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 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***.

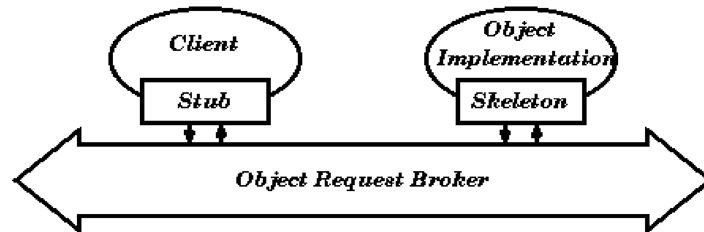
Tipologie di sistemi RMI (1)

- **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**
  ⇒ elevata complessita' del codice



La struttura base di CORBA

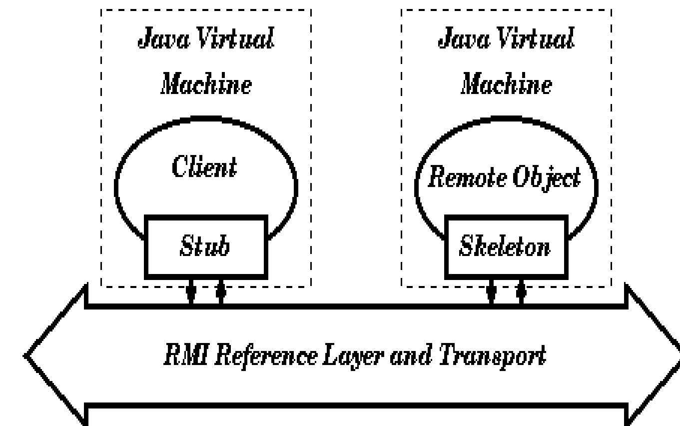
Tipologie di sistemi RMI (2)

- **Java RMI**

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

Eterogeneità di ambienti come conseguenza della portabilità del linguaggio Java
⇒ 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 invocabili da un'altra JVM in esecuzione su un host differente;
- un oggetto descritto tramite interfacce remote che dichiarano i metodi accessibili da remoto.

Concetti estesi al distribuito:

- polimorfismo

- garbage collection

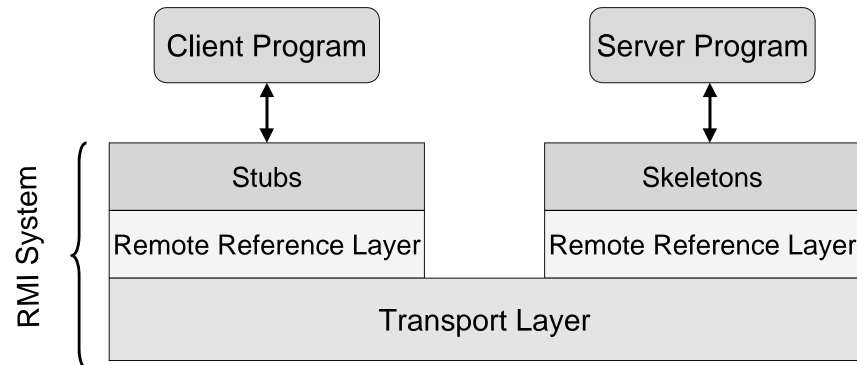
Garbage collection di oggetti remoti

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.

Architettura di RMI



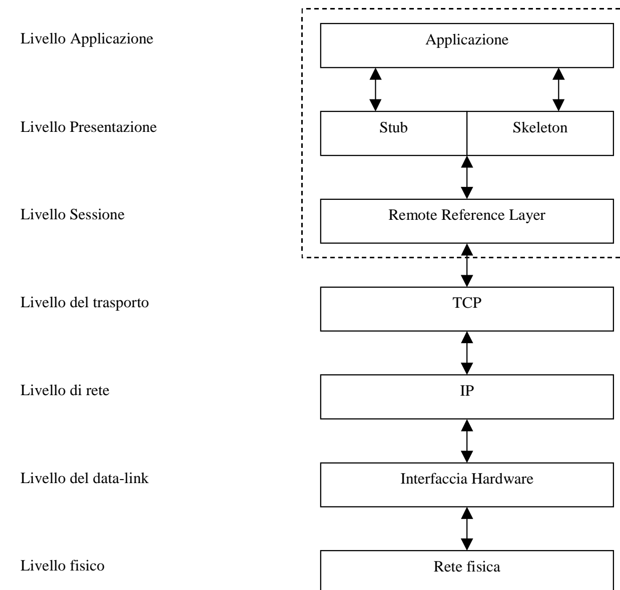
Stub: proxy locale su cui vengono fatte le invocazioni destinate all'oggetto remoto

Skeleton: elemento remoto che riceve le invocazioni fatte sullo stub e le realizza effettuando le corrispondenti chiamate sul server (ormai solo per back compatibility)

Remote Reference Layer:

- fornisce l'oggetto (`RemoteRef`) su cui lo stub inoltra la chiamata (`invoke()`)
- definisce e supporta la semantica dell'invocazione (*at-most-once*) e della comunicazione (*unicast,* con *activable remote object, multicast* in futuro…)

Transport Layer: localizza il server RMI relativo all'oggetto remoto richiesto, gestisce le connessioni (TCP/IP, timeout) e le trasmissioni (sequenziali, serializzate), usando il ***Java Remote Method Protocol (JRMP).*** Fornisce inoltre strategie di penetrazione di Firewall.



I tre livelli di RMI nel modello OSI

La serializzazione

Marshalling: processo di codifica di argomenti e risultati per la trasmissione.
Unmarshalling: processo inverso di decodifica di argomenti e risultati ricevuti.

In Java questo problema è risolto usando la *serializzazione*

Serializzazione: trasformazione automatica di oggetti complessi in semplici sequenze di byte
Deserializzazione: decodifica di una sequenza di byte per costruire una copia dell'oggetto originale

Esempio di codice per **serializzare:**

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

Esempio di codice per **deserializzare:**

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


Si possono serializzare soltanto istanze di oggetti serializzabili, ovvero che:

- implementano l'interfaccia **`Serializable`**
- contengono esclusivamente oggetti (o riferimenti a oggetti) serializzabili

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`**

Usato per motivi di efficienza (serializzazione usa reflection, meccanismo flessibile ma costoso)

Passaggio di parametri

Tipo	Metodo Locale	Metodo Remoto
Tipi primitivi	Per valore	Per valore
Oggetti	Per riferimento	Per valore (deep copy)
Oggetti Remoti Esportati	Per riferimento	Per riferimento remoto

Shallow Copy vs Deep Copy

Passaggio per valore => Serializable Objects
Passaggio per riferimento => Remote Objects

Serializable Objects
Oggetti la cui locazione non è rilevante per lo stato.
Sono passati **per valore**: ne viene serializzata l'istanza che sarà deserializzata a destinazione per crearne una copia locale.

Remote Objects
Oggetti la cui funzione è strettamente legata alla località in cui eseguono (server).
Sono passati **per riferimento**: ne viene serializzato lo stub, creato automaticamente dal proxy (stub o skeleton) su cui viene fatta la chiamata in cui compaiono come parametri

Interazione Client/Server: Socket e Thread

Server:
L'esportazione di un oggetto remoto provoca la creazione di
- una **socket d'ascolto** (ServerSocket)
- un **listener** (Thread)

associati all'oggetto remoto.

RMI permette a più thread di accedere ad uno stesso oggetto server (gestione concorrente delle richieste)

=> **l'implementazione dei metodi remoti deve essere *thread-safe***

Client:
L'invocazione di un metodo remoto implica la creazione di una connessione (Socket) con l'oggetto remoto, o il riuso di una creata in precedenza

RMI utilizza la **condivisione delle socket tra la JVM client e la JVM server,** cercando di riutilizzarle quando possibile

=> numero di socket allocate pari al numero di richieste contemporanee

Stub e Skeleton

Stub e Skeleton sono oggetti generati dal compilatore RMI che gestiscono

marshalling/unmarshalling e
comunicazione (socket) tra client e server

Procedura di comunicazione:
1. il client ottiene un'istanza dello stub
2. il client chiama metodi sullo stub
3. lo stub:
   - crea una connessione con lo skeleton (o ne usa una già esistente)
   - marshalling delle informazioni associate alla chiamata (identificativo del metodo e argomenti)
   - invio delle informazioni allo skeleton
4. lo skeleton:
   - unmarshalling dei dati ricevuti
   - effettua la chiamata sull'oggetto che implementa il server
   - marshalling del valore di ritorno e invio allo allo stub
5. lo stub:
   - unmarshalling del valore di ritorno e restituzione del risultato al client

Stub

Estende java.rmi.server.RemoteStub
Implementa java.rmi.Remote
InterfacciaRemotaServer

Realizza le chiamate ai metodi dell'interfaccia remota del server usando il **metodo `invoke(…)`** di **`java.rmi.server RemoteRef`**

Skeleton

Implementa java.rmi.server.Skeleton

Inoltra le richieste al server usando il **metodo `dispatch(…)`**

Interfacce e Implementazione

Separazione tra

definizione del comportamento => interfacce

implementazione del comportamento => classi

Per realizzare componenti utilizzabili in remoto:

1. definire comportamento -> interfaccia che
   - estende `java.rmi.Remote` e
   - propaga `java.rmi.RemoteException`

2. implementare comportamento -> classe che
   - implementa l'interfaccia
   - estende `java.rmi.UnicastRemoteObject`

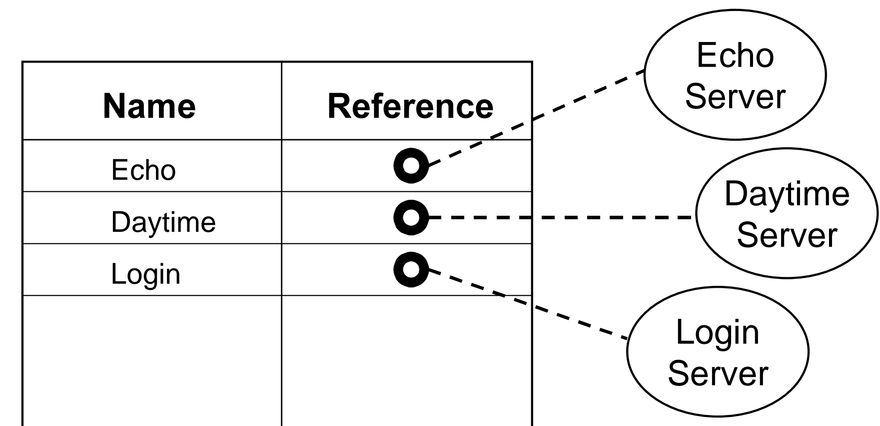
RMI Registry

Problema di ***Bootstrapping***: un client in esecuzione su un macchina ha bisogno di localizzare un server a cui vuole connettersi, che è in esecuzione su un'altra macchina. Tre possibili soluzioni:

- Il client conosce in anticipo dov'è il server
- L'utente dice all'applicazone client dov'è il server (es. e-mail client)
- Un servizio standard (***naming service***) in una locazione ben nota, che il client conosce, funziona come *punto di indirezione*

Java RMI utilizza un naming service: ***RMI Registry***

Mantiene un insieme di coppie **{name, reference}**
Name: stringa arbitraria non interpretata



Operazioni

Metodi della classe ***java.rmi.Naming***:

```
public static void bind(String name, Remote obj)
public static void rebind(String name, Remote obj)
public static void unbind(String name)
public static String[] list(String name)
public static Remote lookup(String name)
```

name -> combina la locazione del registry e il nome logico del servizio, nel formato:

//registryHost:port/logical_name

A default:
registryHost = macchina su cui esegue il programma che invoca il metodo
port = 1099

Il Registry è un server RMI
in esecuzione su *registryHost*
in ascolto su *port*

Ognuno di questi metodi crea una connessione (socket) con il registry identificato da host e porta

Implementazione del Registry

Il Registry è un server RMI

Classe d'implementazione:
sun.rmi.registry.RegistryImpl

Interfaccia: *java.rmi.registry.Registry*

```
public interface Registry extends Remote {

    public static final int REGISTRY_PORT = 1099;

    public Remote lookup(String name)
        throws RemoteException, NotBoundException,
                                AccessException;
    public void bind(String name, Remote obj)
        throws RemoteException, AlreadyBoundException,
                                AccessException;
   public static void rebind(String name, Remote obj)
           throws  RemoteException, AccessException;
    public static void unbind(String name)
        throws RemoteException, NotBoundException,
                                AccessException;
          public static String[] list(String name)
            throws  RemoteException, AccessException;
}
```

Perchè usare anche la classe Naming?

Problema di bootstrapping anche per localizzare il registry -> uso delle classi Naming e LocateRegistry

- Naming: metodi statici, no istanza
- LocateRegistry implementa i metodi:

```
public static Registry createRegistry(int port)
public static Registry createRegistry(int port,
                    RMIClientSocketFactory csf,
                    RMIServerSocketFactory ssf)
public static Registry getRegistry()
public static Registry getRegistry(int port)
public static Registry getRegistry(String host)
public static Registry getRegistry(String host,
                                        int port)
public static Registry getRegistry(String host,
             int port, RMIClientSocketFactory csf)
```

Soluzione al bootstrapping per il registry:

1. Naming."metodo()" con host e porta
2. invocazione di LocateRegistry.getRegistry() che restituisce lo stub del registry
3. invocazione di "metodo()" sullo stub

Uso del Registry

Due possibilità:

- programma ***rmiregistry*** di Sun, lanciato specificando o meno la porta:

```
rmiregistry
rmiregistry 10345
```

⇨ istanza separata della JVM
⇨ struttura e comportamento standard

- creare all'interno del codice un ***proprio registry***:

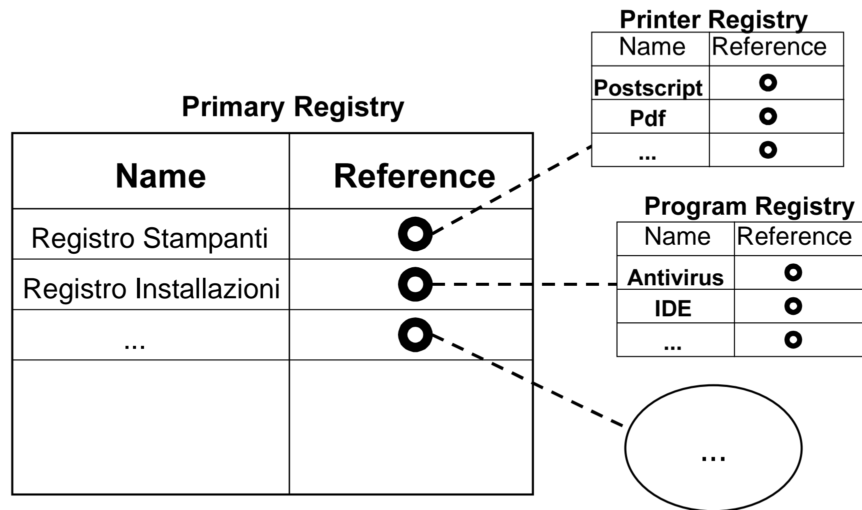
```
public static Registry createRegistry(int port)
```

⇨ stessa istanza della JVM
⇨ struttura e comportamento personalizzabile (organizzazione gerarchica, uso di SocketFactory)

Organizzazione gerarchica

Un registry è un server RMI
=> è possibile registrarlo in un altro registry

```
Registry ourRegistry =
      LocateRegistry.createRegistry(OUR_PORT);
Registry preexistingRegistry =
              LocateRegistry.getRegistry(1099);
preexixtingRegistry.rebind
           ("secondary registry", ourRegistry);
```



Sicurezza del registry

Problema: accedendo al registry (individuabile interrogando tutte le porte di un host) è possibile ridirigere per scopi maliziosi le chiamate ai server RMI registrati (es. `list()+rebind()`)

Soluzione: i metodi bind(), rebind() e unbind() sono invocabili **solo dall'host su cui è in esecuzione il registry** -> no modifiche della struttura client/server dall'esterno

- ⇨ sull'host in cui vengono effettuate le chiamate al registry deve essercene almeno uno in esecuzione

Distribuzione delle classi

In una applicazione RMI è necessario che siano disponibili gli opportuni file .class nelle località dove necessario (per l'esecuzione o per la deserializzazione)

Il Server deve poter accedere a:

- interfacce che definiscono il servizio
- implementazione del servizio
- stub e skeleton delle classi di implementazione
- altre classi utilizzate dal server

Il Client deve poter accedere a:

- interfacce che definiscono il servizio
- stub delle classi di implementazione del servizio
- classi del server usati dal client (es. valori di ritorno)
- altre classi utilizzate dal client

E' necessario:

1. localizzare il codice (in locale o in remoto)
2. effettuare il download (se in remoto)
3. eseguire in modo sicuro il codice scaricato

Localizzazione del codice

Le informazioni relative a dove reperire il codice sono 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 le classi necessarie.

L'URL può essere:
una directory del file system locale (**file://**)
l'indirizzo di un server ftp (**ftp://**)
l'indirizzo di un server http (**http://**)

Il codebase è una proprietà del server che viene annotata nel reference pubblicato sul registry

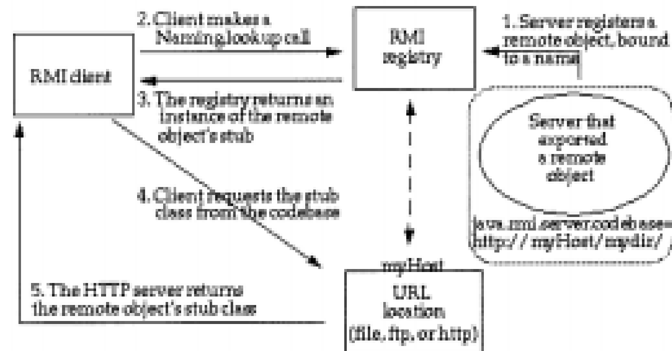
Esempio:

```
java   -Djava.rmi.server.codebase
              = http://nome_host:port/rmi_dir/
       NomeApplicazione
```

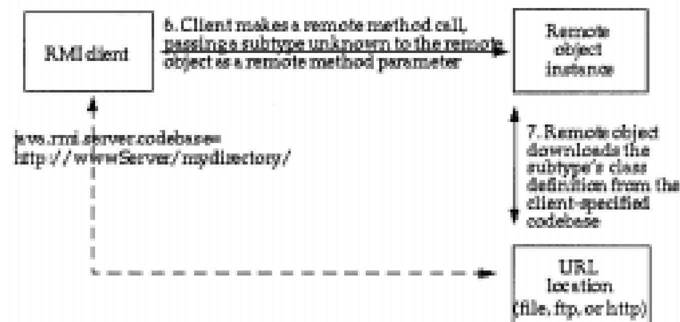
Le classi vengono cercate sempre **prima nel CLASSPATH locale**, solo in caso di insuccesso vengono cercate nel codebase

Utilizzo del codebase

Il codebase viene usato dal client per scaricare le classi necessarie relative al server (interfaccia, stub, oggetti restituiti come valori di ritorno)



Il codebase viene usato dal server per scaricare le classi necessarie relative al client (oggetti passati come parametri nelle chiamate)



Download del codice

Utilizzo di **`RMIClassLoader`**

usato dal supporto RMI per caricare le classi

Esecuzione del codice

Utilizzo di **`RMISecurityManager`**

Istanziato all'interno dell'applicazione RMI
(se non ce n'è già uno istanziato)

Sia il server che il client devono essere lanciati specificando il file con le autorizzazioni:

```
-Djava.security.policy=esempio.policy
```

Dove esempio.policy è del tipo:

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

Uso di RMI

Per sviluppare un'applicazione distribuita usando RMI si deve:

1. Definire le interfacce e le implementazioni dei componenti utilizzabili in remoto

2. Compilare (con **javac**) le classi e generare stub e skeleton (con **rmic**) delle classi utilizzabili in remoto

3. Il servizio deve essere pubblicato
   - attivazione del registry
   - registrazione del servzio

4. Il server deve essere attivato (a meno che non sia attivabile da remoto, Remote Object Activation)

5. Il client deve ottenere il reference all'oggetto remoto
   - lookup sul registry

A questo punto l'interazione tra il client e il server può procedere, e by need vengono scaricate le classi necessarie utilizzando il codebase

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

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

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

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

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.

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

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’.

Bibliografia

Sito della Sun:
http://java.sun.com/products/jdk/rmi/

W.Grosso, ***"Java RMI"***, Ed. O'Reilly (2002)

E. Pitt, K. McNiff, ***"java.rmi, The Remote Methode Invocation Guide"***, Ed. Addison Wesley (2001)

R. Öberg, ***"Mastering RMI, Developing Enterprise Applications in Java and EJB"***, Ed. Wiley (2001)