Remote Method Invocation

30/11/2021

Federica Paganelli

Source: slides corso LPR A AA 2017/18

OLTRE I SOCKETS....

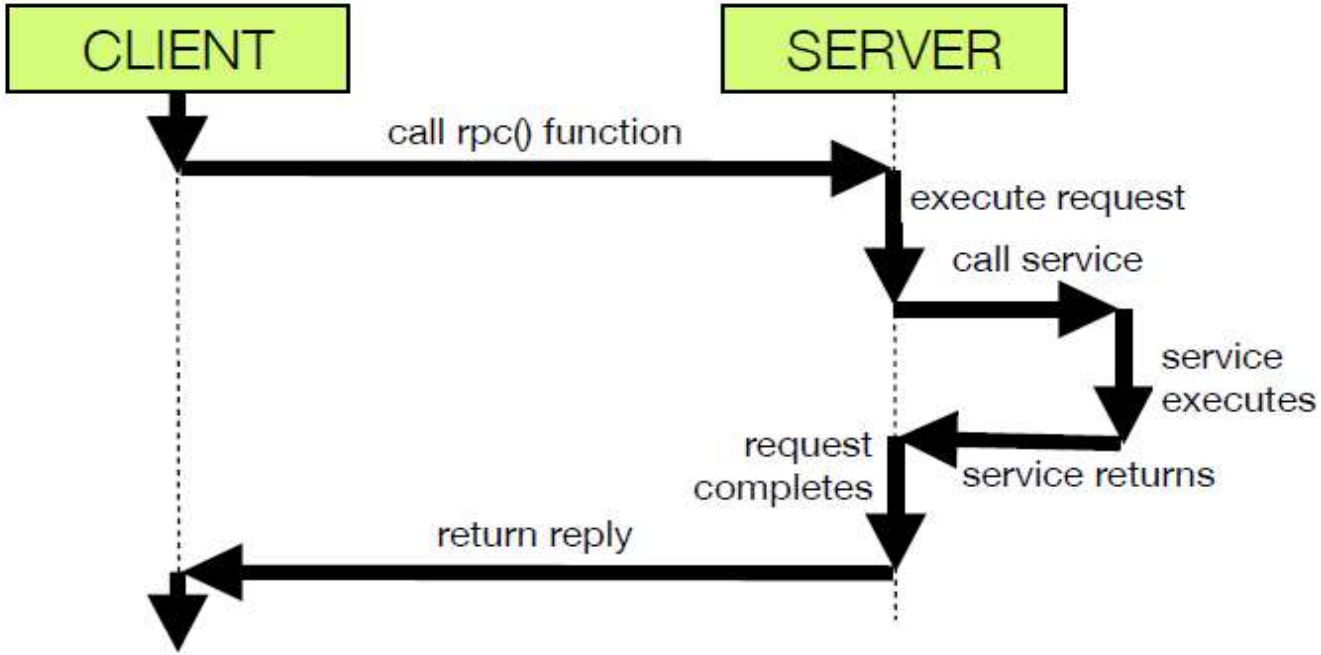
- un'applicazione JAVA distribuita:
  - è composta da computazioni eseguite su JVM differenti,
  - possibilmente in esecuzione su host differenti comunicanti tra loro (un'applicazione multithreaded non è distribuita!)

- Socket: un meccanismo flessibile e potente per la programmazione di applicazioni distribuite, ma di basso livello:
  - richiede la progettazione di veri e propri protocolli di comunicazione e la verifica non banale delle loro funzionalità
  - la serializzazione consente di ridurre complessità dei protocolli inviando dati strutturati

- Un'alternativa è utilizzare una tecnologia di più alto livello, originariamente indicata come **Remote Procedure Call (RPC)**
- interfaccia di comunicazione rappresentata dall'invocazione di una procedura remota, invece che dall'utilizzo diretto di una socket

REMOTE PROCEDURE CALL (RPC)

- paradigma di interazione a domanda/risposta
  - il client invoca una procedura del server remoto
  - il server esegue la procedura con i parametri passati dal client e restituisce a quest'ultimo il risultato dell'esecuzione.
- la connessione remota è trasparente rispetto a client e server
- in genere prevede meccanismi di affidabilità a livello sottostante



Ad esempio: richiesta di un servizio di stampa e restituzione esito operazione

RPC: UN ESEMPIO

Paradigma di interazione domanda/risposta:

- un client richiede ad un server la stampa di un messaggio e attende l'esito dell'operazione
- Il server restituisce un codice che indica l'esito della operazione

I meccanismi utilizzati dal client sono gli stessi utilizzati per una invocazione di procedura locale, ma …

- l'invocazione di procedura avviene sull' host su cui è in esecuzione il client
- la procedura viene eseguita sull' host su cui è in esecuzione il server
- i parametri della procedura vengono inviati automaticamente sulla rete dal supporto

Il programmatore non deve più preoccuparsi di sviluppare protocolli che si occupino del trasferimento dei dati, della verifica, e della codifica/decodifica.

- operazioni interamente gestite dal supporto
- utilizzo di **stub** o **proxy** (alla lettera moncone, mozzicone) presenti sul client

REMOTE METHOD INVOCATION

A partire dagli inizi degli anni ’90 sono state proposte diverse tecnologie, dopo SUN RPC:

- **CORBA**: sviluppato con l'obiettivo di supportare applicazioni scritte in linguaggi diversi su piattaforme differenti.
  - obiettivo: inter-operabilità
  - l'”object model” di riferimento deve essere “language neutral”
- **DCOM (Distributed component Object Module)**: supporta applicazioni scritte in linguaggi differenti, ma su piattaforme Win32. Esistono delle implementazioni per sistemi Unix.
- **.NET remoting:** supporta applicazioni scritte in linguaggi differenti, su piattaforma Windows.
- **Web Service/SOAP:** un framework per oggetti remoti basato su http e XML
- **Java RMI**: supporta applicazioni JAVA distribuite, ovvero le applicazioni possono essere distribuite su differenti JVM: ambiente omogeneo. Supporta le specifiche IIOP (Internet InterORB Protocol) per l’integrazione tra JAVA RMI ed altri framework basati sul paradigma ad oggetti

REMOTE METHOD INVOCATION IN BREVE

- un'applicazione RMI (Remote Method Invocation) è, in generale, composta da due programmi separati: un server ed un client
- il server
  - crea un oggetto remoto e pubblica un riferimento all'oggetto
  - attende che i client invochino metodi sull'oggetto remoto
- il client
  - ottiene un riferimento all'oggetto remoto
  - invoca i suoi metodi
- obiettivo:
  - permettere al programmatore di sviluppare applicazioni JAVA distribuite utilizzando la stessa sintassi e semantica utilizzate per i programmi non-distribuiti
  - raggiunto in parte: trasparenza buona, ma non totale

REMOTE METHOD INVOCATION

Permette di usare oggetti “remoti” (cioè su altri nodi della rete Internet) come se fossero oggetti locali, senza doversi preoccupare della realizzazione dei protocolli, della connessione, degli stream di input e output, etc. etc.

- l’utilizzo di oggetti remoti risulta largamente trasparente: una volta localizzato l’oggetto, il programmatore utilizza i metodi dell’oggetto come se questo fosse locale
- codifica, decodifica, verifica, e trasmissione dei dati sono effettuati dal supporto RMI in maniera completamente trasparente all’utente

From: Java RMI, Zambonelli

REMOTE METHOD INVOCATION

Problemi da gestire:

- il cliente deve in qualche modo trovare un riferimento all'oggetto remoto (è diverso dal creare l'oggetto o dal farsi dare un riferimento passato come parametro)

- l'oggetto che vuole rendere i suoi servizi invocabili deve esplicitamente dire che vuole fare ciò e deve rendersi reperibile

From: Java RMI, Zambonelli

JAVA RMI: GENERALITA'

Oggetto Remoto è un oggetto i cui metodi possono essere acceduti da un diverso spazio di indirizzamento

- una JVM diversa
- che è potenzialmente in esecuzione su un altro host

- sfrutta le caratteristiche della programmazione ad oggetti
- tutte le funzionalità standard di JAVA sono disponibili in Java RMI meccanismi di sicurezza, serializzazione dei dati, JDBC, …
- si avvale di un Java Security Manager per controllare che le applicazioni distribuite abbiano i diritti necessari per essere eseguite

RMI: ARCHITETTURA AD ALTO LIVELLO

- L'architettura RMI prevede tre entità
  - **Server:** esporta gli oggetti remoti
  - **Client**: richiede i metodi degli oggetti remoti
  - **Registry:** è un servizio di naming che agisce da 'yellow pages‘:
    - registra i nomi e i riferimenti degli oggetti i cui metodi possono essere invocati da remoto
    - tali oggetti devono essere registrati (i.e. “**bind**” presso il naming service), con un nome pubblico
    - altri oggetti (client) possono richiedere (“**lookup**”) oggetti registrati chiedendo, a partire dal nome pubblico, un riferimento all’oggetto

RMI: ARCHITETTURA AD ALTO LIVELLO

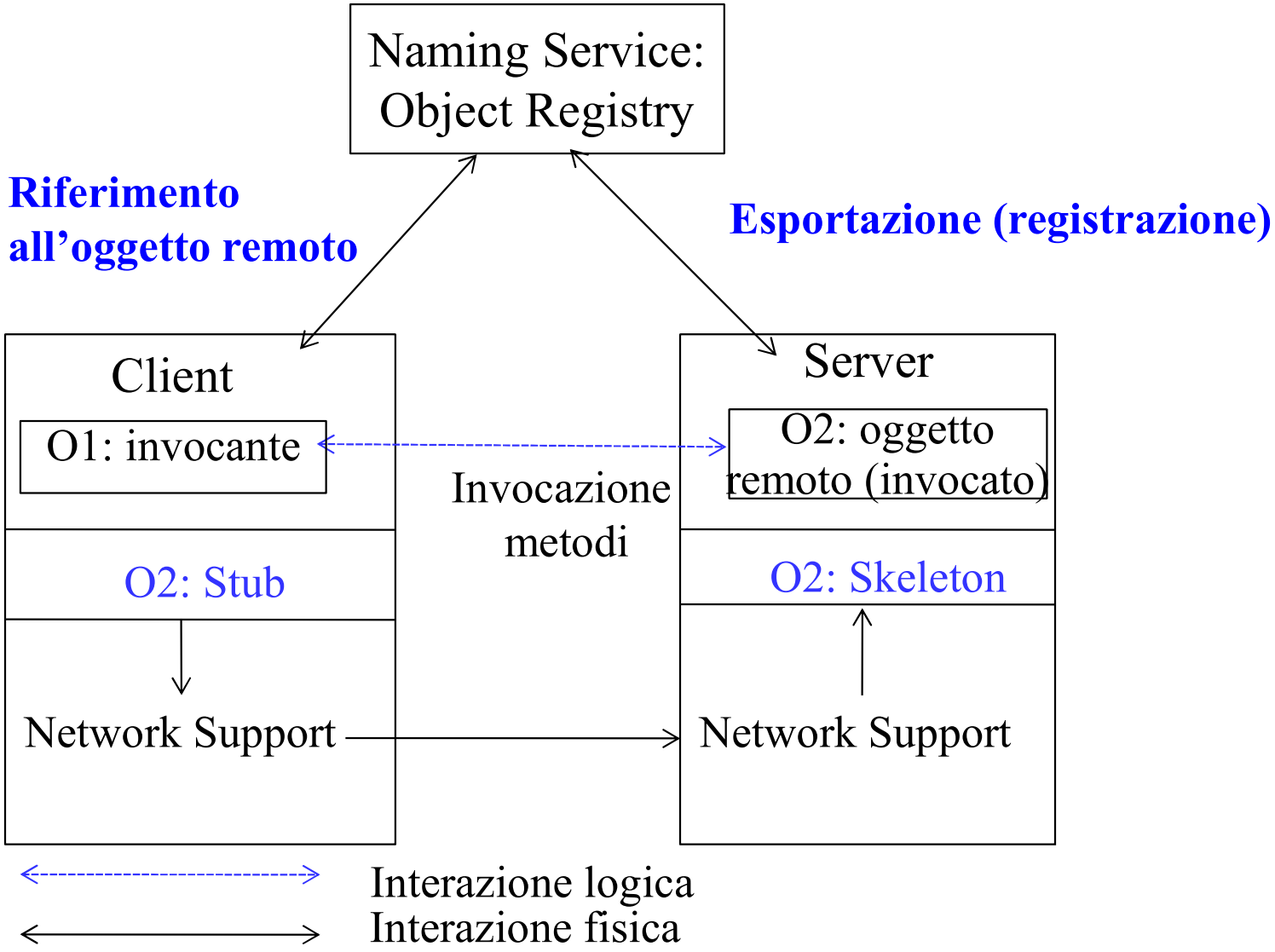
- principali operazioni da effettuare rispetto all'uso di un oggetto locale:
  1. il server deve esportare gli oggetti remoti, il client deve individuare un riferimento all'oggetto remoto.
  2. server registra gli oggetti remoti nel registry, tramite la bind
  3. I client cercano gli oggetti remoti, tramite la lookup, chiedendo, a partire dal nome pubblico dell'oggetto, un riferimento all'oggetto remoto al registry
  4. Il client invoca il servizio mediante chiamate di metodi che sono le stesse delle invocazioni di metodi locali

```
OggettoOvunque cc;
cc.metodo()
```

RMI: ARCHITETTURA AD ALTO LIVELLO

- invocazione dei metodi di un oggetto remoto:
  - a livello logico: identica all'invocazione di un metodo locale
  - a livello di supporto: gestita dal supporto *RMI* che provvede a trasformare i parametri della chiamata remota in dati da spedire sulla rete. Il network support provvede quindi all'invio vero e proprio dei dati sulla rete

RMI: SCHEMA ARCHITETTURALE



Stub: proxy locale su cui vengono fatte le invocazioni destinate all'oggetto remoto
Skeleton: entità che riceve le invocazioni fatte sullo stub e le realizza effettuando le corrispondenti chiamate sul server

RMI: SCHEMA ARCHITETTURALE

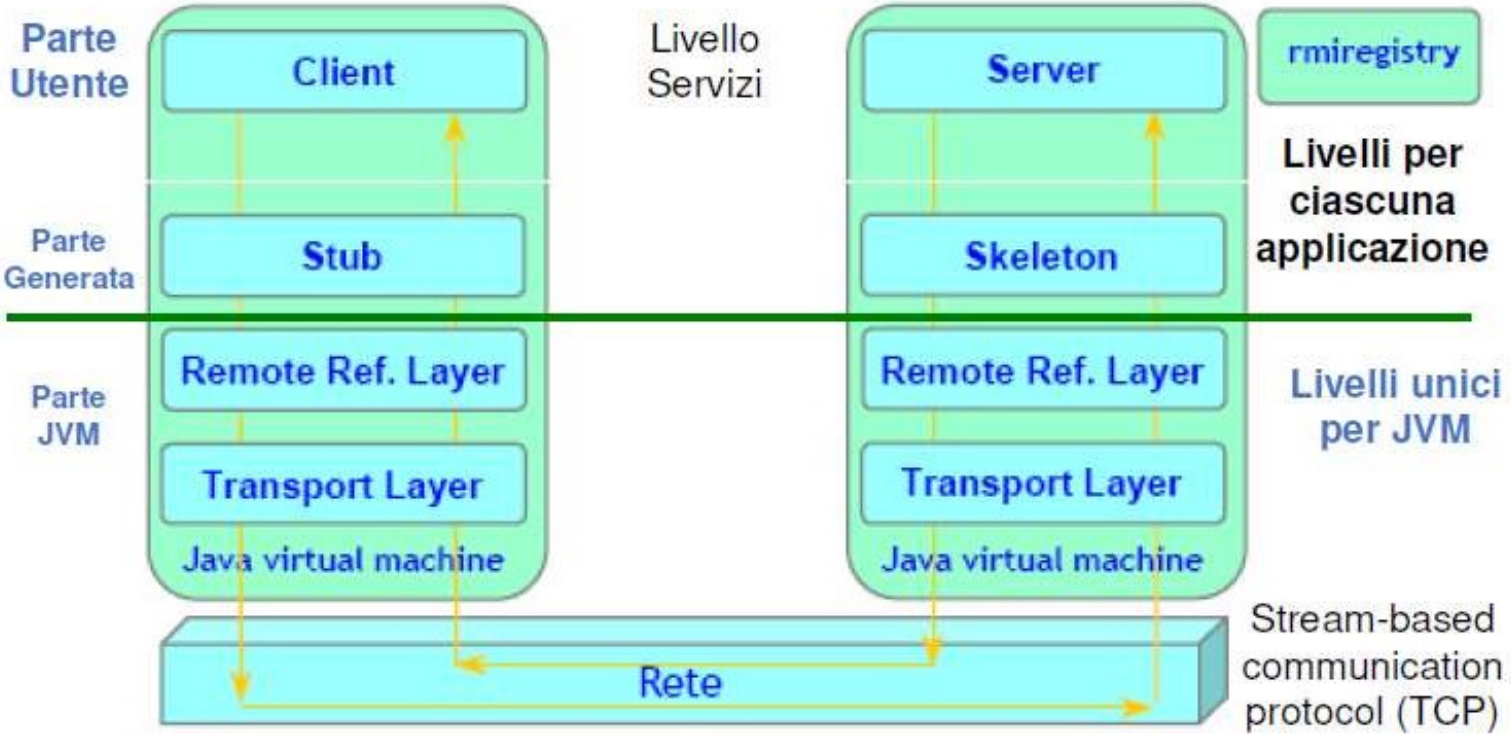
quando O1 invoca un servizio di O2, lo chiede al suo proxy (O2 Stub, un rappresentante locale di O2) che:

- In modo trasparente a O1, stabilisce una connessione Socket TCP con il nodo dove c'è O2 *vero*

- Sul nodo di O2, un componente associato a O2, detto "skeleton" riceve la connessione e la richiesta di servizio, e provvede ad invocarla su O2 *vero (lo skeleton* è un rappresentante locale del client)

- quando O2 vero risponde, lo skeleton manda la risposta indietro al proxy, il quale poi risponde a O1 come se fosse stato realmente lui ad eseguire il servizio

RMI: SCHEMA ARCHITETTURALE



Il livello Remote Reference Layer (RRL):

• Responsabile della gestione dei riferimenti agli oggetti remoti, dei parametri e delle astrazioni di stream-oriented connection

From: Corradi, Foschini, RMI, 2019/20

RMI: SCHEMA ARCHITETTURALE

Alcune considerazioni pratiche:

- **Separazione** tra
  - Definizione del comportamento -> **interfacce**
  - Implementazione del comportamento -> **classe**
- I **componenti remoti** sono riferiti tramite **variabili interfaccia**
- Alcuni vincoli sui componenti
  1. **Definizione** del comportamento, con
     - **interfaccia** deve estendere **java.rmi.Remote**
     - **ogni metodo** deve propagare **java.rmi.RemoteException**
  2. **Implementazione** comportamento, classe che
     - deve **implementare** l'interfaccia definita
     - deve **estendere java.rmi.UnicastRemoteObject**

RMI STEP BY STEP

Illustreremo RMI attraverso un esempio di applicazione

Definire un server che implementi, mediante un oggetto remoto, il seguente servizio:

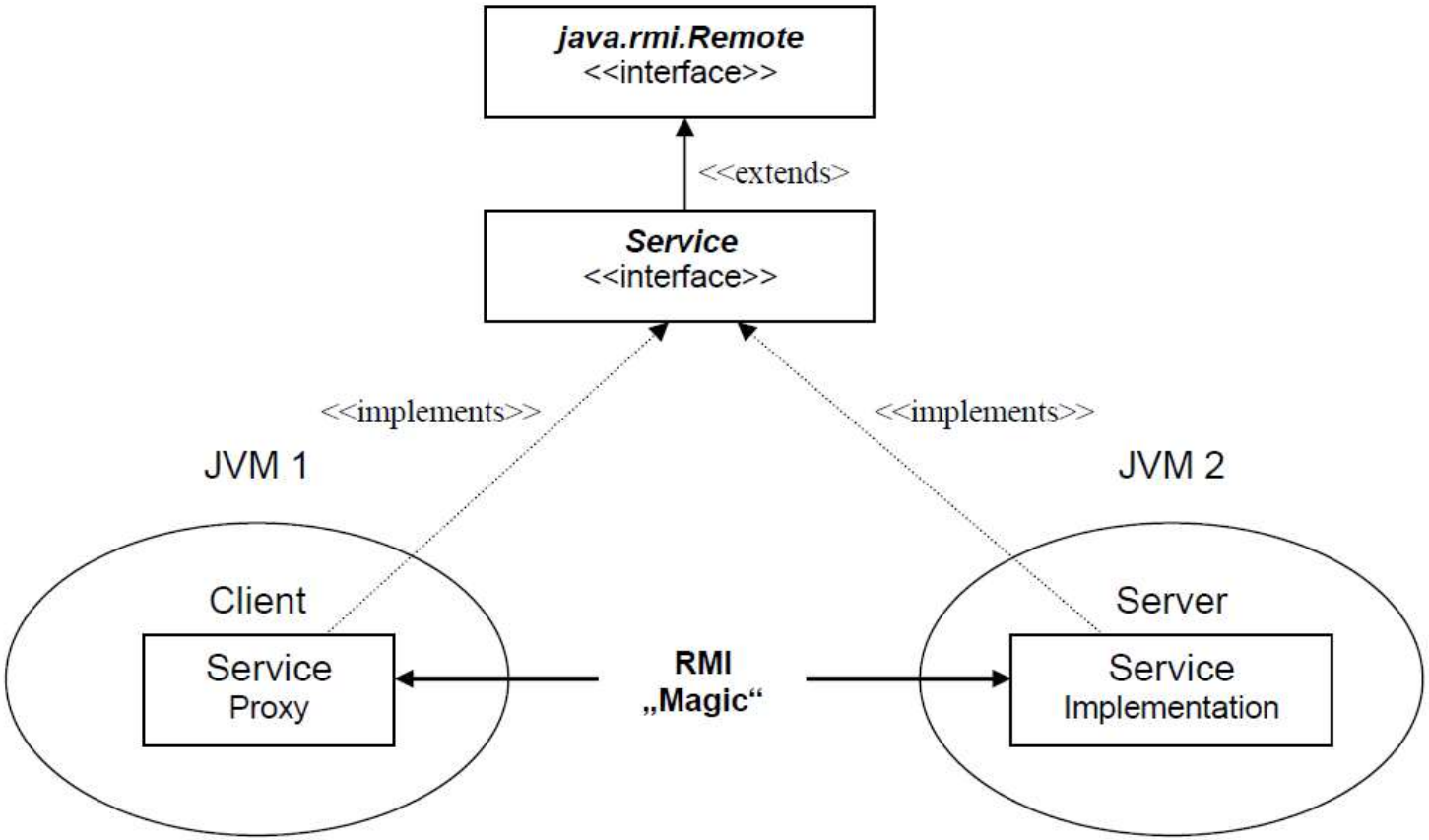
su richiesta del client, il server restituisce le principali informazioni relative ad un paese dell'Unione Europea di cui il client ha specificato il nome

- *linguaggio ufficiale*
- *popolazione*
- *nome della capitale*

RMI : INTERFACCIA REMOTE

`Remote`: interfaccia JAVA che dichiara i metodi accessibili da remoto:

- è implementata da due classi:
  - sul server, la classe che implementa il servizio
  - sul client, la classe che implementa il proxy del servizio remoto



RMI : L'INTERFACCIA

- un'interfaccia è remota se e solo se estende **java.rmi.Remote** o un'altra interfaccia che estenda **java.rmi.Remote**

- **Remote** *(tag interface)* non definisce alcun metodo, il solo scopo è solo quello di identificare gli oggetti che possono essere utilizzati in remoto
- i metodi definiti devono dichiarare di sollevare eccezioni remote, della classe java.rmi.RemoteException, in aggiunta a eccezioni specifiche della applicazione

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface IntRemota extends Remote {
      public int remoteHash (String s) throws RemoteException {
      …
      }
```

RMI : L'INTERFACCIA

- I metodi remoti devono dichiarare di sollevare l'eccezione `Java.rmi.RemoteException` – ma il servente non deve sollevare eccezioni di questo tipo.

The exception java.rmi.RemoteException is thrown when a remote method invocation fails for some reason.

Some reasons for remote method invocation failure include:

- *communication failure*
- *failure during parameter or return value marshalling or unmarshalling*
- *protocol errors*

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface IntRemota extends Remote {
     public int remoteHash (String s) throws RemoteException;}
```

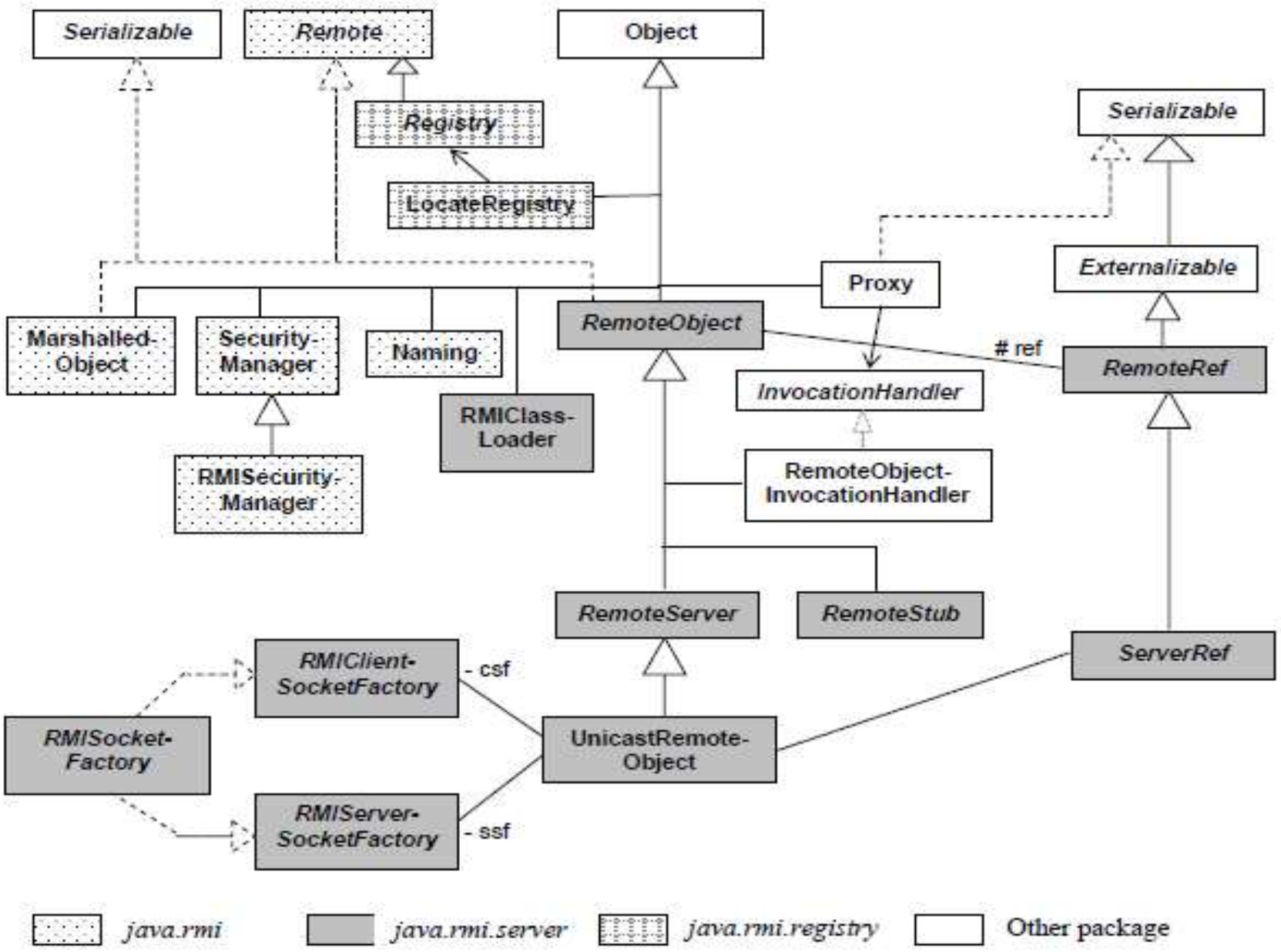
STEP 1: DEFINIZIONE DELL'INTERFACCIA REMOTA

```
import javarmi.Remote;
import java.rmi.RemoteException;

public interface EUStats extends Remote {
        String getMainLanguages(String CountryName)
                                                throws RemoteException;

        int getPopulation(String CountryName)
                                                throws RemoteException

        String getCapitalName(String CountryName)
                                                throws RemoteException;
                                                }
```

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO REMOTO

- secondo passo: implementazione del servizio remoto
- l'implementazione dei metodi può essere effettuata come per un metodo qualsiasi, ma....
  - tenere presente che l'oggetto remoto (`RemoteObject`) ha alcune caratteristiche diverse dagli altri oggetti
    - le operazioni base sugli oggetti remoti hanno una semantica diversa da quella degli oggetti standard, che sono istanze della classe Object
    - l'oggetto deve essere esportato
- per la definizione di oggetti remoti:
  - Package `java.rmi.server` supporta la definizione degli oggetti remoti

RMI: PACKAGES E CLASSI



STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

- classi del package `java.rmi.server`

  `UnicastRemoteObject < RemoteServer < RemoteObject < Object`

  dove A < B significa che A è una sottoclasse di B.

- la semantica di default di alcuni metodi "base" della classe **Object** non risulta appropriata per oggetti remoti.
  - metodi da riconsiderare: **`hash`**, **`equal`**, **`toString....`**

- **`RemoteObject`** fornisce implementazioni per i metodi hashCode, equals, and toString (implementandone il comportamento per un oggetto remote)
- **`RemoteServer`**: super classe astratta comune per tutte le implementazioni di oggetti remoti (`UnicastRemoteObject` è una implementazione)
- **`UnicastRemoteObject`**: definisce un oggetto remoto i cui riferimenti sono validi solo mentre il processo server è attivo. Fornisce supporto per riferimenti ad oggetti (chiamate, parametri e risultati) utilizzando flussi TCP

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

- L'implementazione della interfaccia può quindi fare uso delle classi descritte nelle slide precedenti, secondo le seguenti tre soluzioni:

Soluzione 1

- definire una classe che implementi i metodi della interfaccia remota ed estenda la classe `RemoteServer`

```
public class MyServerRemoto extends RemoteServer implements IntRemota
{
    public MyServerRemoto() throws RemoteException {  ..... }
        ......}
```

Richiede esportazione esplicita dell'oggetto remoto

- Vantaggi: si eredita la ridefinizione della semantica degli oggetti remoti definita da `RemoteServer`
- Svantaggi: a causa dell'eredità singola, non si possono estendere altre classi

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

Soluzione 2

- definire una classe che implementi i metodi della interfaccia remota ed estenda la classe `UnicastRemoteObject`
- **`UnicastRemoteObject`:**
  - estende `RemoteServer`, che estende, a sua volta, `RemoteObject`
  - **contiene metodi per costruire ed esportare un oggetto remoto**.

```
public class MyServerRemoto extends UnicastRemoteObject
                                    implements IntRemota {
  public MyServerRemoto() throws RemoteException {
        super(); // E' qui che il server viene esportato
            .....}
```

- simile alla soluzione precedente, stessi vantaggi e svantaggi
- differenza nell'esportazione dell'oggetto: il costruttore di `UnicastRemoteObject` crea automaticamente una server socket per ricevere le invocazioni di metodi remoti ed inizia ad attendere invocazioni su di esso

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

Soluzione 3

- definire una classe che implementi i metodi della interfaccia remota senza estendere alcuna delle classi viste.

```
public class MyServerRemoto extends MyClass implements
            IntRemota {
    public MyServerRemoto() throws RemoteException {.....}
        .....
     }
```

Richiede esportazione esplicita

- vantaggi: possibilità di estendere un'altra classe utile per l'applicazione
- svantaggi: semantica degli oggetti remoti demandata al programmatore (overriding metodi `equals`, `hashcode`,…)

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

- applichiamo i concetti visti alla implementazione del servizio `EUStatServer`
- utilizziamo una HashTable per memorizzare i dati riguardanti le nazioni europee. In questo modo definiamo un semplice data base delle nazioni
  - utilizzata la classe **`EUData`** per definire oggetti (record) che descrivono la singola nazione
- definiamo una classe **`EuStatServer`** che, utilizzando la classe **`EUData`**, implementa l'oggetto remoto e ne crea un'istanza
  - implementa l'interfaccia **`EUStats`**
  - il main della classe contiene il "launch code" dell'oggetto remoto
    1. crea una istanza dell'oggetto remoto
    2. esporta l'oggetto remoto
    3. associa all'oggetto remoto un nome simbolico e registra il collegamento nel registry

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

Una classe di appoggio:

```
class EUData {
        private String Language;
        private int population;
        private String Capital;
        EUData(String Lang, int pop, String Cap) {
                        Language = Lang;
                        population = pop;
                        Capital = Cap;
                        }
    String getLangs() { return Language; }
    int getPop()      { return population; }
    String getCapital() { return Capital; }
}
```

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

```
import java.rmi.*;                  // Classes and support for RMI
import java.rmi.server.*;           // Classes and support for RMI servers
import java.util.Hashtable;  // Contains Hashtable class
public class EUStatsServiceImpl extends RemoteServer
                                    implements EUStats {
/* Store data in a hashtable */
Hashtable<String, EUData> EUDbase = new  Hashtable<String, EUData>();
/* Constructor - set up database */
    EUStatsServiceImpl() throws RemoteException {
       EUDbase.put("France", new EUData("French",57800000,"Paris"));
       EUDbase.put("United Kingdom", new EUData("English",
                                                  57998000,"London"));
       EUDbase.put("Greece", new EUData("Greek",10270000,"Athens"));
   ........  }
```

STEP 2: IMPLEMENTAZIONE DEL SERVIZIO

```
/* implementazione dei metodi dell'interfaccia */
public String getMainLanguages(String CountryName)
                                        throws RemoteException {
        EUData Data = (EUData) EUDbase.get(CountryName);
        return Data.getLangs();}

public int getPopulation(String CountryName)
                                       throws RemoteException {
        EUData Data = (EUData) EUDbase.get(CountryName);
        return Data.getPop();    }

public String getCapitalName(String CountryName)
                                              throws RemoteException {
        EUData Data = (EUData) EUDbase.get(CountryName);
        return Data.getCapital();
}
```

PASSO 3: ATTIVAZIONE SERVIZIO

- Il servizio viene creato allocando una istanza dell'oggetto remoto
- Il servizio viene attivato mediante:
  - creazione dell'oggetto remoto
  - registrazione dell'oggetto remoto in un registry

PASSO 3:ATTIVAZIONE DEL SERVIZIO

```
*** classe EustatServer ****

public static void main (String args[]) {
try {
        /* Creazione di un'istanza dell'oggetto EUStatsServiceImpl */
    EUStatsServiceImpl statsService = new EUStatsServiceImpl();

        /* Esportazione dell'Oggetto */
    EuStats stub = (EuStats)
        UnicastRemoteObject.exportObject(statsService, 0);

   /* Creazione di un registry sulla porta args[0]
    LocateRegistry.createRegistry(args[0]);

    Registry r=LocateRegistry.getRegistry(args[0]);

   /* Pubblicazione dello stub nel registry */
    r.rebind("EUSTATS-SERVER", stub);

System.out.println("Server ready");}

/* If any communication failure occurs... */
catch (RemoteException e) {
System.out.println("Communication error " + e.toString());}}}
```

PASSO 3: ATTIVAZIONE del SERVIZIO

Il main della classe EustatServer:

- crea un'istanza del servizio (oggetto remoto)
- invoca il metodo statico **UnicastRemoteObject.exportObject(obj, 0)** che esporta dinamicamente l'oggetto, se si indica la porta 0 viene scelta una porta anonima
- Restituisce il riferimento allo stub dell'oggetto remoto, che “rappresenta” il riferimento all'oggetto remoto

- pubblica il riferimento all'oggetto remoto nel registry

PASSO 3: ATTIVAZIONE del SERVIZIO

metodo statico **`UnicastRemoteObject.exportObject(obj, 0)`**

The static method `UnicastRemoteObject.exportObject` exports the supplied remote object to receive incoming remote method invocations on an anonymous TCP port and returns the stub for the remote object to pass to clients.

The returned stub implements **the same set of remote interfaces as the remote object's class and contains the host name and port over which the remote object can be contacted**.

NB So, the client can call the same methods in the stub that it wants to call in this server. But, the functions in the stub are filled with network-related code, not the actual implementation of the required function. For example, if a server implements an add(int,int) function, the stub also will have an add(int,int) function, but it won't contain the actual implementation of the addition function; instead**, it will contain the code to connect to the remote skeleton, to send details about the function to be invoked, to send the parameters, and to get the results back**.

https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/hello/hello-world.html
https://www.developer.com/java/other/article.php/3455311/Understanding-Java-RMI-Internals.htm

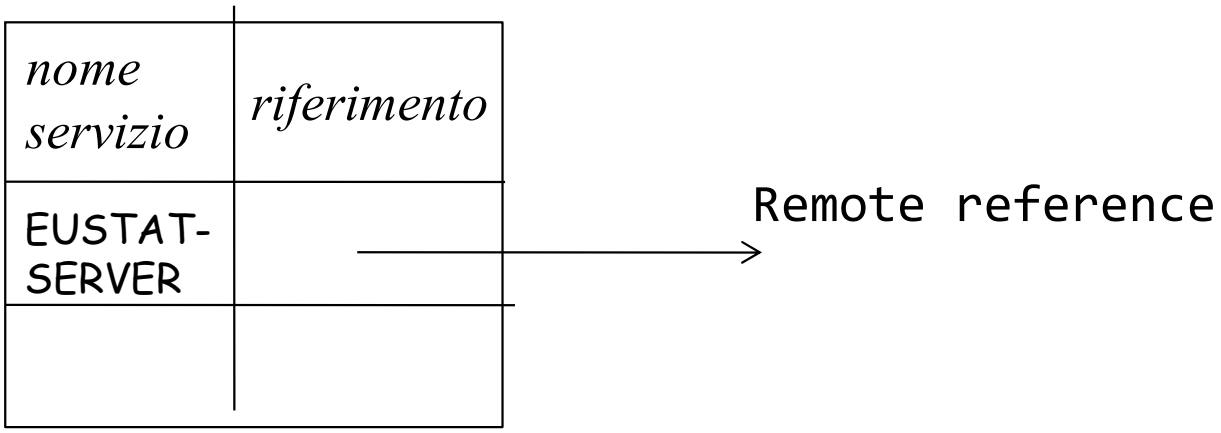
RMI: IL REGISTRY

JAVA mette a disposizione del programmatore un semplice name server (registry) che consente la registrazione ed il reperimento dello Stub

Registry:

- server remoto in ascolto sulla porta indicata.
- simile ad un DNS per oggetti remoti, contiene legami tra il nome simbolico dell'oggetto remoto ed il riferimento all'oggetto
- Possiamo supporre Registry in esecuzione su localhost:

```
LocateRegistry.createRegistry(args[0]);
Registry r=LocateRegistry.getRegistry(args[0]);
```



RMI: IL REGISTRY

- per creare e gestire direttamente da programma oggetti di tipo Registry, utilizzare la classe **LocateRegistry,** alcuni dei metodi statici implementati

```
public static Registry createRegistry(int port)
public static Registry getRegistry(String host, int port)
```

- `createRegistry`: lanciare un registro RMI sull'host locale, su una porta specificata e restituisce un riferimento al registro

- `getRegistry` restituisce un riferimento ad un registro RMI su un certo host ad una certa porta.
  - ottenuto il riferimento al registro RMI (è uno stub anche questo), si possono invocare i metodi definiti dall'interfaccia Registry. Solo allora viene creata una connessione col registro.
  - restituita una eccezione se non esiste il registro RMI corrispondente

RMI: IL REGISTRY

Supponiamo che r sia l'istanza di un registro individuato mediante `getRegistry( )`

- **`r.bind()`**
  - crea un collegamento tra un nome simbolico ed il riferimento ad un oggetto remoto.
  - se esiste già un collegamento per lo stesso oggetto all'interno di r, viene sollevata una eccezione

- **`r.rebind()`**
  - crea un collegamento tra un nome simbolico (qualsiasi) ed un riferimento all'oggetto.
  - se esiste già un collegamento per lo stesso oggetto all'interno dello stesso registry, tale collegamento viene sovrascritto
- è possibile inserire più istanze dello stesso oggetto remoto nel registry, con nomi simbolici diversi

RMI REGISTRY

- **`list() e lookup(String name)`** sono utilizzati dal client per interrogare il registro RMI.
  - entrambi i metodi, dopo aver fatto un parsing del parametro, aprono una connessione socket verso il registro RMI.

- **`list()`**
  - restituisce un array di stringhe dei nomi dei server remoti presenti nel registro RMI.

- **`lookup(String name)`**
  - passa al registro RMI un nome di servizio remoto
  - riceve una copia serializzata dello stub associato a quel server remoto
  - può restituire un'eccezione `NotBoundException` se il server non è presente nel registro + altre eccezioni

PASSO 4: IL CLIENT RMI

Per accedere all'oggetto remoto, il client

- deve ricercare lo `Stub` dell'oggetto remoto
- accede al `Registry` attivato sul server mediante il nome simbolico dell'oggetto remoto.

- il riferimento restituito è di tipo generico (Remote): è necessario effettuarne il casting al tipo definito nell'interfaccia remota

- invoca i metodi dell'oggetto remoto come fossero metodi locali (l'unica differenza è che occorre intercettare **`RemoteException`**)

PASSO 4: IL CLIENT RMI

```
import java.rmi.*;
public class EUStatsClient {
public static void main (String args[]) {
        EUStats serverObject;
        Remote remoteObject;
        /* Check number of arguments */
        /* If not enough, print usage string and exit */
        if (args.length < 2) {
        System.out.println("usage: java EUStatsClient
                                                port countryname");
         return;
        }
```

IL CLIENT RMI

```
try {
   Registry r = LocateRegistry.getRegistry(args[0]);
   remoteObject = r.lookup("EUSTATS-SERVER");
   serverObject = (EUStats) remoteObject;
   System.out.println("Main language(s) of " + args[1] + "
        is/are " + serverObject.getMainLanguages(args[1]));
   System.out.println("Population of " + args[1] + " is "
        +  serverObject.getPopulation(args[1]));
   System.out.println("Capital of " + args[1] + " is "
                  + serverObject.getCapitalName(args[1]));}
                  catch (Exception e) {
        System.out.println("Error in invoking object method " +
                                   e.toString() + getMessage());
        e.printStackTrace();}}}
```

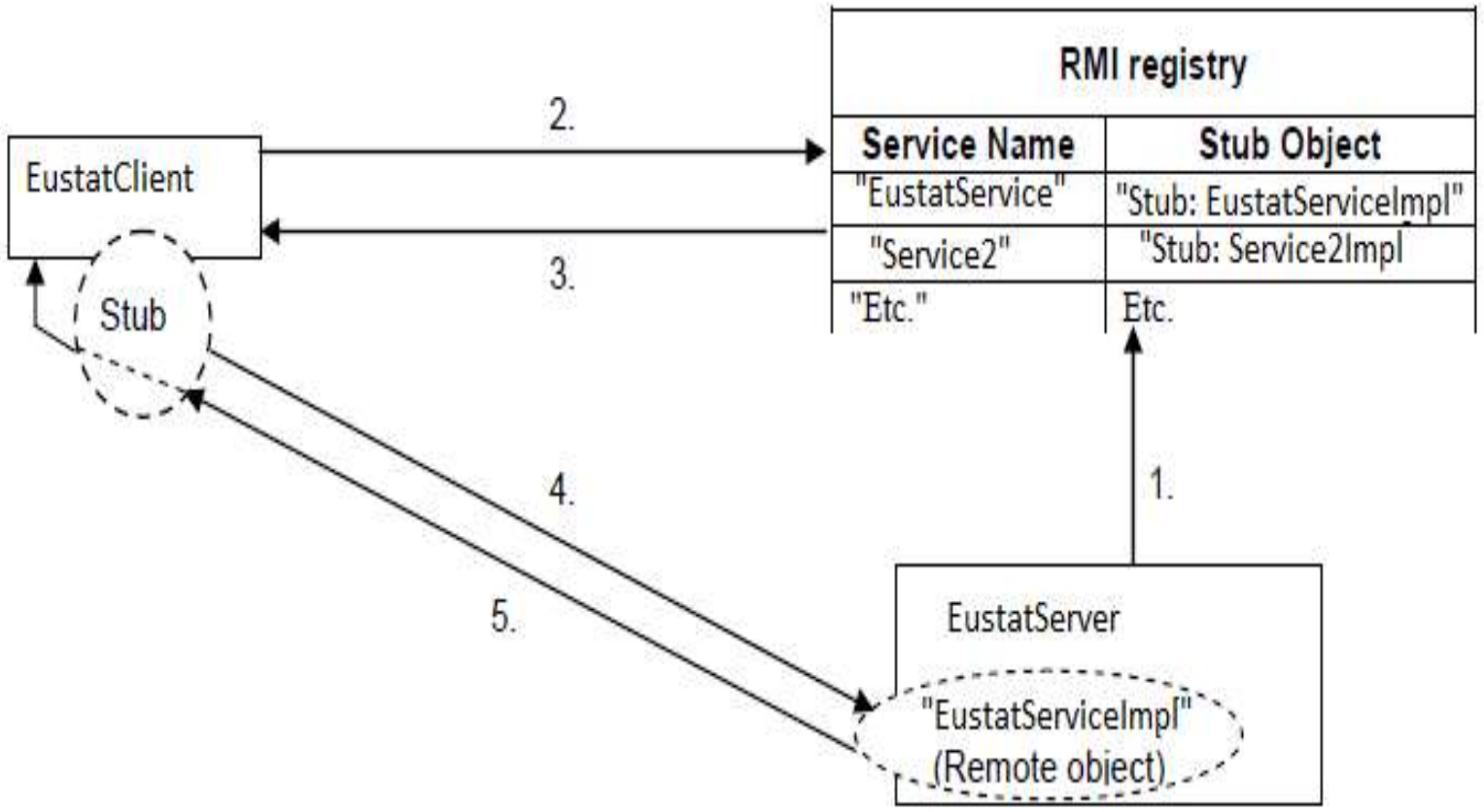
IL CLIENT RMI

- Gli argomenti passati al client da riga di comando sono il nome del paese europeo di cui si vogliono conoscere alcune informazioni e la porta su cui è in esecuzione il servizio di `Registry`.
- Remote indica un oggetto remoto su cui devo fare il casting al tipo definito dalla interfaccia `EUStats`
- NOTA BENE: l'invocazione dei metodi remoti avviene mediante lo stesso meccanismo utilizzato per l'invocazione dei metodi locali
- Compilazione ed invocazione del client

javac EUStats.java EUStatsClient.java
java EUStatsClient ip-host countryname

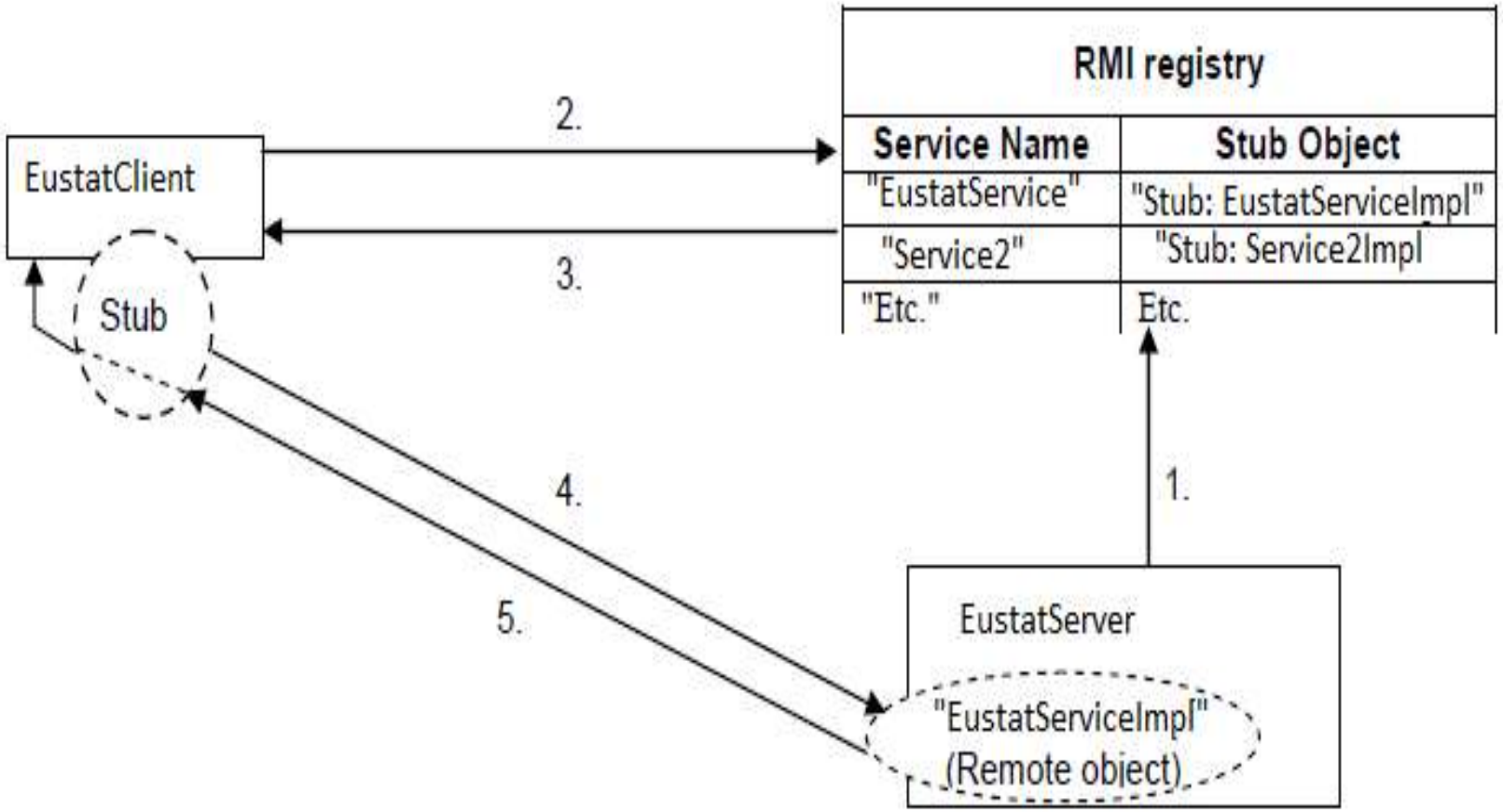
- NOTA BENE: quando viene compilato il codice del client, il compilatore JAVA ha bisogno di accedere al file .class dell'interfaccia `EUStats.class`

JAVA RMI > 1.5: THE EUSTAT APPLICATION



1. `EutStatServer` genera lo stub per l'oggetto remoto `EustatServiceImpl` e lo registra nel `Registry RMI` con il nome: "`EustatService`"
2. `EustatClient` effettua una look up nel `Registry` (`Naming.lookup(...)`)

JAVA RMI > 1.5: THE EUSTAT APPLICATION



3. il registry RMI registry restituisce lo stub al client
4. il client `EustatClient` effettua l'invocazione di metodo remoto mediante lo stub
5. viene restituito al client il servizio richiesto

"UNDER THE HOOD": LO STUB

Struttura di un oggetto `Stub`.

- stato: contiene essenzialmente tre informazioni:
  - l'IP dell'host su cui è in esecuzione il servizio
  - la porta di ascolto
  - un identificativo RMI associato all'oggetto. Viene serializzato quando viene esportato nel `Registry`

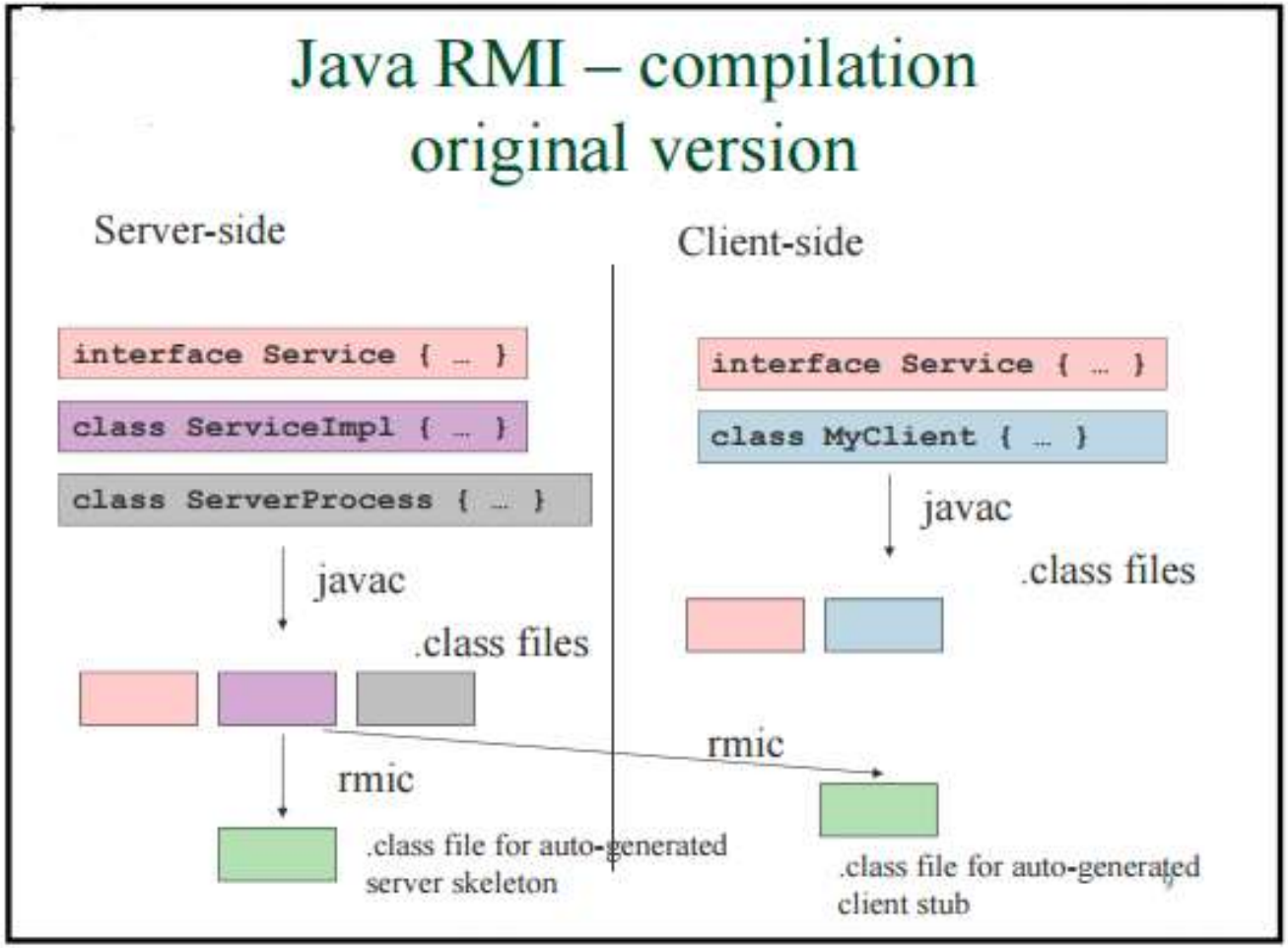
- metodi: la classe `Stub` implementa la medesima interfaccia remota del server remoto.
  - il codice dei metodi dello `Stub` non ha però niente a che vedere coi corrispondenti metodi del server
  - ogni metodo contiene il codice per inviare argomenti, invocare l'oggetto remoti e ricevere risultati
  - generato nella fase di esportazione, utilizzato dal client

PASSO 4: GENERAZIONE STUB

- `Stub`
  - è un oggetto che consente di interfacciarsi con un altro oggetto (il target) in modo da sostituirsi ad esso
  - target: oggetto remoto
  - si comporta da intermediario: inoltra le chiamate che riceve al suo target

- meccanismi diversi nelle diverse versioni di JAVA
  - **`RMI compiler`**, nelle prime versioni <1.5
  - **`Reflection`**, nelle versioni più recenti: noi utilizzeremo questo meccanismo

GENERAZIONE STATICA DI STUB



1) Compilazione
- sull'host del server

```
javac EUstats.java EustatServer.java
```

- analogo per il client

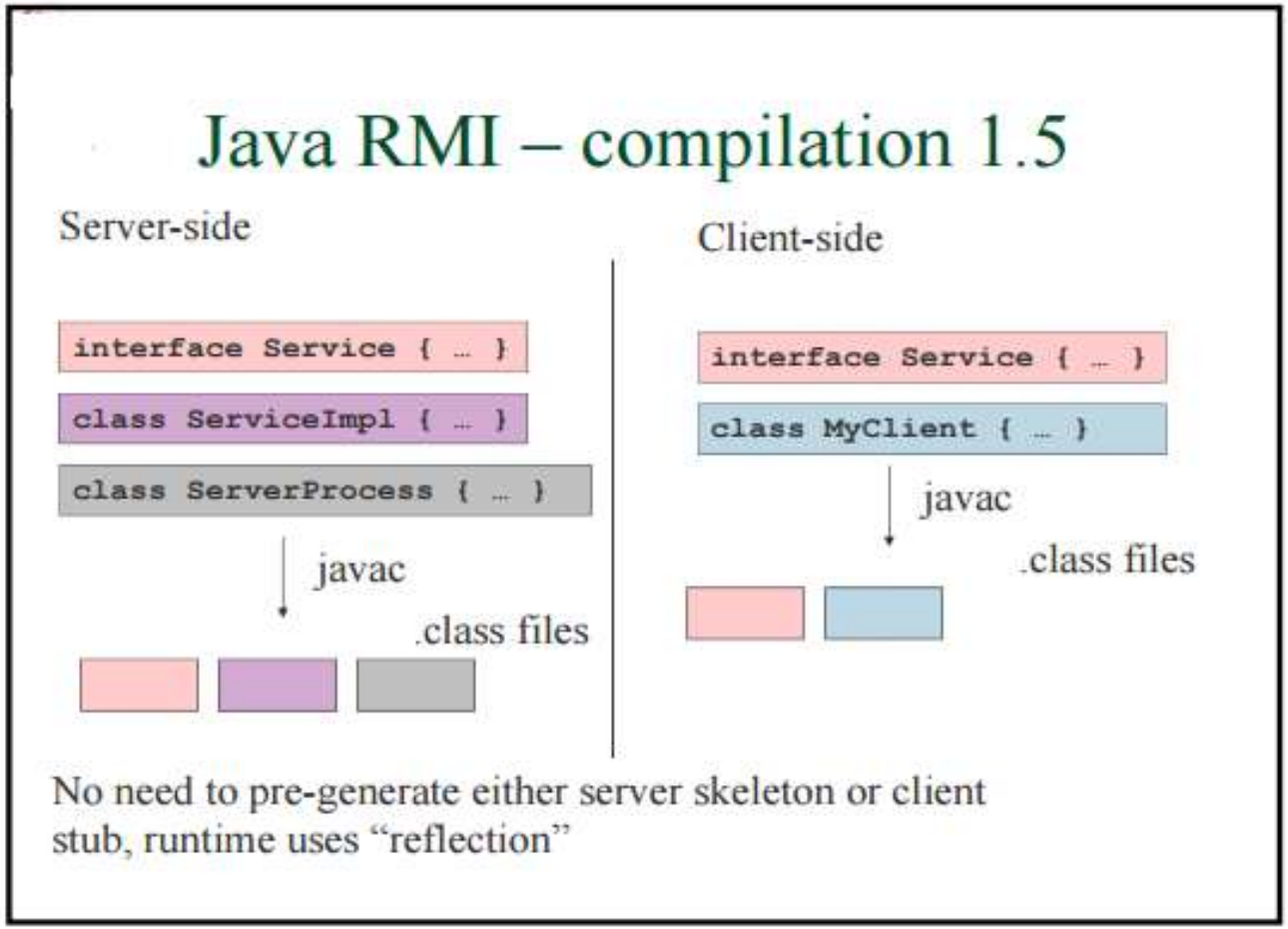
2) nelle versioni di JAVA antecedenti alla 5.0 stub e skeleton generato staticamente mediante un rmi compiler

```
rmic EustatServer
```

da utilizzare per client che utilizzano versioni precedenti di JAVA.

- Crea EustatServer_stub.class (per client)
- EustatServer_skel.class (per server)

GENENERAZIONE DINAMICA DI STUB



- nelle versioni di JAVA >5.0 lo stub viene generato dinamicamente utilizzando il meccanismo delle `Reflection`.

JAVA RMI > 1.5

- `Reflection`: meccanismo che permette a JAVA di ottenere informazioni sui metodi, i campi ed i costruttori di una classe

- Generare altre istanze di classi con le informazioni ottenute

- Esempi:
  - `String n = m.getName()` restituisce il nome del metodo
  - `Class[] cs = m.getDeclaringClass()` restituisce le classi che dichiarano quel metodo
  - `Class[] ps = m.getParameterTypes()` restituisce il tipo dei parametri

- Utilizzate per generare il codice dello stub/skeleton a partire dalla analisi dei metodi della interfaccia remota

JAVA RMI > 1.5

Nelle versioni di JAVA successive alla 1.5:

- se un'applicazione esporta un oggetto remoto:
  - invocando il costruttore di `UnicastRemoteObject`, oppure
  - invocando il metodo statico `exportObject`

contestualmente, se non si trova uno `stub` già generato per l'oggetto remoto, lo stub viene generato automaticamente

- lo stub è un'istanza della classe `java.lang.reflect.Proxy`

JAVA RMI > 1.5

In questo caso tutto è gestito a runtime, grazie alla reflection che si occuperà di creare uno stub da connettersi allo skeleton lato server, in maniera del tutto trasparente (effettuando sempre lo scambio di messaggi su protocollo TCP/IP).

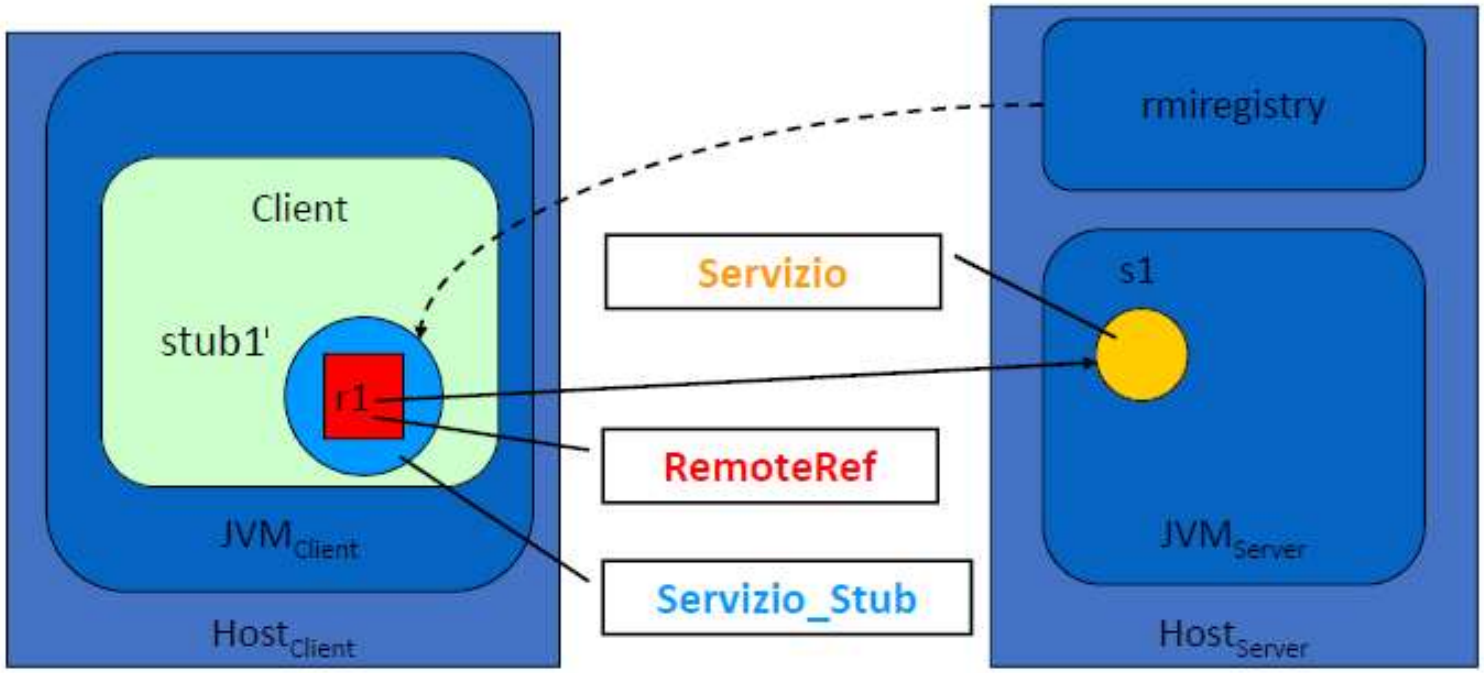
La classe Proxy su cui la generazione dinamica RMI si basa è una classe piuttosto semplice e il concetto alla base è quello di associare un oggetto che implementa l'interfaccia java.lang.reflect.InvocationHandler.

In RMI, viene costruito un oggetto di tipo RemoteObjectInvocationHandler passandogli una remote reference (RemoteRef).

```
public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable
```

- sugli oggetti passati come argomento alla **invoke()** deve essere effettuato marshalling per trasmetterli sulla rete (serializzazione)

STUB E RIFERIMENTI REMOTI



Il **Client** accede al **Server RMI** implementato dalla classe Servizio attraverso il riferimento remoto cioè lo *stub1* (istanza della classe Servizio_Stub e passata dal registry al client)

Servizio_Stub contiene al suo interno un **RemoteRef** (*r1*) che consente al RRL di raggiungere il server

From: Corradi, RMI, Università di Bologna

utente entra nel gruppo.

IL MECCANISMO DELLE CALLBACK

- lo stato di un gioco multiplayer viene gestito da un server.
  - i giocatori notificano al server le modifiche allo stato del gioco.
  - ogni giocatore deve essere avvertito quando lo stato del gioco subisce modifiche.
- gestione distribuita di un'asta:
  - ogni volta che un utente fa una nuova offerta, tutti i partecipanti all'asta devono essere avvertiti

IL MECCANISMO DELLE CALLBACK

Soluzioni possibili per realizzare nel server un servizio di notifica di eventi al client:

1) polling: il client interroga ripetutamente il server, per verificare l'occorrenza dell'evento atteso.

   interrogazione può avvenire mediante l'invocazione di un metodo remoto (mediante RMI).

   svantaggio: alto costo per l'uso non efficiente delle risorse del Sistema

2) registrazione dei client interessati agli eventi e successiva notifica (asincrona) del verificarsi dell'evento ai client, da parte del server

   vantaggi: il client può proseguire la sua elaborazione senza bloccarsi ed essere avvertito, in modo asincrono, quando l'evento si verifica

   occorre un meccanismo utilizzato dal server per risvegliare il client.

CALLBACK VIA RMI

- Si può utilizzare RMI per implementare un meccanismo di registrazione e notifica composto da:
  - interazione client-server (registrazione dell'interesse per un evento)
  - interazione server-client (notifica del verificarsi di un evento)

- notifica asincrona implementata mediante due invocazioni remote sincrone
- Il server definisce un'interfaccia remota **ServerInterface** con un metodo remoto per permettere al client di registrare il suo interesse per un certo evento
  - definisce un oggetto remoto ROS che implementa ServerInterface
- Il client definisce una interfaccia remota **ClientInterface** con un metodo remoto utilizzato dal server per notificare un evento al client
  - definisce un oggetto remoto ROC che implementa ClientInterface

CALLBACK VIA RMI: IL SERVER

- definisce un oggetto remoto ROS che implementa **ServerInterface:**
- implementazione del metodo di registrazione
  - parametro del metodo di registrazione: riferimento allo stub del client
- quando riceve una invocazione del metodo remoto, memorizza lo stub del ROC, riferimento all'oggetto remoto del client, in una sua struttura dati
- al momento della notifica, utilizza ROC per invocare il metodo remoto sul client, per la notifica
- NB il server riceve lo stub dell'oggetto remoto del client al momento della registrazione del client
- non utilizza il registry per individuare l'oggetto remoto del client, ma si fa passare il riferimento dal client che inizia l'interazione

CALLBACK VIA RMI: IL CLIENT

- definisce un `oggetto remoto ROC` che implementa **`ClientInterface`**
  - contiene un metodo che consente la notifica dell'evento atteso.
  - questo metodo verrà invocato dal server.

- ricerca l'oggetto remoto del server ROS che contiene il metodo per la registrazione mediante un servizio di `Registry`

- al momento della registrazione sul server, passa al server lo stub di `ROC`

- non registra l'oggetto remoto ROC in un registro

CALLBACK: UN ESEMPIO

- Un server gestisce le quotazioni di borsa di un titolo azionario. Ogni volta che si verifica una variazione del valore del titolo, vengono avvertiti tutti i client che si sono registrati per quell'evento.

- Il server definisce un oggetto remoto che fornisce metodi per consentire al client di registrare/cancellare una callback

- Il client vuole essere informato quando si verifica una variazione
- espone un metodo per la notifica
- registra una callback presso il server.
- aspetta per un certo intervallo di tempo durante cui riceve le variazioni di quotazione
- alla fine cancella la registrazione della propria callback presso il server

L'INTERFACCIA DEL CLIENT

```
import java.rmi.*;
public interface NotifyEventInterface extends Remote {
    /* Metodo invocato dal  server per notificare un evento ad un
client remoto. */
   public void notifyEvent(int value) throws RemoteException;
}
```

`notifyEvent(...)` è il metodo esportato dal client e che viene utilizzato dal server per la notifica di un nuova quotazione del titolo azionario

L'INTERFACCIA DEL CLIENT: IMPLEMENTAZIONE

```
import java.rmi.*;
import java.rmi.server.*;
public class NotifyEventImpl extends RemoteObject implements
                                NotifyEventInterface {
        /* crea un nuovo callback client */
  public NotifyEventImpl( ) throws RemoteException
        { super( ); }

  /* metodo che può essere richiamato dal servente per notificare
una   nuova quotazione del titolo */
  public void notifyEvent(int value) throws RemoteException {
        String returnMessage = "Update event received: " + value;
        System.out.println(returnMessage); }
        }
```

LANCIO DEL CLIENT

```
import java.rmi.*; import java.rmi.registry.*; import java.rmi.server.*;
import java.util.*;
public class Client {
     public static void main(String args[ ]) {
        try {
                              System.out.println("Cerco il Server");
             Registry registry = LocateRegistry.getRegistry(5000);
             String name = "Server";
             ServerInterface server =
                          (ServerInterface) registry.lookup(name);
             /* si registra per la callback */
              System.out.println("Registering for callback");
              NotifyEventInterface callbackObj =  new NotifyEventImpl();
              NotifyEventInterface stub = (NotifyEventInterface)

        UnicastRemoteObject.exportObject(callbackObj, 0)
              server.registerForCallback(stub);
```

ATTIVAZIONE DEL CLIENT

```
// attende gli eventi generati dal server per
// un certo intervallo di tempo;

 Thread.sleep (20000);

/* cancella la registrazione per la callback */
System.out.println("Unregistering for callback");
server.unregisterForCallback(stub);
  } catch (Exception e)
        { System.err.println("Client exception:"+ e.getMessage());}
 }

}
```

L'INTERFACCIA DEL SERVER

```
import java.rmi.*;
public interface ServerInterface extends Remote
   {
    /* registrazione per la callback */
    public void registerForCallback
       (NotifyEventInterface ClientInterface) throws RemoteException;

    /* cancella registrazione per la callback */
     public void  unregisterForCallback  (NotifyEventInterface
       ClientInterface) throws RemoteException;

   }
```

IL SERVER: IMPLEMENTAZIONE

```
import java.rmi.*; import java.rmi.server.*; import java.util.*;
public class ServerImpl extends RemoteServer implements ServerInterface
    { /* lista dei client registrati */
     private List <NotifyEventInterface> clients;
      /* crea un nuovo servente */
     public ServerImpl()throws RemoteException {
                                super( );
           clients = new ArrayList<NotifyEventInterface>( ); };
     public synchronized void registerForCallback
       (NotifyEventInterface ClientInterface) throws RemoteException {
                                if (!clients.contains(ClientInterface)) {
                                   clients.add(ClientInterface);
               System.out.println("New client registered." );
 }}
```

IL SERVER: IMPLEMENTAZIONE

```
/* annulla registrazione per il callback */
  public synchronized void unregisterForCallback
        (NotifyEventInterface Client) throws RemoteException {
            if (clients.remove(Client))
                  System.out.println("Client unregistered");
             else
                  System.out.println("Unable to unregister client");
               }
 /* notifica di una variazione di valore dell'azione
 /* quando viene richiamato, fa il callback a tutti i client
                registrati */
  public void update(int value) throws RemoteException {
                  doCallbacks(value);
  }
```

IL SERVER: IMPLEMENTAZIONE

```
private synchronized void doCallbacks(int value) throws
                                                    RemoteException {
                System.out.println("Starting callbacks.");
        Iterator i = clients.iterator( );
           //int numeroClienti = clients.size( );
        while (i.hasNext()) {
           NotifyEventInterface client =
                                   (NotifyEventInterface) i.next();
           client.notifyEvent(value);
        }
        System.out.println("Callbacks complete.");
     }
}
```

ATTIVAZIONE DEL SERVER

```
import java.rmi.server.*; import java.rmi.registry.*;
public class Server   {
        public static void main(String[  ] args) {
           try{ /*registrazione presso il registry */
                ServerImpl server =  new ServerImpl( );
                ServerInterface stub=(ServerInterface)
                        UnicastRemoteObject.exportObject (server,39000);
                String name = "Server";
                LocateRegistry.createRegistry(5000);
                Registry registry=LocateRegistry.getRegistry(5000);
                registry.bind  (name, stub);
          while (true) { int val=(int) (Math.random( )*1000);
                        System.out.println("nuovo update"+val);
                        server.update(val);
                        Thread.sleep(1500);}
              } catch (Exception e) { System.out.println("Eccezione" +e);}}}
```

RMI CALLBACKS: RIASSUNTO

- Il client crea un oggetto remoto, **`oggetto callback ROC`**, che implementa un'interfaccia remota che deve essere nota al server
- Il server definisce un **`oggetto remoto ROS`**, che implementa una interfaccia remota che deve essere nota al client
- Il client recupera **`ROS`** mediante il meccanismo di lookup di un registry
- **`ROS`** contiene un metodo che consente al client di registrare il proprio ROC presso il server
- quando il server ne ha bisogno, recupera un riferimento ad ROC dalla struttura dati in cui lo ha memorizzato al momento della registrazione e contatta il client via RMI

PASSAGGIO DI PARAMETRI

- tipi di entità che possono essere passate tra il client ed il server
  - valori di tipi di dati primitivi
  - oggetti remoti
  - oggetti locali serializzabili

- passati come argomenti di metodi e come variabili di ritorno

- regole per il passaggio di parametri
  - tipi di dati primitivi passati **per valore**
  - oggetti remoti passati **per riferimento**: viene passato lo Stub
  - oggetti locali passati **per valore, utilizzando la serializzazione**

- l'esempio dei lucidi successivi mostra l'utilizzo di queste regole

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

```
import java.rmi.*;

public interface Point extends Remote {
     public void move(int x, int y) throws RemoteException;
     public String getCoord() throws RemoteException;
}

import java.rmi.*;
public interface Circle extends Remote {
     String SERVICE_NAME = "CircleService";
     Point getCenter() throws RemoteException;
     double getRadius() throws RemoteException;
     void setCircle(Point c, double r) throws RemoteException;
}
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

```
import java.rmi.*;
import java.rmi.server.*;
public class PointImpl extends UnicastRemoteObject
                       implements Point {
    private int x, y;
    public PointImpl(int x, int y) throws RemoteException {
           this.x = x; this.y = y; }
    public void move(int x, int y) {
           this.x += x;
           this.y += y;
           System.out.println("Point has been moved to: " +
getCoord());  }
    public String getCoord() {
           return "[" + x + "," + y + "]"; }
}
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

```
import java.rmi.*;
import java.rmi.server.*;
public class CircleImpl extends UnicastRemoteObject implements Circle
{
      private Point center;
      private double radius;
      public CircleImpl(int x, int y, double r) throws
                                          RemoteException
             {setCircle(new PointImpl(x, y), r); }
      public void setCircle(Point c, double r) throws RemoteException
             {center = c; radius = r;
              System.out.println("Circle defined - Center: " +
                     center.getCoord() + " Radius: " + radius);}
      public Point getCenter() { return center; }
      public double getRadius() { return radius; }}
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

```
import java.rmi.*;
import java.rmi.registry.*;
public class CircleServer {
      public static void main(String args[]) throws Exception {
         Circle circle = new CircleImpl(10, 20, 30);
         LocateRegistry.createRegistry(9999);
         Registry r=LocateRegistry.getRegistry(9999);
         r.rebind(Circle.SERVICE_NAME, circle);
         System.out.println("Circle bound in registry");
     }}
```

Output:

```
Circle defined - Center: [10,20] Radius: 30.0
Circle bound in registry
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

```
import java.rmi.*
import java.rmi.registry.*;
public class CircleClient {
    public static void main(String[] args) throws Exception {
            Registry reg= LocateRegistry.getRegistry(9999);
            Circle circle = (Circle) reg.lookup(Circle.SERVICE_NAME);
            System.out.println(circle);
            double r = circle.getRadius() * 2;
            Point  p = circle.getCenter();
            System.out.println(p);
            p.move(30, 50);
            System.out.println("Circle - Center: " + p.getCoord() +
                               " Radius: " + r);
            circle.setCircle(p, r); }
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

Sul Server:

```
Point has been moved to: [40,70]
Circle defined - Center: [40,70] Radius: 60.0
```

Sul Client:

- i valori restituiti per `Circle` e `Point`, sono riferimenti remoti

```
Proxy[Circle,RemoteObjectInvocationHandler[UnicastRef [liveRef:
[endpoint:[192.168.1.25:54074](remote),objID:[-13ec7e8b:1546e1cec0e:-7fff,
465557802154876232 3]]]]]
Proxy[Point,RemoteObjectInvocationHandler[UnicastRef [liveRef:
[endpoint:[192.168.1.25:54074](remote),objID:[-13ec7e8b:1546e1cec0e:-7ffe,
212241888553049932 6]]]]]
```

- i valori restituiti per Center e Radius sono i loro valori reali

```
Circle - Center: [40,70] Radius: 60.0
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD"

Supponiamo ora che il server remoto non esporti l'oggetto `Point`.

```
public class PointLoc implements java.io.Serializable {
        private int x, y;
        public PointLoc (int x, int y) { this.x = x; this.y = y; }
        public void move(int x, int y) {
          this.x += x;
          this.y += y;
           System.out.println("Point has been moved to: " +
getCoord());
         }
        public String getCoord() {
            return "[" + x + "," + y + "]"}}
```

PASSAGGIO DI PARAMETRI: "UNDER THE HOOD

Si ottiene la seguente stampa:

```
Proxy[Circle,RemoteObjectInvocationHandler[UnicastRef [liveRef:
[endpoint:[192.168.1.25:54285](remote),objID:[-
16781560:1546e5ae81b:
-7fff, 2510422791299955034]]]]]
```

Viene stampata la seguente stringa per Point

```
PointLoc@1be6f5c3
```

```
Circle - Center: [40,70] Radius: 60.0
```

ASSIGNMENT 11: GESTIONE CONGRESSO

Si progetti un'applicazione Client/Server per la gestione delle registrazioni ad un congresso. L'organizzazione del congresso fornisce agli speaker delle varie sessioni un'interfaccia tramite la quale iscriversi ad una sessione, e la possibilità di visionare i programmi delle varie giornate del congresso, con gli interventi delle varie sessioni.

Il server mantiene i programmi delle 3 giornate del congresso, ciascuno dei quali è memorizzato in una struttura dati come quella mostrata di seguito, in cui ad ogni riga corrisponde una sessione (in tutto 12 per ogni giornata). Per ciascuna sessione vengono memorizzati i nomi degli speaker che si sono registrati (al massimo 5).

Sessione	Intervento 1	Intervento 2	…	…	Intervento 5
S1	Nome Speaker1	Nome Speaker2			
S2					
S3					
…					
S12					

ASSIGNMENT 11: GESTIONE CONGRESSO

Il client può richiedere operazioni per

- registrare uno speaker ad una sessione;
- ottenere il programma del congresso;

Il client inoltra le richieste al server tramite il meccanismo di RMI. Prevedere, per ogni possibile operazione una gestione di eventuali condizioni anomale (ad esempio la richiesta di registrazione ad una giornata e/o sessione inesistente oppure per la quale sono già stati coperti tutti gli spazi d'intervento)

Il client è implementato come un processo ciclico che continua a fare richieste sincrone fino ad esaurire tutte le esigenze utente. Stabilire una opportuna condizione di terminazione del processo di richiesta.