Introduzione alla Remote Methode Evaluation (RMI) (1)

Obiettivo: creare un'applicazione distribuita

→ Varie possibilità:

- **Socket**

  Canali di comunicazione tra processi residenti su host diversi

  E' necessario per ogni applicazione, stabilire un protocollo di trasmissione per la codifica e la decodifica dei dati

  Spostano la programmazione delle applicazioni ad un livello troppo basso

- **Remote Procedure Call (RPC)**

  Astrazione della comunicazione fra i processi al livello di una chiamata di procedura.

  I parametri vengono impacchettati e spediti a destinazione dopo essere stati codificati. Ad una operazione analoga vengono sottoposti i valori restituiti.

  Consente l'invio di solo di tipi di dato primitivi

  La conversione dei dati tra piattaforme diverse introduce un overhead molto elevato

  E' legata al concetto di processo e non si integra nel codice Object Oriented

Introduzione alla Remote Methode Evaluation (RMI) (2)

- **Remote Method Invocation (RMI)**

  Meccanismo che permette ad un'applicazione in esecuzione su una macchina locale di invocare i metodi di un'altra applicazione in esecuzione su una macchina remota

  Viene creato localmente solo il **riferimento ad un oggetto remoto**, che è effettivamente attivo su un host distinto. Un programma invoca i metodi attraverso questo riferimento locale.

  La struttura che si occupa di intercettare le invocazioni dei metodi per trasmetterli (con i relativi argomenti) all'oggetto sul server è denominata ***Object Request Broker*** o ***ORB***.

Tipologie di sistemi RMI (1)

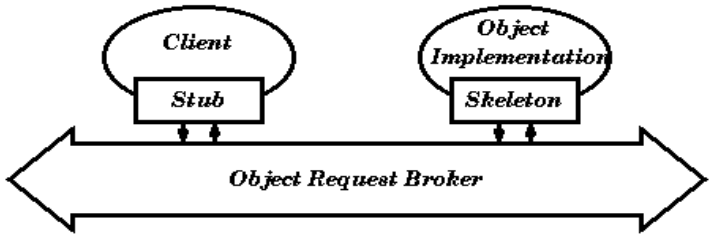
- **Common Object Request Broker Architecture (CORBA)**

  Permette la comunicazione fra **applicazioni scritte in linguaggi differenti** mediante un modello di oggetto neutro.

  Permette la comunicazione di applicazioni preesistenti sviluppate in ambienti eterogenei e per le quali non sia possibile effettuare una conversione in un linguaggio comune.

  Eterogeneità di **ambienti** e di **linguaggi**

  $\Rightarrow$ elevata complessita' del codice



La struttura base di CORBA

Tipologie di sistemi RMI (2)

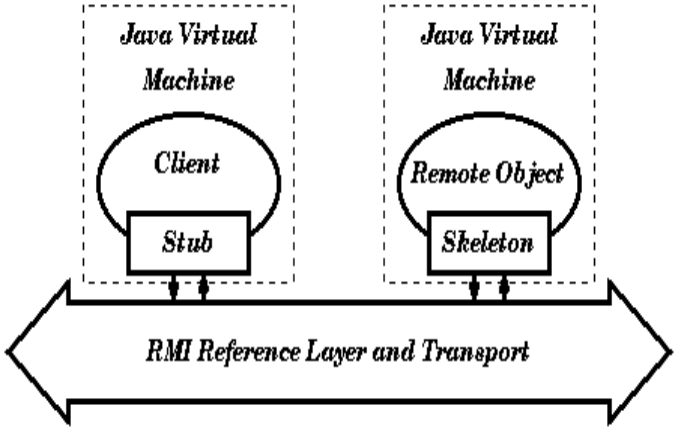
- **Java RMI**

  Permette la comunicazione fra **applicazioni scritte in Java**.

  Eterogeneità di ambienti come conseguenza della portabilità del linguaggio Java

  $\Rightarrow$ codice più semplice

  Con Java RMI si utilizzano oggetti remoti che seguono per quanto possibile il modello ad oggetti di Java



La struttura dell'RMI

Java RMI

Java RMI è un'API che permette ad un'applicazione java in esecuzione su una macchina locale (*client*) di invocare i metodi di un'altra applicazione java (*server*) in esecuzione su una macchina remota

Modello a oggetti distribuito

Nel modello ad oggetti distribuito di Java un oggetto remoto consiste in:

- Un oggetto i cui metodi sono invocati da un'altra JVM presente su un host differente.
- Un oggetto descritto tramite interfacce remote che dichiarano i metodi accessibili da remoto.

Caratteristiche del modello a oggetti distribuito

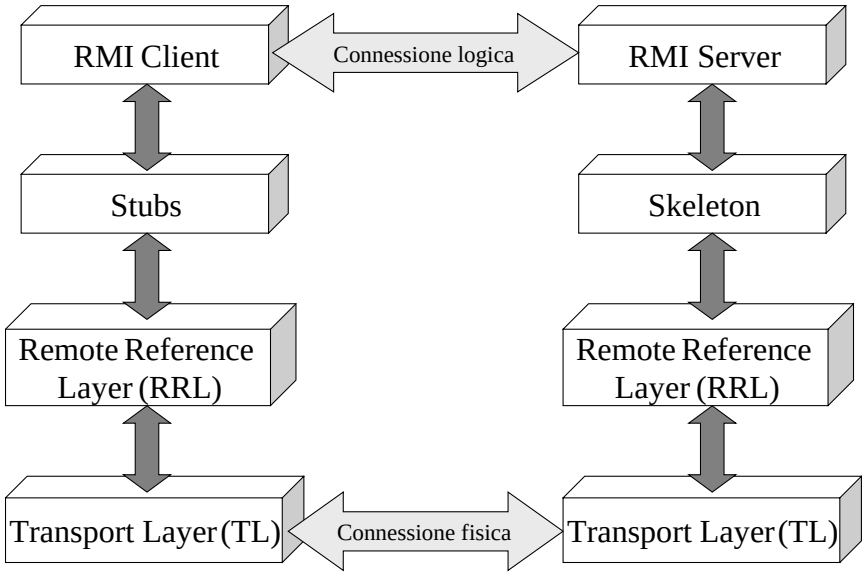
Nel modello distribuito un oggetto differisce nei seguenti aspetti da un oggetto locale:

1. I Client di oggetti remoti interagiscono con le interfacce remote e devono gestire delle eccezioni per eventuali fallimenti di una RMI
2. Gli argomenti ed i risultati dei metodi remoti sono passati per valore e non per riferimento, sfruttando il concetto di *serializzazione*
3. Un oggetto remoto è sempre passato per riferimento.
4. La semantica di alcuni dei metodi definiti dalla classe *Object* è stata resa specifica per gli oggetti remoti.
5. Dei meccanismi di sicurezza sono stati introdotti per il controllo del comportamento delle classi e dei riferimenti.

I modelli coincidono per i seguenti aspetti:

1. Un riferimento ad un oggetto remoto può essere un argomento o un valore di ritorno indistintamente dal tipo di invocazione (locale o remota).
2. Su un oggetto remoto si può effetture il cast ad una qualsiasi delle interfacce remote che implementa.

Architettura di RMI



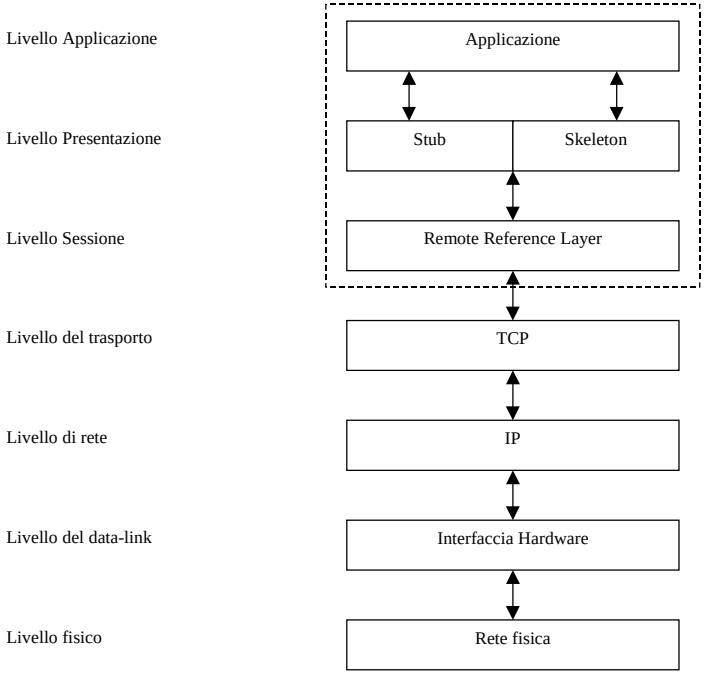
Stub: reference all'oggetto remoto; proxy locale su cui vengono fatte le invocazioni destinate all'oggetto remoto

RRL: instaura un collegamento logico tra i due lati e gestisce la semantica dell'invocazione:
- sul client riceve dallo stub le richieste da inviare al server e le codifica
- sul server decodifica le richieste e le inoltra allo skeleton

Skeleton: elemento che riceve le invocazioni all'oggetto remoto e le realizza comunicando col server

TL: localizza il server RMI relativo all'oggetto remoto richiesto, gestisce le connessioni (con timeout) e le trasmissioni (sequenziali, serializzate)

Gli appropriati stub e skeleton vengono determinati a tempo di esecuzione.



I tre livelli di RMI nel modello OSI

Garbage collection di oggetti remoti

In un sistema distribuito si vuole avere la deallocazione automatica degli oggetti remoti che non hanno più nessun riferimento presso dei client.

- Il sistema RMI utilizza un algoritmo di garbage collection basato sul conteggio dei riferimenti
- Ogni JVM aggiorna una serie di *contatori* ciascuno associato ad un determinato oggetto.
- Ogni contatore rappresenta il numero dei riferimenti ad un certo oggetto che in quel momento sono attivi su una JVM.
- Ogni volta che viene creato un riferimento ad un oggetto remoto il relativo contatore viene incrementato. Per la prima occorrenza viene inviato un messaggio che avverte l'host del nuovo client.
- Quando un riferimento viene eliminato il relativo contatore viene decrementato. Se si tratta dell'ultima occorrenza un messaggio avverte il server.
- Quando nessun client ha più riferimenti ad un oggetto, il runtime di RMI utilizza un "*weak reference*" per indirizzarlo.
- Il *weak reference* è usato nel garbage collector per eliminare l'oggetto nel momento in cui anche dei riferimenti locali non sono più presenti.

La serializzazione

Meccanismo alla base delle trasmissioni di dati fra client e server in RMI

La serializzazione è la trasformazione automatica di oggetti complessi e strutture di oggetti in semplici sequenze di byte, da immettere in uno stream (**`StreamObject`**)

Esempio di **serializzazione:**

```
Record record = new Record();
FileOutputStream fos = new
                    FileOutputStream("data.ser");
ObjectOutputStream oos = new
                           ObjectOutputStream(fos);
oos.writeObject(record);
```

Esempio di **deserializzazione:**

```
FileInputStream fis = new
                    FileInputStream("data.ser");
ObjectInputStream ois = new
                           ObjectInputStream(fis);
record = (Record)ois.readObject();
ois.close();
```

Vincolo: si possono serializzare soltanto istanze di oggetti serializzabili, ovvero che

- implementano l'interfaccia **`Serializable`**
- contengono esclusivamente oggetti serializzabili

Esempio:

```
public class Record implements Serializable {
    private String firstName;
    private String lastName;
    private int phone;
    …
}
```

NOTA BENE:

NON viene trasferito l'oggetto vero e proprio ma solo le informazioni che ne caratterizzano l'istanza (no metodi, no costanti, no variabili **`static`**, no variabili **`transient`**).

Al momento della deserializzazione sarà **ricreata una copia** dell'istanza "trasmessa" usando il .class (che deve quindi essere accessibile!!!) dell'oggetto e le informazioni ricevute.

Non solo `Serializable`…

Con l'interfaccia **`Externalizable`** la trasformazione deve essere implementata direttamente dal programmatore ridefinendo i metodi **`readExternal`** e **`writeExternal`**

RMI in pratica

Per sviluppare un'applicazione distribuita usando RMI si deve:

1. Definire e realizzare i componenti utilizzabili in remoto, che devono necessariamente:
   - implementare un'interfaccia che
     - estenda la **`java.rmi.Remote`** e
     - propaghi la **`java.rmi.RemoteException`**
   - estendere **`java.rmi.UnicastRemoteObject`**

2. Compilare le classi (con **javac**) e generare stub e skeleton (con **rmic**)

3. Rendere le classi accessibili in remoto:
   - il server deve rendere noto il possesso di un oggetto abilitato all'invocazione remota
     - creando un'associazione *nome logico* – *oggetto remoto* e memorizzandola nel registro *rmiregistry* (con **`java.rmi.Naming.bind`**)
     - attivando l'applicazione che gestisce tale archivio (**rmiregistry**)
   - il client deve ottenere il reference all'oggetto remoto (con **`java.rmi.Naming.lookup(String url)`**)

Download del codice da remoto

Per deserializzare un oggetto e istanziarlo è necessario accedere al suo codice

⇒ il client che riceve lo stub deve poter accedere anche ai vari .class

Problemi:

1. localizzare il server da cui scaricare il codice
2. effettuare il download
3. eseguire in modo sicuro il codice scaricato

Soluzioni:

1. informazioni relative a dove reperire il codice memorizzate sul server e passate al client by need

   ⇒ server RMI mandato in esecuzione specificando nell'opzione **`java.rmi.server.codebase`** l'URL da cui prelevare via HTTP le classi necessarie

   Esempio:

   ```
   java –Djava.rmi.server.codebase
    = http://nome_host:port/rmi_dir/ NomeApplicaz
   ```

2. utilizzo di **`RMIClassLoader`**

3. utilizzo di **`RMISecurityManager`**

Esempio di applicazione RMI

Applet che conta gli accessi a una pagina web:

"*L'applet comunica ad una applicazione in esecuzione sul server (attraverso l'RMI) un nuovo accesso alla pagina HTML in cui è inserito, riceve il valore del contatore e lo visualizza sul client.*
L'applicazione sul server aggiorna il contatore e salva il valore su file."

Per la scrittura di questa applicazione seguiremo i seguenti passi:

1. Descrizione dell'interfaccia remota della classe in esecuzione sul server.

2. Implementazione dell'interfaccia.

3. Scrittura di un applet che invochi i metodi remoti della classe sul server.

4. Creazione di una pagina web che contenga l'applet.

L'interfaccia Counter

Defininiamo una interfaccia, per la classe eseguita sul server, che:

- eredita da ***java.rmi.Remote***.
- è *public*
- contiene metodi remoti che dichiarano (tutti) di propagare l'eccezione
  java.rmi.RemoteException.

```
public interface Counter
                    extends java.rmi.Remote {
  String getNumber()
             throws java.rmi.RemoteException;
}
```

NOTA:
Gli oggetti remoti debbono essere dichiarati come le interfacce remote e non come le classi di implementazione.

Ad esempio se si passa un oggetto remoto (istanza di una classe *ObjectImplementation* che implementa l'interfaccia *ObjectInterface*[1]) in un metodo *proc* bisogna dichiarare *proc* nel seguente modo:

```
public void proc(ObjectInterface obj,
                 altri parametri …   ) {
      …
      }
```

[1] Questa interfaccia dovrebbe estendere *java.rmi.Remote.*

Implementazione del server

La classe CounterImpl eredita da

java.rmi.server.UnicastRemoteObject

e implementa l'interfaccia ***Counter***

```
public class CounterImpl extends
UnicastRemoteObject implements
  Counter {
  …
}
```

Definizione del costruttore

Il costruttore di un oggetto remoto **deve** avere l'eccezione *RemoteException* nella sua clausola *throws*.

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

Implementazione dei metodi remoti

- I tipi di dato che sono passati come parametro o restituiti da un metodo remoto devono essere serializzabili.

- I metodi dell'implementazione non dichiarati nell'interfaccia possono essere invocati solo localmente.

```
public String getNumber()
                        throws RemoteException {
  String message;
  users++;
  message="Sei l'utente numero " +
                        String.valueOf(users);
  try{
      FileOutputStream fout = new
                 FileOutputStream("users.txt");
      PrintStream myOutput = new
                             PrintStream(fout);
      myOutput.println(users);}
  catch (IOException e){
      System.out.println("Error: " + e);
      System.exit(1);}
  return  message;}
```


Il metodo main

Il metodo main deve effettuare alcune operazioni necessarie al funzionamento del sistema RMI.

- Installare un Security Manager (*RMISecurityManager* o un altro definito dall'utente) che impedisca alle classi ricevute di effettuare operazioni pericolose.

- Creare una istanza dell'oggetto remoto.

- Registrare l'oggetto remoto sul server con un nome *objectname* e associarlo (binding) a un URL nella forma *//hostname/objectname*

```
Naming.rebind("//myhost:portnumber/CounterServer")
```

Nella fase di binding, se non sono stati specificati il nome dell'host e della porta, sono considerati di default l'host corrente e la porta 1099.

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

    System.setSecurityManager(
                        new RMISecurityManager());
    try{
        CounterImpl obj = new
        CounterImpl("CounterServer");
        Naming.rebind("//myhost:1099/CounterServer"
        , obj);
        System.out.println("CounterServer bound in
        registry");
        }
        catch (Exception e) {
        System.out.println("CounterImpl.main an
        exception occurred: " + e.getMessage());
        e.printStackTrace();
        System.exit(1);
        }
}
```


L'Applet

Il client per invocare un metodo di un oggetto remoto deve ottenerne un riferimento dal registro in esecuzione sul server.

- Il metodo *Naming.lookup* contatta il registro ed ottiene il riferimento.

```
import java.rmi.*;
…
public class CounterApplet extends
java.applet.Applet {
  …
  public void init() {
    …
    try {
      Counter obj =
      (Counter)Naming.lookup("//myhost:1099
      /CounterServer");
      message = obj.getNumber();
    }
    catch (Exception e) {
      System.out.println("CounterApplet
      exception: " +  e.getMessage());
            e.printStackTrace();
    }
    …
  }
  …
}
```

La pagina HTML

Una pagina che comprende il nostro semplice applet potrebbe essere la seguente:

```
<html>
<head>
<title>Counter Page</title>
</head>
<body>
<hr>
<applet code=CounterApplet.class
codebase=./codebase width=200 height=100 >
</applet>
<hr>
<a href="prova.java">The source.</a>
</body>
</html>
```

Compilazione

A questo punto si hanno a disposizione 4 file :

- Counter.java
- CounterImpl.java
- CounterApplet.java
- Prova.html

Compiliamo e otteniamo i file .class:

```
javac[2]  Counter.java
          CounterImpl.java
          CounterApplet.java
```

[2] una directory di destinazione per i file generati viene specificata con l'opzione –d

Se ad esempio si pubblica la pagina HTML nella directory **`/public_html`** e le classi in una sua sub-directory **`codebase`**, utilizziamo:

- in Windows:

```
javac –d c:\public_html\codebase Counter.java
CounterImpl.java CounterApplet.java
```

- in UNIX:

```
javac –d /public_html/codebase Counter.java
CounterImpl.java CounterApplet.java
```

Creazione dello Stub e dello Skeleton

Lo Stub e lo Skeleton vengono generati dal programma ***rmic*** a partire dai file .class, e vengono posti nella stessa directory codebase dell'applet.

Attenzione: *rmic* cerca i file .class nelle directory specificate in *CLASSPATH*[3].

Il comando è: **`rmic CounterImpl`**

Che genera i file:
```
CounterImpl_Stub.class
CounterImpl_Skel.class
```

[3] una comoda assegnazione per *CLASSPATH* è quella contente '.'

Esecuzione

I passi da effettuare sono i seguenti:

- Avviamento del registry.
- Avviamento del server
- Caricamento della pagina con un browser che supporti RMI o con l'Appletviewer.

Avviamento del Registry

Il registry è un tool fornito con il JDK che associa un URL ad un oggetto e si esegue con la linea di comando:

```
rmiregistry port-number
```

Se non viene specificata la porta *port-number*, il registry si pone in ascolto sulla porta 1099.

Avviamento del server

Il server viene eseguito con il comando:

```
java[4] -Djava.rmi.server.codebase=
     http://myhost/~myusername/codebase
                                  CounterImpl.class
```

Per l'esecuzione in background:

- Unix: aggiungere una *&*
- Windows: usare *javaw*

Nell'esempio del Counter si assegna la proprietà *java.rmi.server.codebase* l'URL della directory dove si trova l'applet.
I riferimenti agli oggetti remoti creati dal server includono l'URL dal quale il client potrà dinamicamente scaricare gli stub.

Caricamento della pagina HTML

Utilizziamo l'appletviewer per visualizzare l'applet.
Il comando è:

```
appletviewer indirizzo
```

Nella finestra apparirà il messaggio

'You are the user n. 1'.

[4] L'opzione –D specifica il valore di una proprietà all'avviamento del server.