Java

Remote Method Invocation -- RMI

G. Prencipe
prencipe@di.unipi.it

RMI

- RMI è una tecnologia JAVA che permette a una JVM di comunicare con un'altra JVM per farle eseguire metodi
- È possibile che oggetti invochino metodi su altri oggetti remoti come se questi ultimi fossero locali
- L'oggetto remoto può a sua volta restituire il risultato della computazione, inviandolo al richiedente

RMI

- Ogni servizio RMI è definito da un'interfaccia che descrive i metodi che possono essere eseguiti in remoto
- Questa interfaccia deve essere condivisa da tutti gli sviluppatori che scriveranno software per quel servizio
- Può essere creata più di una implementazione dell'interfaccia, e gli sviluppatori non devono necessariamente conoscere quale implementazione è utilizzata o dove essa sia localizzata

RMI

- I sistemi che utilizzano RMI per comunicare sono tipicamente divisi in client e server
  - Il server fornisce un servizio RMI, e un cliente invoca un metodo di quel servizio
- I server RMI devono registrarsi (presso un lookup service) per permettere i clienti di poterli trovare
  - In alternativa possono rendere disponibile un riferimento al servizio in altri modi

RMI

- Inclusa come parte della piattaforma JAVA c'è un'applicazione chiamata **rmiregistry**
  - È un processo separato che permette alle applicazioni di registrare servizi RMI o di ottenere un riferimento a un certo servizio
- Ad ogni registrazione è associato un nome (rappresentato come una **String**a), che viene utilizzato dai clienti per selezionare il servizio appropriato
- Se un servizio si sposta da un server a un altro, il cliente deve solo cercare nel registro per conoscere la nuova locazione

RMI

- Una volta che un servizio è registrato, esso attende per richieste provenienti dai clienti
- I clienti inviano messaggi RMI per invocare metodi di oggetti remoti
- Il cliente, prima di poter effettuare l'invocazione, deve ottenere un riferimento all'oggetto remoto
  - Esso viene tipicamente ottenuto cercando il servizio nel registro
  - L'applicazione cliente richiede il nome di una particolare applicazione, e riceve una URL alla risorsa remota

RMI

- Il seguente formato di URL è utilizzato per rappresentare un riferimento a un oggetto remoto

  rmi://hostname:port/servicename
- Una volta ottenuto il riferimento, il cliente può interagire con il servizio remoto
- I dettagli della rete legati alla comunicazione sono completamente trasparenti allo sviluppatore dell'applicazione
  - Si utilizzano gli oggetti remoti come se fossero locali
- Questo si ottiene grazie alla divisione del sistema RMI in due parti: **stub** e **skeleton**

stub

- L'oggetto **stub** agisce dal lato cliente, e ha il compito di convogliare le richieste al server RMI remoto
  - Funziona un po' come un proxy
- Ogni servizio RMI è definito come un'interfaccia, non come un'implementazione
- L'oggetto **stub** implementa una particolare interfaccia RMI, che l'applicazione cliente può usare come ogni altra implementazione di oggetti

stub

- Lo **stub**, però, invece che fare il lavoro da solo, passa il messaggio (richiesta) al servizio RMI remoto, attende la risposta, e la restituisce al metodo invocante
- Lo sviluppatore dell'applicazione non ha bisogno di conoscere dove si trova la risorsa RMI, su quale piattaforma gira, o come essa risponderà
- Il cliente RMI semplicemente invoca un metodo dell'oggetto **stub**; esso gestisce tutti i dettagli dell'implementazione

skeleton

- Lo **skeleton** è la controparte dello stub sul server
- È responsabile di ricevere le richieste RMI, e di passarle al servizio RMI
  - Lo **skeleton** non fornisce un'implementazione di un servizio RMI
  - Funge solo da ricevitore di richieste
  - Lo **skeleton** invoca l'implementazione dell'interfaccia associata al servizio RMI e invia la risposta allo **stub** nel cliente

Comunicazione

- La comunicazione (scambio dei messaggi) avviene tra lo **stub** e lo **skeleton** utilizzando socket TCP
- Lo **skeleton**, una volta creato, si pone in ascolto di richieste provenienti dallo **stub**
- I parametri che possono essere scambiati in un sistema RMI non si limitano a tipi primitivi
  - Ogni oggetto che è *serializzabile* (*serializable*) può essere inviato come parametro o restituito da un metodo remoto
  - Quando lo **stub** invia una richiesta, deve impacchettare i parametri: *data marshalling*
  - Lo **skeleton** ricostruisce i parametri: *data unmarshalling*

Serializzazione

- La *serializzazione* degli oggetti permette di prendere un oggetto che implementa l'interfaccia **Serializable** e trasformarlo in una sequenza di **byte**s
  - Crea un'immagine dell'oggetto
- Successivamente è possibile prendere questa sequenza e ricomporla per rigenerare l'oggetto di partenza

Serializzazione

- È molto semplice serializzare un oggetto: è sufficiente implementare **Serializable** (che non ha metodi!!)
- Per utilizzare la serializzazione bisogna
  - Creare un oggetto **OutputStream** e racchiuderlo in un oggetto **ObjectOutputStream oos**
  - A questo punto si chiama **oos.writeObject(Object o)** e l'oggetto **o** è serializzato e inviato all'**OutputStream**
  - Simile per la lettura (**ObjectInputStream** e **readObject**)
  - Chiaramente in lettura si ottiene un riferimento a **Object**

Serializzazione

- Nella serializzazione di un oggetto vengono salvati i campi non-statici e non-transienti (vedremo dopo) dell'oggetto
- Inoltre quando si serializza un oggetto vengono salvati anche i suoi riferimenti ad altri oggetti (purchè implementino a loro volta **Serializable**)

L'interfaccia di un servizio RMI

- Ogni sistema che utilizza RMI deve definire un'interfaccia
- L'interfaccia definisce i metodi che possono essere invocati in remoto
  - Specifica anche parametri, tipi di ritorno, e eccezioni
- **Stub**, **skeleton**, e il servizio RMI implementano quest'interfaccia

L'interfaccia di un servizio RMI

- Tutte le interfacce di un servizio RMI estendono l'interfaccia **java.rmi.Remote**
  - Solo i metodi definiti in una interfaccia **java.rmi.Remote** possono essere invocati in remoto
  - Gli altri metodi sono nascosti ai clienti RMI

Esempio: LightBulb

```
public interface RMILightBulb extends
   java.rmi.Remote
{
   public void on () throws java.rmi.RemoteException;
   public void off() throws java.rmi.RemoteException;
   public boolean isOn() throws
                              java.rmi.RemoteException;
}
```

Implementare un servizio RMI

- Una volta che l'interfaccia di un servizio è definita, bisogna implementarla
- Questa implementazione fornisce le funzionalità per ognuno dei metodi definit
  - Può introdurre anche metodi aggiuntivi, che non aranno però accessibili in remoto
  - Gli unici accessibili in remoto sono infatti i metodi definiti nell'interfaccia

Esempio: LightBulb

```
public class RMILightBulbImpl
        extends java.rmi.server.UnicastRemoteObject
                                    implements RMILightBulb
{
   // Definizione del costruttore
   public RMILightBulbImpl() throws java.rmi.RemoteException
   {
        // Valore di default
        setBulb(false);
   }
```

Esempio: LightBulb

```
// Boolean flag: conserva lo stato della lampadina
   private boolean lightOn;
// Metodo accessibile in remoto
   public void on() throws java.rmi.RemoteException
   {
        setBulb (true);
   }
// Metodo accessibile in remoto
   public void off() throws java.rmi.RemoteException
   {
        setBulb (false);
   }
```

Esempio: LightBulb

```
// Metodo accessibile in remoto
   public boolean isOn() throws
                        java.rmi.RemoteException
        {return getBulb();}
// Metodo locale
   public void setBulb (boolean value)
                                {lightOn = value;}
// Metodo locale
   public boolean getBulb ()  {return lightOn;}
}
```

Creare **stub** e **skeleton**

- Le classi **stub** e **skeleton** sono responsabili di inoltrare e processare le richieste RMI
- Gli sviluppatori non dovrebbero scrivere direttamente queste classi
- Una volta che l'implementazione del servizio è scritta, il tool **rmic** (distribuito con la JDK), dovrebbe essere usato per creare **stub** e **skeleton**

Creare **stub** e **skeleton**

- Una volta compilate l'implementazione e l'interfaccia, si esegue

  rmic implementation
- Dove **implementation** è il nome della classe relativa all'implementazione
- Nel nostro esempio

  rmic RMILightBulbImpl
- Vengono generati due file: **RMILightBulbImpl_Stub.class** e **RMILightBulbImpl_Skeleton.class**
- Lo **stub** deve essere distribuito a tutit i clienti

Creare un server RMI

- Il server RMI è responsabile della creazione di un'istanza di una implementazione del servizio, e successivamente della sua registrazione presso il registro (rmiregistry)
- Semplice da fare con poche linee di codice
- Vediamo

Esempio: LightBulb

```
import java.rmi.*;
import java.rmi.server.*;
public class LightBulbServer {
   public static void main(String args[]){
      System.out.println ("Loading RMI service");
      try{
            // Crea un'istanza del servizio
            // e ottiene un riferimento
            RMILightBulbImpl bulbService = new
                              RMILightBulbImpl();
```

Esempio: LightBulb

```
//Stampa l'esatta localizzazione del servizio
//Ogni servizio è legato a una porta TCP, e il rif.
//è composto da un hostname e una porta
      RemoteRef location = bulbService.getRef();
      System.out.println (location.remoteToString());
//Il passo successivo è di registrare il servizio
//1. Controlla se un registro è stato specificato
//Il default è su localhost
      String registry = "localhost";
            if (args.length >=1){registry = args[0];}
```

Esempio: LightBulb

```
//Formato registrazione
//registry_hostname (opzionale):port /service
   String registration = "rmi://" + registry +
                                    "/RMILightBulb";
//2. Effettua la registrazione in modo tale che i clienti
//      possano trovare il servizio
// Si utilizzano i metodi statici della classe
   java.rmi.Naming, responsabili per recuperare e
   piazzare riferimenti remoti a oggetti nel registro
      Naming.rebind(registration, bulbService );
```

Esempio: LightBulb

```
// Cattura delle eccezioni
   }
      catch (RemoteException re){
            System.err.println ("Remote Error - " + re);
      }
      catch (Exception e){
            System.err.println ("Error - " + e);
      }
   }
}
```

Creare un cliente RMI

- Anche la scrittura di un cliente non è complicata
- Il cliente necessita solo di ottenere un riferimento all'interfaccia remota, e non deve sapere come i messaggi sono scambiati, o l'effettiva locazione del servizio
- Per trovare il servizio inizialmente, viene effettuata una ricerca nel registro RMI
- Successivamente, il cliente può invocare i metodi remoti dell'interfaccia del servizio come se fossero invocati su un oggetto locale
- Vediamo

Esempio: LightBulb

```
import java.rmi.*;
public class LightBulbClient {
   public static void main(String args[]){
        System.out.println ("Looking for light bulb
                                                service");
        try {
// Controlla se un registro è stato specificato
              String registry = "localhost";
              if (args.length >=1){
                    registry = args[0];}
```

Esempio: LightBulb

```
String registration = "rmi://" + registry +
                                "/RMILightBulb";
// Cerca il servizio nel registro
   Remote remoteService = Naming.lookup
                                (registration);
// Cast alla interfaccia RMILightBulb
// Se non viene trovata l'interfaccia specificata
// viene lanciata una NotBoundException
   RMILightBulb bulbService = (RMILightBulb)
                                remoteService;
```

Esempio: LightBulb

```
// Accende
System.out.println ("Invoking bulbservice.on()");
bulbService.on();
// Controlla lo stato
System.out.println ("Bulb state : " +
                        bulbService.isOn()  );
// Spegne
System.out.println ("Invoking bulbservice.off()");
bulbService.off();
System.out.println ("Bulb state : " +
                        bulbService.isOn() );
```

Esempio: LightBulb

```
// Gestione delle eccezioni
}
        catch (NotBoundException nbe){
                System.out.println ("No light bulb service
   available  in registry!");
        }
        catch (RemoteException re){
                System.out.println ("RMI Error - " + re);
        }catch (Exception e){
                System.out.println ("Error - " + e);}}}
```

Lanciare il sistema RMI

- Lanciare un sistema RMI ha bisogno di alcune attenzioni, e bisogna seguire un ordine preciso nel lanciare le applicazioni
- Prima che il cliente possa invocare metodi, il registro deve essere in esecuzione
- Nemmeno il servizio può essere lanciato prima che il registro sia in esecuzione, altrimenti viene lanciata un'eccezione al momento della registrazione
- Lo **stub** deve essere fornito ai clienti

Lanciare il sistema RMI

1. Copiare tutti i file necessari sulle macchine dei clienti e del server (**stub** e **skeleton**)
2. Eseguire **rmiregistry**
3. Eseguire il server
4. Eseguire i clienti

Java

Remote Method Invocation -- RMI

fine

Problemi

- Un problema tipico con i servizio RMI è legato alla distribuzione dello **stub** ai clienti
- Può risultare semplice se si conosce la localizzazione die clienti
- Altrimenti non è semplice
- Questi problemi vengono risolti utilizzando la tecnica del *dynamic class loading*