para o servidor RMI (javac)
  - escrever código para o cliente RMI (javac)
  - correr o RMI Registry (rmiregistry &), o servidor e o cliente

Java RMI

- desenhar as interfaces (Calculator.java)
- implementar as interfaces (CalculatorImplementation.java)
- gerar stubs para essa implementação (rmic CalculatorImplementation)
- escrever código para o servidor RMI (CalculatorServer.java)
- escrever código para o cliente RMI (CalculatorClient.java)
- correr o RMI Registry (rmiregistry &), o servidor (java CalculatorServer) e o cliente (java CalculatorClient)

Java RMI

- a implementação do servidor no CalculatorImplementation.java extende java.rmi.server.UnicastRemoteObject
- isto torna mais simples o código de registo do serviço e procura do mesmo pelo servidor e cliente, respectivamente
- no entanto, em certos casos, o servidor poderá querer extender outra classe Java pelo a solução acima poderá não ser adequada (o Java só permite herança simples)

Java RMI

- nestes casos, podemos implementar o servidor sem fazer essa extensão
- o código para o registo do serviço fica um pouco mais extenso e é necessário fazer a exportação do serviço explicitamente com UnicastRemoteObject.exportObject()

Java RMI

(demo)

Java RMI
(mobilidade de código)

Java RMI

- Exploramos agora uma das características mais poderosas do Java RMI: a capacidade de carregar classes dinamicamente a partir da Web para uma aplicação local
- Corresponde à extensão para um ambiente distribuído do modelo de execução e carregamento de classes da JVM
- A JVM carrega classes dinamicamente a partir de servidores Web ou FTP especificados inicialmente

Java RMI

- Exemplo: Compute Server
- Neste exemplo, um servidor recebe um objecto executável
- Só sabe deste que tem o método **void execute()**
- Os clientes enviam diferentes objectos que implementam esta interface para execução no servidor
- O servidor não conhece à priori o código das classes para os ditos objectos → tem de carregá-lo dinamicamente

Java RMI

- num primeiro passo criamos as interfaces que definem o protocolo entre um clientes e o servidor em compute/
- criamos um ficheiro .jar com as interfaces definidas para disponibilizar aos clientes e aos servidores
- desta forma podemos desenvolver clientes e servidores para o serviço com base no contracto definido pela interface
- javac compute/*.java
- jar cvf compute.jar compute/*.class

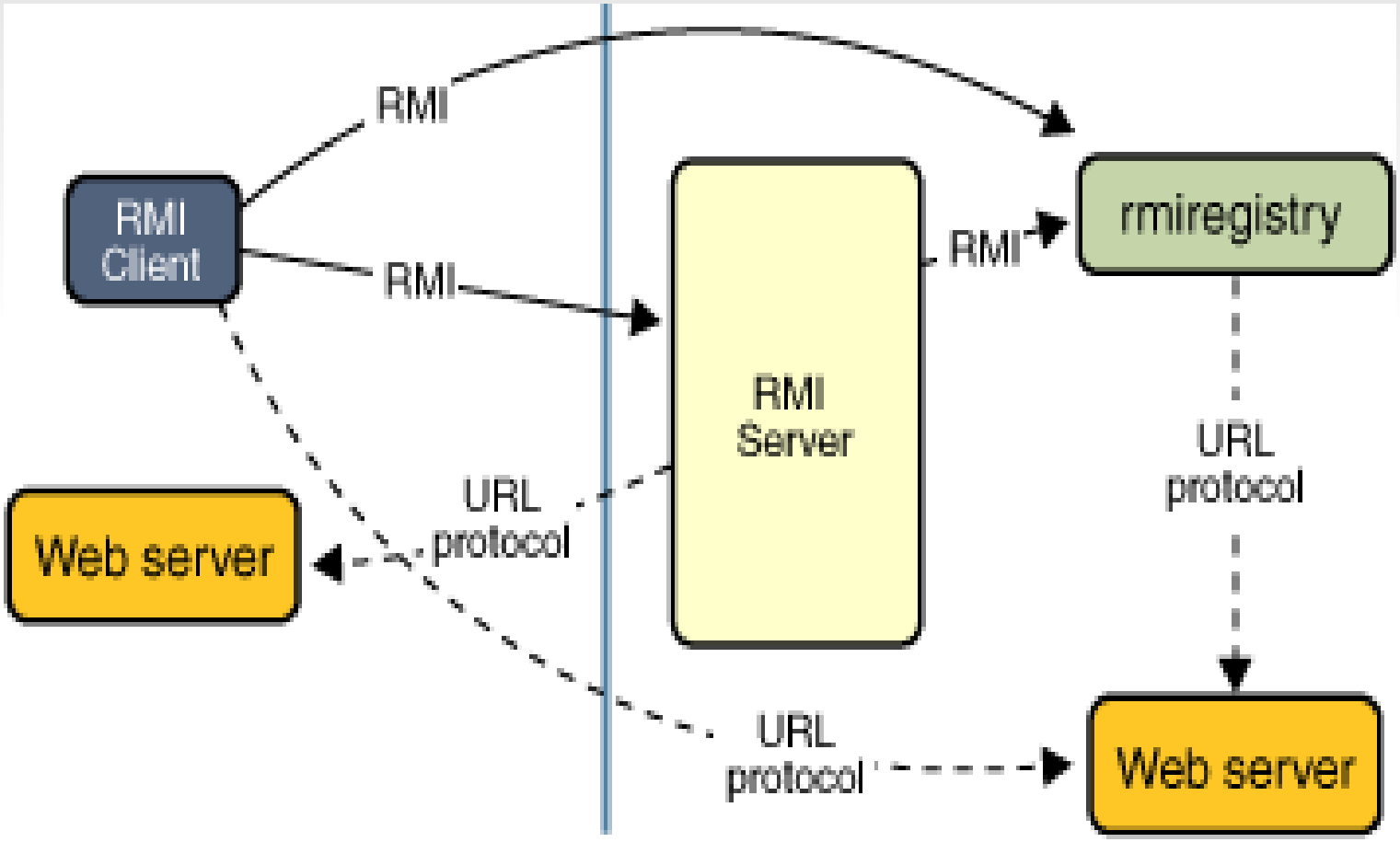
Java RMI

- regra geral as interfaces em compute.jar ficam disponíveis na rede
- na plataforma Java, isto é feito regra geral recorrendo a servidores de HTTP ou FTP

Java RMI

- por exemplo, no caso do HTTP, os servidores possibilitam a colocação de ficheiros de utilizadores em http://web_server/~user_server, que corresponde ao caminho /home/user_server/public_html
- aqui vamos usar o URL, http://web_server/~user_server/classes (/home/user_server/public_html/classes)
- scp compute.jar user_server@web_server:public_html/classes/

Java RMI


RMI
RMI Client
RMI
RMI Server
RMI
rmiregistry
URL protocol
URL protocol
Web server
URL protocol
Web server

Java RMI

para compilar a classe ComputeEngine, a única do lado do servidor, é necessário que as interfaces compute.jar estejam na sua CLASSPATH. Por exemplo:

```
cd /home/user_server/src/
javac  -cp /home/user_server/public_html/classes/compute.jar
server/ComputeServer.java
```

Java RMI

- para compilar as classes do lado do cliente fazemos algo de semelhante para ComputePi e Pi
- aqui vamos usar o URL, http://web_server/~user_client/classes (/home/user_client/public_html/classes)

Java RMI

```
cd /home/user_client/src/

javac -cp /home/user_client/public_html/classes/compute.jar
client/Client.java client/Pi.java
```

Em web_server:

```
mkdir /home/user_client/public_html/classes/client

scp client/Pi.class
user_client@web_server:public_html/classes/client
```

Apenas a classe Pi precisa de ficar disponível na rede pois é ela que é passada como argumento ao método remoto executeTask ()

Java RMI

- os programas cliente e servidor correm com um "security manager" instalado que controla a execução de código de classes locais e remotas (as últimas potencialmente "untrusted") na JVM local
- estes ”security managers” implementam políticas definidas em ficheiros server.policy e client.policy
- nos exemplos seguintes só permitimos a classes locais a realização de operações que precisem de permissões especiais (e.g. acesso a ficheiros, devices)

Java RMI

server.policy

```
grant codeBase "file:/home/user_server/src/" {
      permission java.security.AllPermission;
};
```

client.policy

```
grant codeBase "file:/home/user_client/src/" {
      permission java.security.AllPermission;
};
```

Java RMI

- a propriedade java.rmi.server.codebase é usada para especificar uma lista de URLs onde são mantidas as classes de objectos que são descarregadas da JVM corrente para outras JVMs
- se um programa numa JVM envia um objecto para outra JVM (e.g. como valor de retorno de uma chamada a um método), essa outra JVM necessita do código para a classe associada a esse objecto

Java RMI

rmiregistry &

arranque do servidor:

```
cd /home/user_server/src
java  -cp /home/user_server/src:/home/user_server/public_html/classes/compute.jar
-Djava.rmi.server.codebase=http://web_server/~user_server/classes/compute.jar
-Djava.rmi.server.hostname=server_hostname -Djava.security.policy=server.policy
server.ComputeServer
```

Java RMI

java.rmi.server.codebase especifica o local de onde podem ser descarregadas classes ou interfaces pelos clientes associadas com o servidor (no caso apenas o compute.jar)

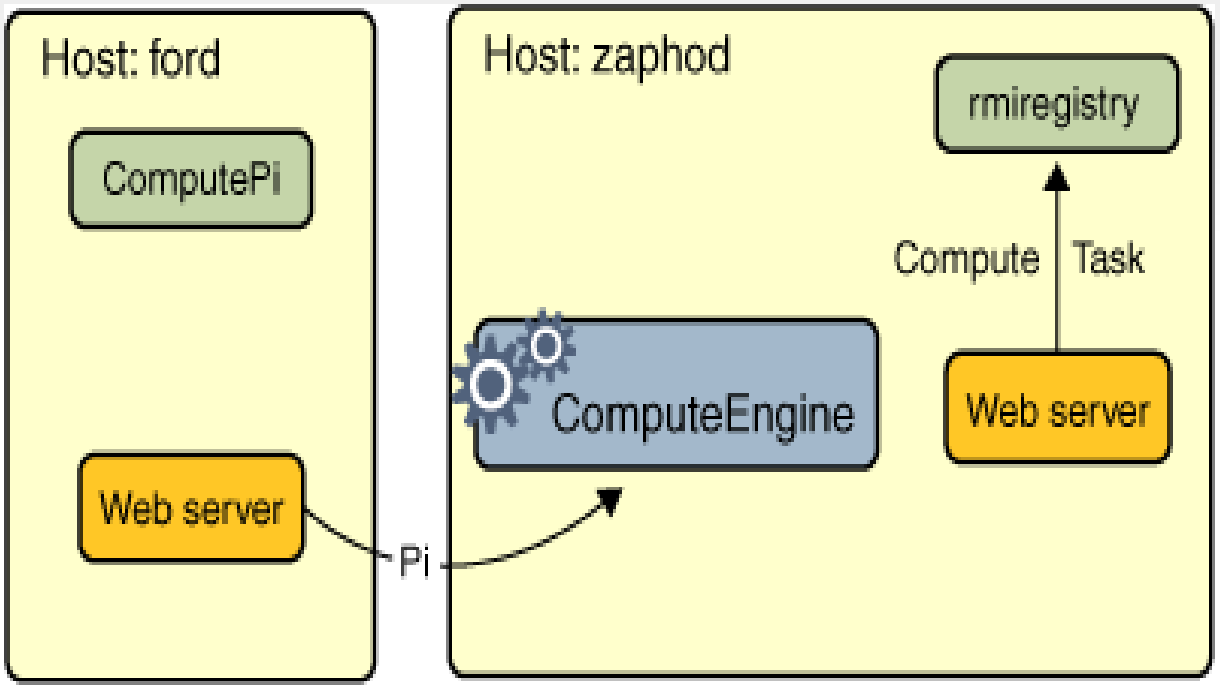
arranque do cliente:

```
cd /home/user_client/src
java -cp
/home/user_client/src:/home/user_client/public_html/classes/compute.jar
    -Djava.rmi.server.codebase=http://web_server/~user_client/classes/
    -Djava.security.policy=client.policy
     client.Client  server_hostname  45
```

Java RMI

- java.rmi.server.codebase especifica onde podem ser descarregada a interface compute.jar e a classe Pi.class a partir da rede
- server_hostname especifica a máquina em que corre o servidor
- 45 especifica a precisão em número de dígitos no valor de pi

Java RMI


Host: ford
ComputePi
Web server
Pi
Host: zaphod
rmiregistry
Compute
Task
ComputeEngine
Web server

Java RMI

- quando o ComputeEngine se regista no registry, o código para as interfaces em compute.jar é descarregado do servidor para o registry, pois o stub/proxy do servidor precisa desse código
- o ComputePi usa apenas as interfaces compute.jar e não recebe classes novas do servidor (note-se que o último não retorna objectos por ele definidos)
- o ComputeEngine, por seu lado, descarrega a classe Pi do web server do cliente pois recebe um objecto do tipo Pi quando o cliente nele invoca o método executeTask com um objecto Pi como argumento