

RMI

Remote Method Invocation

Ivan Lanese, 20/4/2010

Slides originali di Enrico Tassi e Nicola Gessa

Remote Method Invocation

Si colloca nel mondo della programmazione distribuita.

L'obiettivo è di fornire al programmatore un'astrazione sulla collocazione (in rete) degli oggetti che il suo programma utilizza.

Metodi appartenenti a oggetti remoti vengono eseguiti dal calcolatore remoto in modo trasparente.

Si definisce **client** il programma chiamante che ottiene il riferimento all'oggetto remoto, **server** il programma che crea gli oggetti remoti. Tali applicazioni sono anche denominate *distributed object application*

Quali vantaggi?

Semplicità nella gestione delle risorse distribuite
(possibile) Miglioramento delle prestazioni complessive

Es: suddivisione di una computazione pesante in sottoprocedure più piccole, eseguite su macchine diverse, possibilmente in parallelo, diminuendo in tal modo il tempo complessivo di esecuzione.

Sistema distribuito....

.... quindi è necessario un meccanismo di comunicazione

Abbiamo visto i socket nella precedente lezione, ma tale tecnologia non è sufficiente:

- non definisce le modalità di invocazione di metodi
- non definisce protocolli per lo scambio di informazioni

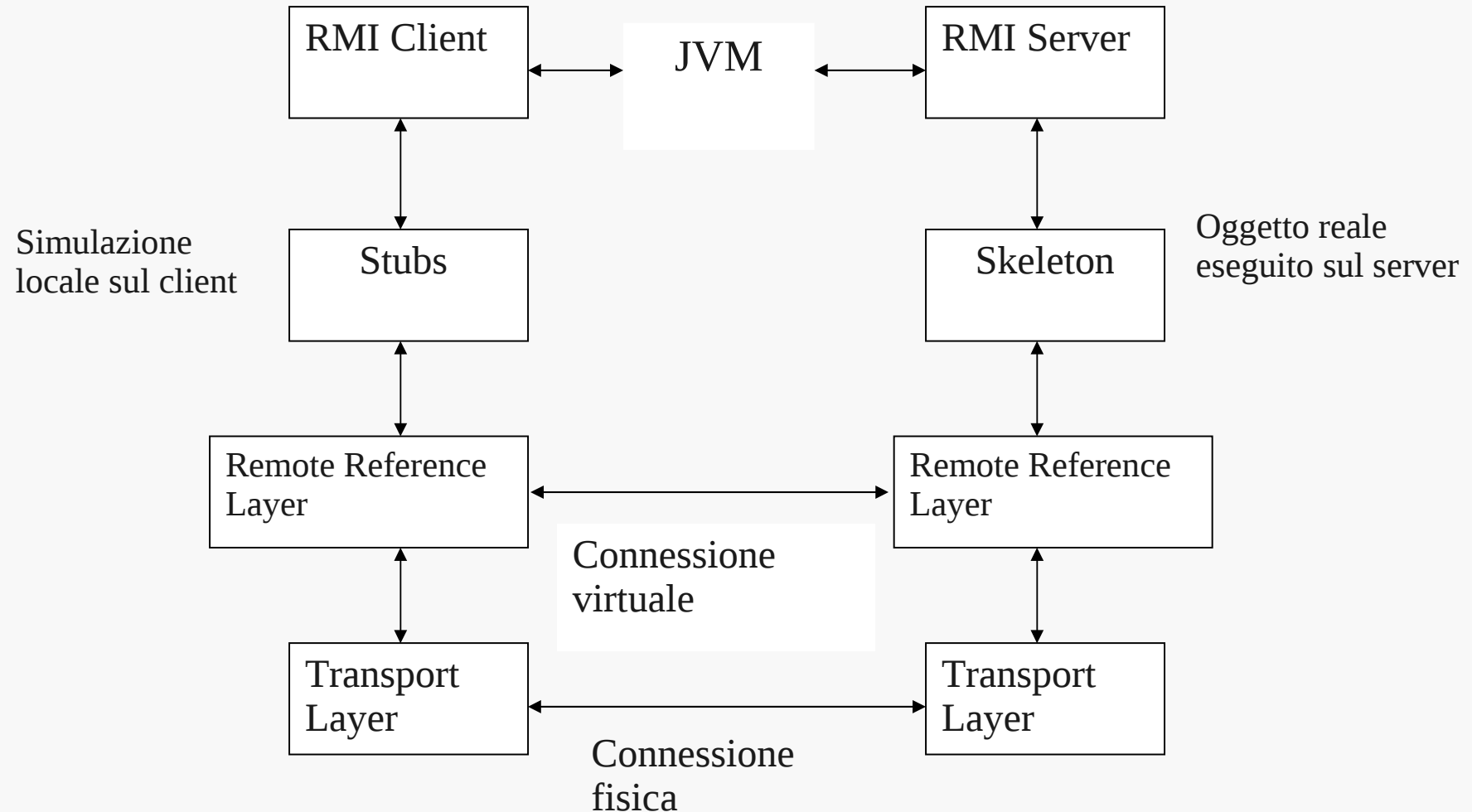
Caratteristiche di RMI

Utilizzando la tecnologia RMI di Java un oggetto in esecuzione su una JVM può richiedere l'esecuzione di un metodo di un oggetto in esecuzione su un'altra JVM.

- RMI richiede l'utilizzo esclusivo di Java come linguaggio di sviluppo:
 - difficile integrazione con altri linguaggi
- Semplicità e velocità nello sviluppo
- Dynamic Code Loading

Architettura dell'RMI

La struttura di un'applicazione RMI è organizzata in strati orizzontali sovrapposti



Architettura dell'RMI

- Lo strato più alto è costituito da applicazioni (client e server) eseguite dalla Java Virtual Machine
- Lo stub e lo skeleton forniscono la rappresentazione dell'oggetto remoto: lo stub gestisce la simulazione locale sul client e agendo come proxy consente la comunicazione con l'oggetto remoto, lo skeleton ne consente l'esecuzione sul server
- Il client esegue i metodi dell'oggetto remoto in modo del tutto analogo alla chiamata locale:

```
ris = OggettoRemoto.nomeMetodo(par1, par2,...)
```

senza preoccuparsi dei dettagli della comunicazione
- Il client ha solo in *minima* parte la sensazione di usare un oggetto remoto

Passaggio di parametri

Percorso dei parametri

- Serializzati dalla Virtual Machine
- Inviati sotto forma di stream al server
- Deserializzati dal server che li utilizza all'interno del corpo del metodo invocato
- Il risultato segue il percorso inverso fino ad arrivare al client

La serializzazione

- Il meccanismo utilizzato da RMI per la trasmissione dei dati fra client e server è quello della serializzazione, che permette il flusso di dati complessi all'interno di stream (può venir usata indipendentemente da applicazioni RMI)
- La serializzazione consiste nella trasformazione automatica di oggetti e strutture in sequenze di byte manipolabili con le classi della "famiglia" Stream del package java.io
- Gli stream sono associabili sia a socket che a file

es:

```
Record record = new Record()
```

```
ObjectOutputStream oos = new ObjectOutputStream(myos)
```

```
oos.writeObject(record);
```

La serializzazione

- Un oggetto è serializzabile se implementa l'interfaccia Serializable

Es:

```
public class Record implements Serializable{
    private String Nome;
    public Record(String Nome) {
        this.Nome = Nome
    }
}
```

La serializzazione

- La serializzazione è ricorsiva (un oggetto serializzabile deve contenere oggetti serializzabili)
- La maggior parte delle classi del JDK è serializzabile (eccetto alcune che adottano strutture dati binarie dipendenti dalla piattaforma)

La serializzazione - UID

- La serializzazione richiede che ogni classe abbia un ID univoco
- Se non viene dichiarato viene creato dal compilatore
- Tool serialver per vederlo
- Utilizzato per controllare che un oggetto deserializzato appartenga alla classe corretta

```
private static final long  
    serialVersionUID = 1234567890L;
```

Gli strati RRL e TL

Si occupano della gestione “a basso livello” (all’interno dell’architettura RMI) della comunicazione

- Il Remote Reference Layer (RRL) ha il compito di instaurare un connessione virtuale fra il client e il server (esegue operazioni di codifica e decodifica dei dati). Questo adotta un protocollo generico e indipendente dal tipo di stub o skeleton utilizzato
- Il Transport Layer esegue la connessione vera e propria tra le macchine utilizzando le funzionalità standard di networking di Java, ovvero i socket (protocollo TCP/IP)

Il Transport Layer

- Il TL è responsabile del controllo dello stato delle varie connessioni
- I dati vengono visti come sequenze di byte da inviare o da leggere
- IL TL si incarica di localizzare il server RMI relativo all'oggetto remoto richiesto
- Esegue la connessione per mezzo di un socket

Programmare con RMI

- Le classi e i metodi sono contenuti nei package `java.rmi` e `java.rmi.server`
- Si definisce *oggetto remoto* un oggetto che implementi l'interfaccia `Remote` i cui metodi possono essere eseguiti da un'applicazione client posta su un'altra macchina virtuale
- Un oggetto diventa remoto se implementa un'interfaccia remota, ossia un'interfaccia che estende l'interfaccia `java.rmi.Remote`

Le classi di oggetti remoti

- Una classe può implementare un numero a piacere di interfacce remote.
- Una classe può estendere una classe che implementa un'interfaccia remota.
- Una classe può definire anche dei metodi che non fanno parte delle interfacce remote e che pertanto sono invocabili solo in locale
- La classe deve supportare accessi concorrenti perché più client potrebbero usarla contemporaneamente.
(Bisogna usare metodi/blocchi **synchronized** dove serve)

Client e server in RMI?

- Sia il “client” che il “server” di un'applicazione possono allocare degli oggetti remoti ed esportarli con un nome sul registry oppure possono passarli come parametri di altri oggetti remoti.
- Dal punto di vista del singolo oggetto remoto il server è chi lo alloca e lo esporta mentre il client è chi richiede un riferimento ad un oggetto remoto.
- In definitiva quindi con RMI tutte le parti di una applicazione distribuita possono agire sia come client che come server.

Come trasformare un oggetto in un oggetto remoto?

Oggetto "classico"

```
public class MyServer{  
    public String concat(String a, String b){  
        return a+b;  
    }  
}
```

Corrispettiva classe remota

```
public interface MyServerInterface extends Remote{  
    public String concat (String a, String b)throws  
        RemoteException;  
}
```

```
public class MyServerImpl implements MyServerInterface  
    extends UnicastRemoteObject{  
    public MyServerImpl() throws RemoteException{}  
    public String concat(String a, String b)throws  
        RemoteException{ return a+b;}  
}
```

Hello World con RMI

- Scrivere l'interfaccia dell'oggetto remoto :
HelloRemoteInterface
- Scrivere il client che richiede l'accesso all'oggetto remoto : Client
- Scrivere una classe HelloImpl che implementa HelloRemoteInterface e estende UnicastRemoteObject
- Scrivere il server che rende accessibile un oggetto della classe HelloRemoteInterface

Hello World con RMI - l'interfaccia

- L'interfaccia remota deve dichiarare tutti i metodi che si vuole poter richiamare in maniera remota
 - deve essere dichiarata *public*
 - deve estendere l'interfaccia *java.rmi.Remote*
 - ogni metodo deve dichiarare *java.rmi.RemoteException* nella sezione *throws* per proteggere l'applicazione da anomalie derivanti dall'utilizzo di risorse remote

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

Implementare l'interfaccia remota

```
public class HelloImpl extends UnicastRemoteObject  
implements Hello{
```

- dichiara che implementa l'interfaccia Hello (con tutti i suoi metodi)
- estende la classe UnicastRemoteObject per consentire la creazione di un oggetto remoto che
 - adotta i protocolli di comunicazione di default di RMI basata sui socket e TCP
 - rimane costantemente attivo

Definire i costruttori

```
public HelloImpl() throws RemoteException{  
    super();  
}
```

- il metodo `super()` chiama il costruttore della classe `UnicastRemoteObject` che esegue le inizializzazioni necessarie per consentire di rimanere in attesa (`listen`) di richieste remote su una porta e poterle gestire (`accept`)
- potrebbe generare l'eccezione `RemoteException` se la connessione non fosse possibile

Implementazione dei metodi remoti

```
public String sayHello(){  
    return "Hello World";  
}
```

- devono essere implementati tutti i metodi dell'interfaccia
- gli argomenti dei metodi e i risultati restituiti devono essere oggetti serializzabili

Esportazione dell'oggetto remoto

- Creare e installare il security manager
- Creare una o più istanze della classe che implementa l'interfaccia remota
- Identificare e registrare tali oggetti con un nome nel rmiregistry
- Deve essere in esecuzione l'utility rmiregistry!

Il security manager

Chi garantisce che il codice scaricato da remoto non esegua operazioni pericolose?

```
if(System.getSecurityManager() == null){  
    System.setSecurityManager(new  
        RMISecurityManager());  
}
```

- Il metodo main deve creare e installare un security manager, che può essere il `RMISecurityManager` o definito facendone una sottoclasse
- Il S.M. garantisce che le classi che vengono caricate non eseguano operazioni per le quali non siano abilitate

Security manager

- La possibilità di scaricare codice è controllata e regolata da un security manager che opera in accordo ad una policy
- Il client e il server devono essere eseguiti con una policy adeguata alle loro esigenze e alle esigenze di sicurezza del sistema specificando il parametro

-Djava.security.policy=**java.policy**

al momento del lancio dell'esecuzione. java.policy è il nome del file dove viene specificata la politica di sicurezza.

I file java.policy

Esempi di tali file possono essere:

```
grant {  
    permission java.security.AllPermission;  
};
```

```
grant {  
    permission java.net.SocketPermission "*" :1024-  
65535", "connect,accept";  
    permission java.net.SocketPermission "*" :80",  
"connect";  
};
```

Istanziare gli oggetti remoti

```
HelloImpl obj = new HelloImpl();
```

- Si devono creare una o più istanze dell'oggetto che fornisce il servizio
- Una volta creato, l'oggetto è pronto per accettare richieste remote

Registrare l'oggetto remoto

```
registry.rebind("//esempio/HelloServer", obj);
```

Il client deve poter ottenere un riferimento all'oggetto remoto:

- RMI fornisce un registry degli oggetti per creare un'associazione (bind) fra un nome URL-formatted e un oggetto
- l'RMI registry è un name-service che consente ai client di ottenere riferimenti agli oggetti: una volta registrato l'oggetto i client possono farne richiesta attraverso il nome e invocarne i metodi
- nessun protocollo deve essere specificato nel primo argomento
- la porta di default su cui risponde il RMIregistry è la 1099, ma può essere cambiata

Registrare l'oggetto remoto

- Un'applicazione può creare un'associazione solo sul RMIregistry del host locale
- I client possono invece eseguire la propria ricerca di oggetti su qualunque host
- L'operazione di registrazione può generare varie eccezioni
 - AlreadyBoundException, se il nome logico è già utilizzato
 - MalformedURLException, per errori nella sintassi dell'URL
 - RemoteException negli altri casi

Fare la scansione di un registry RMI

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.RemoteException;

public class LsRegistry {
    static void ls(String host) throws RemoteException {
        Registry registry = LocateRegistry.getRegistry(host);
        System.out.println(
            "Names bound in registry at "+host+":");
        for(int i=0; i<registry.list().length; i++){
            System.out.println(registry.list()[i]);
        }
    }
}
```

Un client per richiamare gli oggetti remoti

- Deve ottenere un riferimento all'oggetto remoto che vuole richiamare dal RMIRegistry eseguito sul server

```
Obj = (HelloRemoteInterface) registry.  
      lookup("//esempio/HelloServer");
```

- Richiama i metodi dell'oggetto secondo la sintassi solita

```
message = obj.sayHello();
```


Compilazione ed esecuzione

- Compilare i file .java con javac per creare i file .class
- Generare stub e skeleton col comando rmic sul file che contiene l'implementazione dell'oggetto remoto (HelloImpl.class) questo genera i file HelloImpl_Stub.class e HelloImpl_Skel.class (Non più necessario da java 5, ma per dialogare con JVM precedenti serve ancora)
- Lanciare in background sul server l'RMIregistry col comando rmiregistry &. Questo utilizza per default la porta 1099; per utilizzarne un'altra specificarla come parametro (es. rmiregistry 2001 &). Ogni volta che viene modificata un'interfaccia remota deve essere rieseguito
- Lanciare il server
- Lanciare il client

Esempio sulla concorrenza:l'oggetto remoto

Creiamo un'interfaccia remota che consenta di incrementare un valore e proviamo a chiamarla da client diversi.

Cosa succede?

Se vogliamo proteggere parti di codice dei metodi degli oggetti remoti da accessi concorrenti pericolosi dobbiamo definire questo codice all'interno di blocchi synchronized

Esempio sulla concorrenza:l'oggetto remoto

```
package ValNumero;

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

public interface newValNumero extends Remote{
    int getValore() throws RemoteException;
    int setValore(int i) throws RemoteException;
    int incValore() throws RemoteException;
}
```

```
synchronized public int incValore() {
    int j;
    j=this.getValore();
    this.setValore(j+1);
    return j+1;
}
```

Esempio: Implementare un Compute server

- Un compute server è un oggetto remoto che consente ad un server di ricevere dei task dai client, eseguirli e restituire il risultato.
- Il task viene definito dal client ma viene eseguito sulla macchina del server.
- Il task può variare indipendentemente dal server, l'importante è che rispetti una determinata interfaccia
- Il compute server scarica dal client il codice del task e lo esegue all'interno della propria Java virtual machine

Interfacce utilizzate

Per implementare un compute server servono due interfacce

- L'interfaccia Compute, che consenta ai client di inviare task al compute server
- L'interfaccia Task, che consenta al compute server di eseguire i task

Interfaccia Compute

```
package compute;

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

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

Questa interfaccia definisce i metodi che possono essere chiamati da altre virtual machine. Gli oggetti che implementano questa interfaccia diventano oggetti remoti.

Interfaccia Task

```
package compute;  
  
import java.io.Serializable;  
  
public interface Task extends Serializable {  
    Object execute();  
}
```

Questa interfaccia è usata come argomento nel metodo `executeTask` dell'interfaccia `Compute` (definito precedentemente) e fornisce al `Compute Server` il meccanismo per eseguire il task. Non è un'interfaccia remota, quindi non è associata ad oggetti remoti.

Il metodo `execute` deve essere presente in ogni implementazione di questa interfaccia.

Implementazione del Compute Server

```
//oggetto remoto per l'esecuzione dei task

import java.rmi.*;
import java.rmi.server.*;
import compute.*;

public class Server extends UnicastRemoteObject
                        implements Compute
{
    public Server() throws RemoteException {
        super();
    }
    public Object executeTask(Task t) {
        return t.execute();
    }
}
```


Riassunto degli elementi necessari

- Due interfacce :
 - Compute.java
 - Task.java
- Il server che implementa l'interfaccia compute e comprende il metodo main per l'esecuzione: Server.java
- L'implementazione del task da far eseguire: MyTask.java
- Il client che richiede l'esecuzione del task al server: Client.java
- Una java.policy per -Djava.security.policy
- Un RMISecurityManager
- La classe MyTask disponibile via http per -Djava.rmi.server.codebase

Fine