RMI - REMOTE METHOD INVOCATION

Un esempio funzionante

Gabriele Vailati - Tarcisio Zago

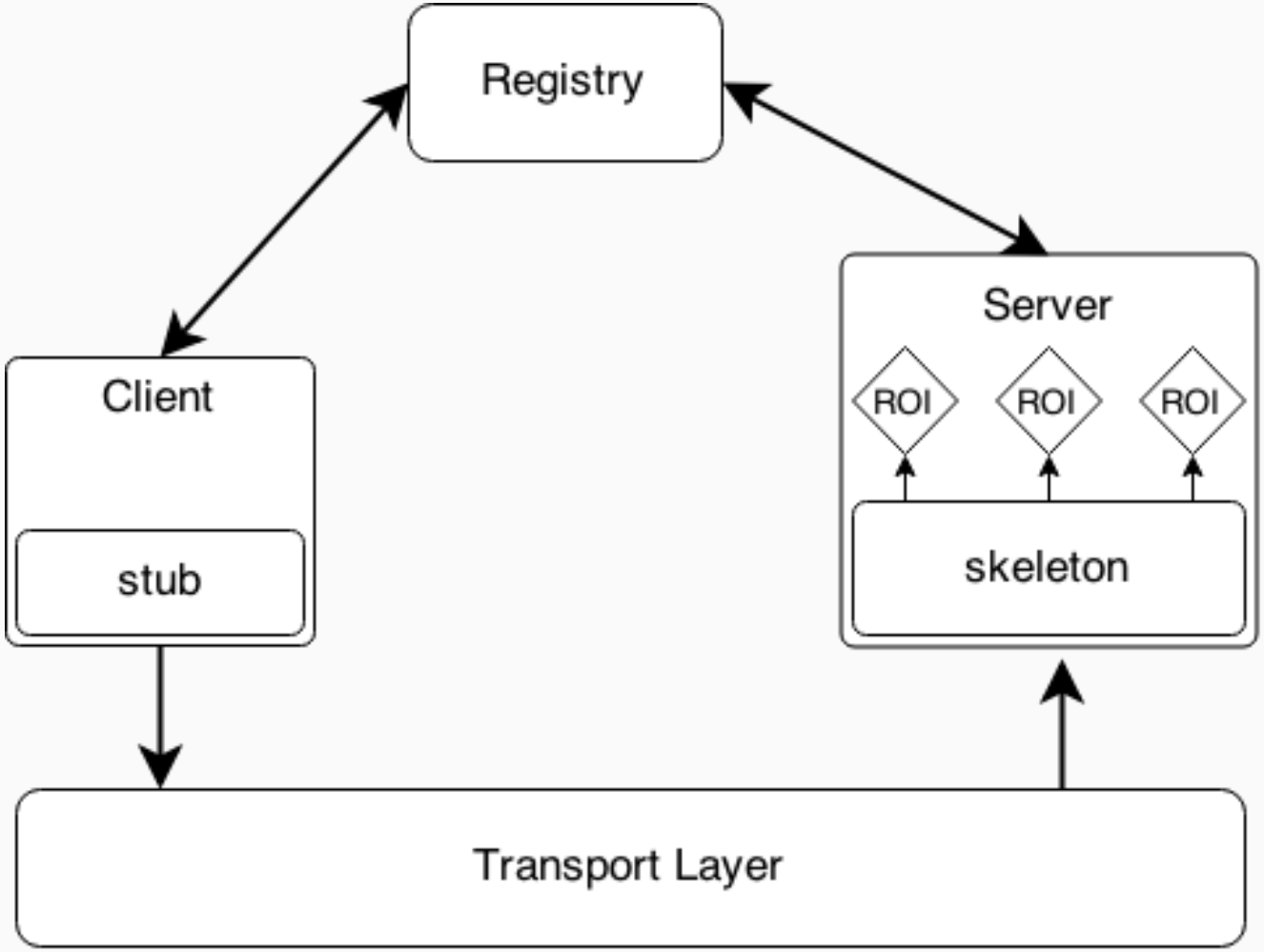
Università degli Studi di Milano

INTRODUZIONE A RMI

COSA È JAVA RMI

- Evoluzione Object Oriented del paradigma RPC sviluppata da Sun Microsystems
- Paradigma distribuito che permette ad un'applicazione Java di invocare metodi di oggetti attivi su una JVM differente.
- Adotta il modello client/server.

ARCHITETTURA DI UN'APPLICAZIONE RMI



ATTORI

Client	il processo che invoca i metodi remoti
Server	il processo che mette a disposizione un servizio o una risorsa
Registry	naming service (RMI registry) su cui saranno effettuare le operazioni di bind e lookup
Stub	rappresentazione locale dell'oggetto remoto, si connette allo skeleton e invia le richieste del client
Skeleton	riceve le richieste dello stub, le inoltra all'implementazione locale del metodo e restituisce i valori di ritorno

CICLO DI VITA DEL SERVER

1. Pubblicazione del servizio:
   - esportazione dell'oggetto remoto su una porta (1100)
   - creazione registro su un'altra porta (1099)
   - *bind(NomeServizio, RiferimentoOggetto)* su RMI registry
2. Attende le richieste del client
3. Delega l'elaborazione della richiesta ad un server thread
4. Torna nello stato di attesa (passo 2)

CICLO DI VITA DEL CLIENT

1. Si connette al server
2. *lookup(NomeServizio)* su RMI registry
3. Download stub
4. Invocazione metodo remoto
5. Attesa della risposta

RMI REGISTRY

- Naming service attraverso la quale il server registra i servizi che offre (bind)
- Il client lo interroga per sapere quali sono i servizi (lookup)
- Struttura dati in cui ogni entry è una coppia *(Nomeservizio, RiferimentoOggetto)*

VANTAGGI: LA TRASPARENZA

- L'invocazione remota avviene con la stessa sintassi delle invocazioni locali
- Nessun bisogno di un IDL specifico
- Interfacce remote definite con le espressioni native del linguaggio
- Possibilità di passare come argomento tipi di dato primitivi e oggetti locali

PASSAGGIO DEI PARAMETRI

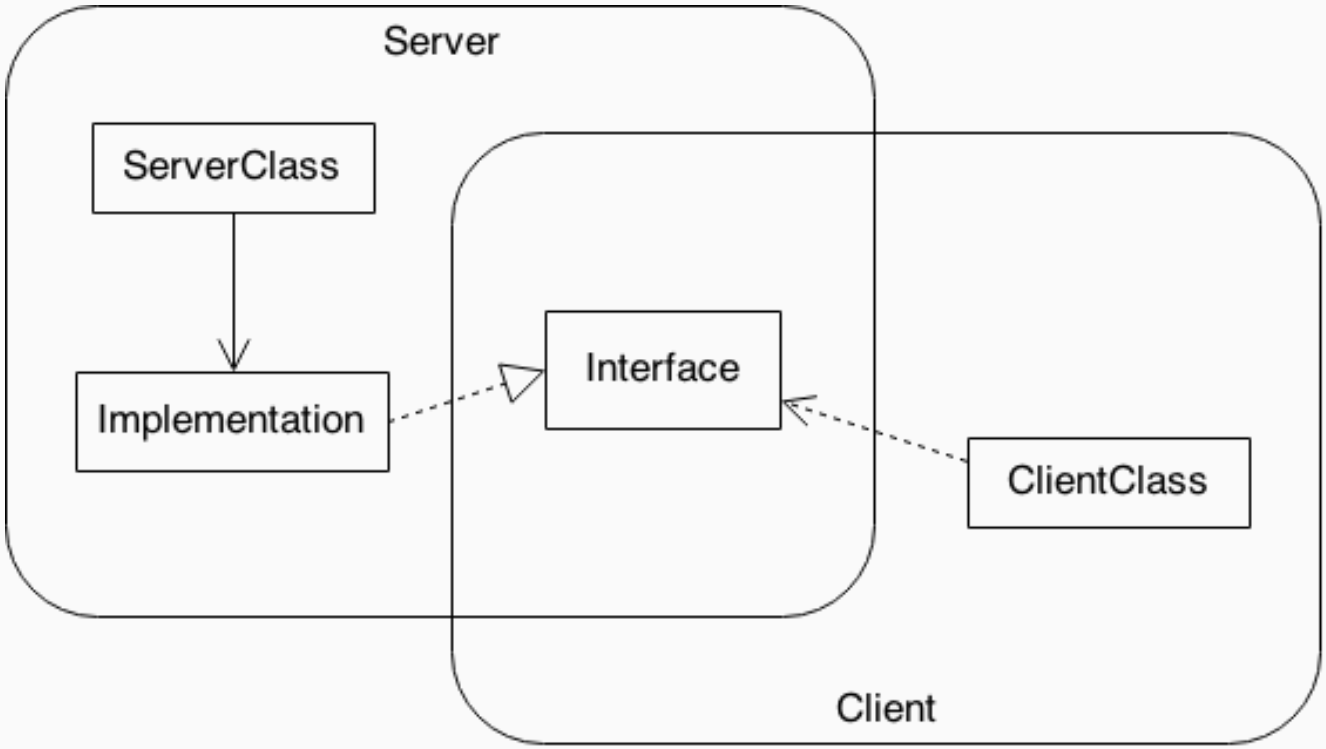
- Tipo di dato primitivo
  - *call-by-value*
  - marshalling e unmarshalling dei dati
- Oggetto locale
  - *call-by-reference* non possibile $\rightarrow$ simulazione
  - serializzazione degli oggetti in uscita
    - trasformazione dell'oggetto in una sequenza di byte
  - deserializzazione degli oggetti in entrata
    - ricostruzione dell'oggetto a partire dallo stream di byte

SEMANTICA DI RMI

- Le invocazioni remote sono *asincrone*
- Possibilità di fallimento
- Semantica *at-most-once*:
  - Il metodo non è stato eseguito il server invia un messaggio di errore $\rightarrow$ il client ripete la richiesta
  - altrimenti il metodo è assunto come eseguito (i metodi void non ritornano nessun valore)

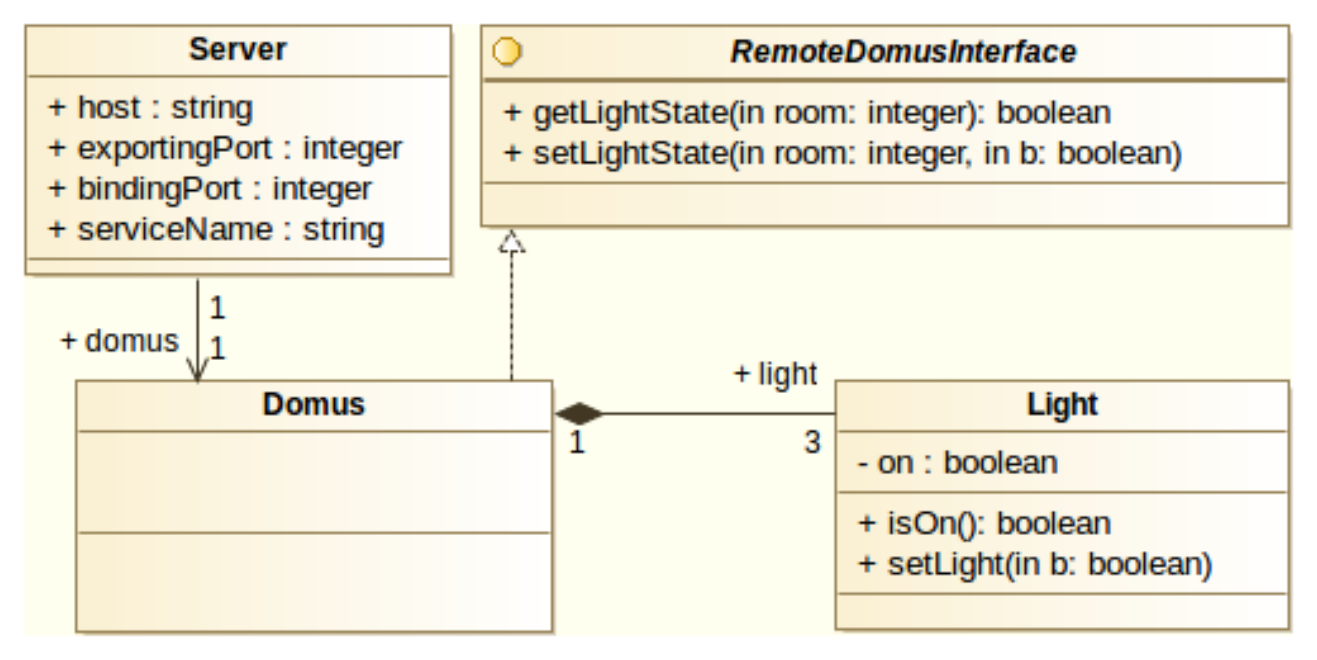
ESEMPIO DI IMPLEMENTAZIONE DI UN SERVIZIO

DISTRIBUZIONE DELLE CLASSI



PROGETTO

- L’oggetto remoto esposto rappresenta il sistema domotico di un’abitazione
- Per lo scopo di questa presentazione, si mostrerà l’implementazione di due semplici metodi per ottenere e modificare lo stato delle luci



INTERFACCIA REMOTA

```java
package domus.server;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RemoteDomusInterface extends Remote{

   public boolean getLightState(int room) throws RemoteException;
   public void setLightState(int room, boolean b) throws RemoteException;

}
```

METODI ESPOSTI DA DOMUS

```
7  public class Domus implements RemoteDomusInterface {
8     Light[] light;
9
10    public Domus() {
11       light = new Light[3];
12       light[0] = new Light("Kitchen");
13       light[1] = new Light("Living room");
14       light[2] = new Light("Bathroom");
15    }
16
17    @Override
18    public boolean getLightState(int room) throws RemoteException {
19       return light[room].isOn();
20    }
21
22    @Override
23    public void setLightState(int room, boolean b) throws RemoteException {
24       light[room].setLight(b);
25    }
26 }
```

IMPLEMENTAZIONE DI LIGHT

```java
package domus.server.sensors;

public class Light{
   private boolean on;

   public Light() {
      on = false;
   }

   public boolean isOn() {
      return on;
   }

   public void setLight(boolean b) {
      on = b;
   }
}
```

SERVER RMI

SERVER RMI (1)

Il principale compito del server RMI è quello di pubblicare il servizio sul naming service.
Per farlo dovrà predisporre la JVM ad accettare connessioni in entrata su una porta specifica.
Per questo scopo sono usate le classi

- Registry l'interfaccia degli oggetti remoti
- LocateRegistry fornisce i metodi per ottenere un riferimento all'oggetto remoto
- UnicastRemoteObject permette di esportare un oggetto remoto e di ottenere uno stub su cui effettuare le chiamate

SERVER RMI (2)

Per prima cosa, il server crea l'oggetto di cui esporrà i metodi e definisce alcuni attributi che serviranno alla pubblicazione del servizio.

```
10 public class Server {
11    Domus domus;
12    static String host ="<hostname>";
13    static int exportingPort = 1100;
14    static int bindingPort = 1099;
15    static String serviceName = "Domus";
16
17    public Server() {
18       super();
19       domus = new Domus();
20    }
```

SERVER RMI (3)

```
22    public static void main(String[] args) {
23       Server server = new Server();
24
25       System.setProperty("java.rmi.server.hostname", host);
26       try {
27          RemoteDomusInterface remoteObjectStub =
28              (RemoteDomusInterface) UnicastRemoteObject
29                .exportObject(server.domus, exportingPort);
30
31          Registry registry = LocateRegistry.createRegistry(bindingPort);
32
33          registry.rebind(serviceName, remoteObjectStub);
```

SERVER RMI (4)

- Assegnamento nome host con il metodo `setProperty()`
- Esportazione dell'oggetto con `exportObject()`
- Creazione del registro con `createRegistry()`
- Bind del servizio su RMI registry con il metodo `rebind()`

SERVER RMI (5)

```
36          Object alive=new Object();
37          synchronized (alive) {
38             try{
39                alive.wait();
40             } catch(InterruptedException e) {}
41          }
```

SERVER RMI (6)

- Creazione di un oggetto di classe Object che rimane in attesa e impedisce la terminazione del processo server da parte di JVM nei periodi di lunga inattività

CLIENT: RMI E JSP

CLIENT

Il client dovrà esporre un'interfaccia all'utente per visualizzare lo stato delle luci e permettergli di modificarlo. L'interfaccia sarà generata da una pagina JSP, mentre la parte relativa alla comunicazione con il server RMI sarà implementata dalla classe *Client.java*.

CLIENT RMI (1)

```
9  public class Client {
10    private RemoteDomusInterface rdi;
11
12    public Client() {
13       try {
14          String serviceName = "Domus";
15
16          Registry registry = LocateRegistry.getRegistry("<hostname>");
17
18          rdi = (RemoteDomusInterface) registry.lookup(serviceName);
19       } catch (RemoteException e) {
20          e.printStackTrace();
21       } catch (Exception e) {
22          e.printStackTrace();
23       }
24    }
```

CLIENT RMI (2)

- Utilizzo del metodo `getRegistry()` per ottenere un riferimento del registro
- Interrogazione del registro per ottenere un riferimento all'oggetto remoto con il metodo `lookup()`

CLIENT RMI (3)

```
26    public boolean getLightState(int room) {
27       try {
28          return rdi.getLightState(room);
29       } catch (RemoteException e) {
30          e.printStackTrace();
31       }
32       return false;
33    }
34
35    public void setLightState(int room, boolean b){
36       try {
37          rdi.setLightState(room, b);
38       } catch (RemoteException e) {
39          e.printStackTrace();
40       }
41    }
```

CLIENT RMI (4)

Il client utilizza i metodi remoti come se fossero locali attraverso il riferimento all'oggetto remoto *rdi*. L'unico accorgimento è intercettare eventuali *RemoteException* nel caso in cui si verifichino degli errori di rete (e.g. il server non è disponibile).

JAVA SERVER PAGES (1)

La pagina *index.jsp* si occupa di istanziare un oggetto di tipo *Client*, con cui potrà effettuare le invocazioni dei metodi remoti, e di presentare all'utente un'interfaccia grafica. Java Server Pages è una tecnologia che permette la creazione di pagine web dinamiche attraverso l'uso di specifici tag. Le pagine JSP dovranno essere eseguite all'interno di un application server (e.g. Apache Tomcat)

JAVA SERVER PAGES (2)

Creazione del Bean con cui si effettueranno le invocazioni a metodi remoti (*index.jsp*)

```
28  <body>
29   <jsp:useBean id="remote_object" scope="session"
30   class="domus.client.Client" />
```

JAVA SERVER PAGES (3)

Visualizzazione dello stato delle luci, chiamando il metodo remoto *getLightState()* per ogni stanza.

```
51 <%
52 if (remote_object.getLightState(0))
53    out.print("<a href=\"setLight.jsp?state=off&room=0\">
54              <span class=\"label label-success\">on</span>
55              </a>");
56 else
57    out.print("<a href=\"setLight.jsp?state=on&room=0\">
58              <span class=\"label label-danger\">off</span>
59              </a>");
60 %>
```

Lettura del form e invocazione del metodo remoto *setLightState()*

```
11  <jsp:useBean id="remote_object" scope="session"
12     class="domus.client.Client" />
13  <%
14     try {
15        String state = request.getParameter("state");
16        String tmpRoom = request.getParameter("room");
17        int room = Integer.parseInt(tmpRoom);
18
19        if(state.equals("on")) {
20           remote_object.setLightState(room, true);
21        } else {
22           remote_object.setLightState(room, false);
23        }
24
25     } catch (Exception e) { }
26     response.sendRedirect("index.jsp");
27  %>
```

DEPLOYMENT

Questo semplice esempio può essere riprodotto in locale eseguendo il main di *Server.java* e la pagina *index.jsp* (per cui sarà necessaria l'installazione di Apache Tomcat).

Domande?