Lezione n.9
LPR- A
REMOTE METHOD INVOCATION

23/11/2008

Laura Ricci

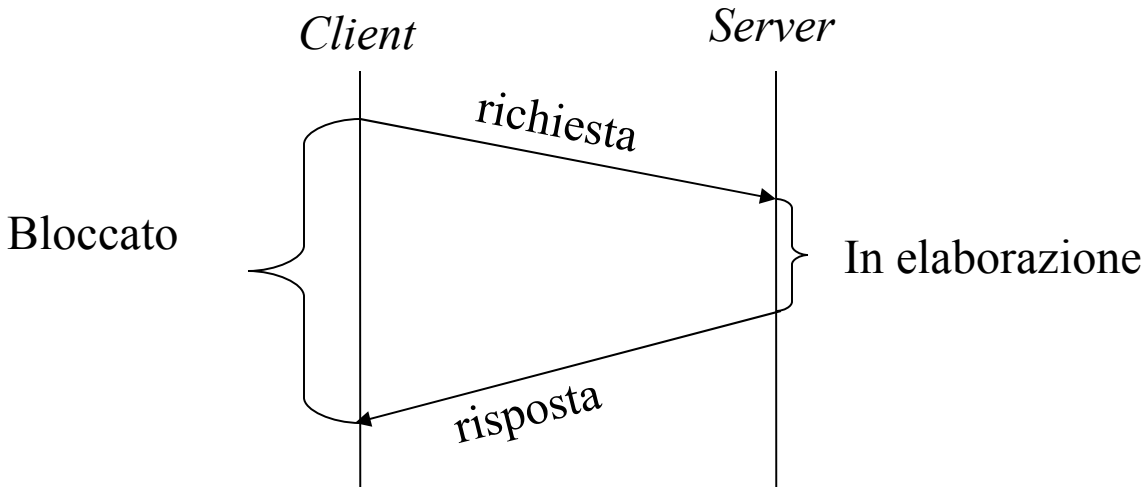
RIASSUNTO DELLA LEZIONE

- paradigma di interazione domanda/risposta
- remote procedure call
- RMI (Remote Method Invocation): API JAVA
- Esercizio
- Programmazione concorrente: il problema dei 5 filosofi

PARADIGMA DI INTERAZIONE A DOMANDA/RISPOSTA

Paradigma di interazione basato su richiesta/risposta

- il client invia ad un server un messaggio di richiesta
- il server risponde con un messaggio di risposta
- il client rimane bloccato (sospende la propria esecuzione) finchè non riceve la risposta dal server

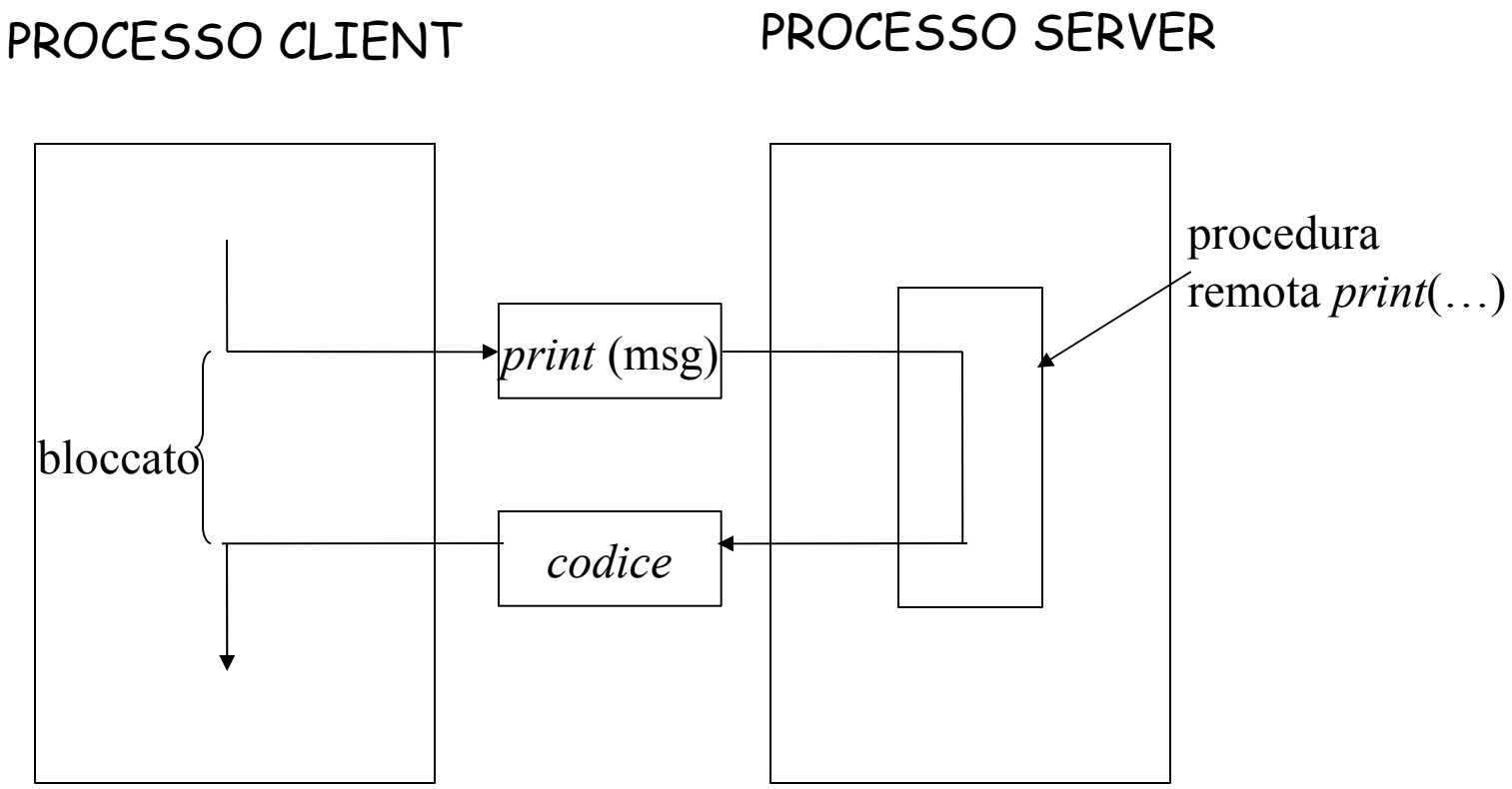


REMOTE PROCEDURE CALL

Esempio interazione domanda/risposta:

- un client richiede ad un server la stampa di un messaggio. Il server restituisce al client un codice che indica l'esito della operazione. Il client attende l'esito dell'operazione
- richiesta del client al server = invocazione di una procedura definita sul server
- Il client invoca una procedura remota RPC (Remote Procedure Call)
- I meccanismi utilizzati dal client sono gli stessi utilizzati per una normale invocazione di procedura, 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 all'RPC

REMOTE PROCEDURE CALL



Esempio: richiesta stampa di messaggio e restituzione esito operazione

REMOTE METHOD INVOCATION

implementazioni RPC

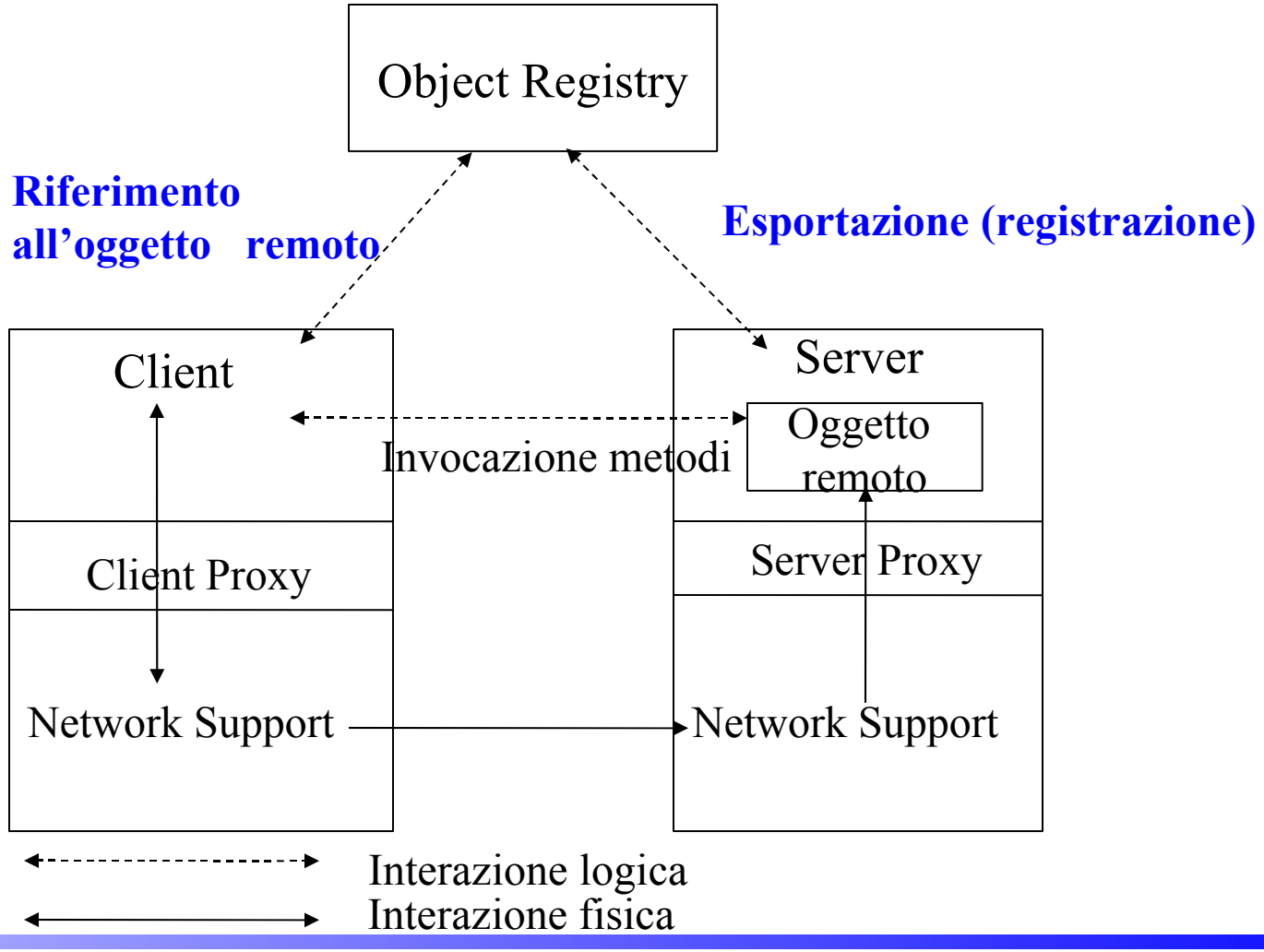
- Open Network Computing Remote Procedure Call (Sun)
- Open Group Distributed Computing Environment (DCE)
- …

Evoluzione di RPC = paradigma di interazione basato su oggetti distribuiti

remote method invocation (RMI): evoluzione del meccanismo diinvocazione di procedura remota al caso di oggetti remoti

JAVA RMI: JAVA API per la programmazione distribuita ad oggetti

OGGETTI DISTRIBUITI: ARCHITETTURA GENERALE



OGGETTI DISTRIBUITI: ARCHITETTURA GENERALE

Il server che definisce l'oggetto remoto:

- definisce un oggetto distribuito = un oggetto i cui metodi possono essere invocati da parte di processi in esecuzione su hosts remoti
- esporta (pubblica) l'oggetto: crea un mapping

  nome simbolico oggetto/ riferimento all'oggetto

  e lo pubblica mediante un servizio di tipo registry
  (simile ad un DNS per oggetti distribuiti)

OGGETTI DISTRIBUITI: ARCHITETTURA GENERALE

Quando il client vuole accedere all'oggetto remoto

- ricerca un riferimento all'oggetto remoto mediante i servizi offerti dal registry
- invoca i metodi definiti dall'oggeto remoto (remore method invocation).
- invocazione dei metodi di un oggetto remoto
  - a livello logico: identica all' invocazione di un metodo locale
  - a livello di supporto: è gestita da un client proxy 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

OGGETTI DISTRIBUITI: ARCHITETTURA GENERALE

Quando il server che gestisce l'oggetto remoto riceve un'invocazione per quell'oggetto

- il network support passa i dati ricevuti al server proxy
- il server proxy trasforma i dati ricevuti dal network support
  - in una invocazione ad un metodo locale
  - occorre trasformare i dati ricevuti dal network support nei parametri del metodo invocato

RMI: API JAVA

I metodi definiti dall'oggetto remoto ed i rispettivi parametri (le segnature dei metodi) devono essere noti:

- al client, che richiede un insieme di servizi mediante l'invocazione di tali metodi
- al server, che deve fornire un'implementazione di tali metodi

Il client non è interessato all' implementazione di tali metodi

In JAVA:

- definizione di un' interfaccia che contiene le segnature di un insieme di metodi, ma non il loro codice

- definizione di una classe che implementi l'interfaccia: contiene il codice dei metodi elencati nella interfaccia

INTERFACCE JAVA: RIPASSO

Una interfaccia JAVA può contenere solamente:

- metodi astratti e costanti: niente costruttori, niente variabili, solo l'intestazione dei metodi.
- i metodi sono tutti astratti, anche se manca la parola chiave abstract: infatti al posto del corpo c'è solo un punto e virgola;
- si può dichiarare che una classe implementa (implements) una data interfaccia: deve allora fornire un'implementazione per tutti i suoi metodi.
- una classe può implementare più di una interfaccia: la relazione implements non deve rispettare la regola dell'ereditarietà singola

INTERFACCE: RIPASSO

```
interface Figure {
/* gli oggetti delle classi che realizzano questa interfaccia sono
      caratterizzati da  un tipo  e da un'area
   */
   int PROVA = 5;
   double area(  );

   String tipo(   );
}
```

INTERFACCE: RIASSUNTO

```
class RadiceFigure {
    protected double dim1;
    protected double dim2;
      RadiceFigure(double a, double b)
      { dim1 = a; dim2 = b; }  }
class Rectangle extends RadiceFigure implements Figure{
       Rectangle(double a, double b) { super(a, b); }
 // definisce area() di Figure
 public double area( ) {   return dim1 * dim2;  }
// definisce tipo() di Figure
 public String tipo( ) { return "Rectangle";} }
```

INTERFACCE: RIASSUNTO

```
class Triangle extends RadiceFigure implements Figure {
        Triangle(double a, double b) {
                super(a, b);  }
    // definisce area( ) di Figure
    public double area( ) {  return dim1 * dim2 / 2;  }
    // definisce tipo( ) di Figure
    public String tipo( ) {
    return "Triangle ";
    }
}
```

INTERFACCE: RIPASSO

```
class prova {
public static void main (String args[ ]) {
// Figure f = new Figure(10, 10); // questo è illegale
Figure figref; //  OK non creo l' oggetto
figref = new Rectangle(9, 5);
System.out.println(figref.tipo( ) + figref.area( ));
figref = new Triangle(10, 8);
System.out.println(figref.tipo( ) + figref.area( ));
// figref.dim1 = 0 ; //anche questo e` illegale
 }
```

JAVA: REMOTE INTERFACE

Esempio 2: un Host è connesso ad una stazione metereologica che rileva temperatura, umidità,….mediante diversi strumenti di rilevazione. Sull'host è in esecuzione un server che fornisce queste informazioni agli utenti interessati.

```
import java.rmi.*;
public interface weather extends Remote;

public double getTemperature ( )    throws RemoteException;
public double getHumidity      ( )  throws RemoteException;
public double getWindSpeed   ( )    throws RemoteException;
```

JAVA RMI

I metodi esportati di un oggetto remoto devono essere definiti mediante un'interfaccia remota

- estende l'interfaccia Remote
- i metodi definiti possono sollevare eccezioni remote. Una eccezione remota indica un generico fallimento nella comunicazione remota dei parametri e dei risultati al/da il metodo remoto

Esempio 1: Definiamo un oggetto remoto che fornisce un servizio di echo, cioè ricevuto un valore come parametro, restituisce al chiamante lo stesso valore

- Passo 1. Definire una interfaccia che includa le segnature dei metodi che possono essere invocati da remoto
- Passo 2. Definire una classe che implementi l'interfaccia. Questa classe include l'implementazione di tutti i metodi che possono essere invocati da remoto

JAVA RMI

Passo 1. Definizione dell'interfaccia

```
import java.rmi.*;
public interface EchoInterface extends Remote {
        String getEcho (String Echo) throws RemoteException;  }
```

Remote è una interfaccia che non definisce alcun metodo. Il solo scopo è quello di identificare gli oggetti che possono essere utilizzati in remoto

Passo 2. Implementazione dell'interfaccia

```
public class Server implements EchoInterface {
public Server(   ) {   }
public String getEcho (String echo) {return echo ;     }   }
```

- definizione di una classe che implementa l'interfaccia remota
- la classe può definire ulteriori metodi pubblici, ma solamente quelli definiti nella interfaccia remota possono essere invocati da un altro host

JAVA RMI

Passo 3. Definire ed esportare un'istanza dell'oggetto remoto

```
import java.rmi.registry.Registry; import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;
public class ServerActivate {
public static void main(String args[ ]) {
try {  Server obj = new Server(   );
    EchoInterface stub = (EchoInterface)
                                UnicastRemoteObject.exportObject(obj, 0);
     Registry registry = LocateRegistry.getRegistry (  );
     registry.bind  ("Echo", stub);
     System.err.println("Server ready");
   } catch (Exception e) {
   System.err.println("Server exception: " + e.toString());  }}}
```

CREAZIONE E PUBBLICAZIONE DELL'OGGETTO REMOTO

Il server

- crea un'istanza dell'oggetto (new)
- invoca il metodo statico UnicastRemoteObject.exportObject(obj, 0) che
  - esporta l'oggetto remoto obj creato in modo che le invocazioni ai suoi metodi possano essere ricevute sulla porta specificata. La porta 0 specifica l'uso di una porta anonima
  - restituisce lo stub dell'oggetto remoto.
  - Stub = contiene uno scheletro per ogni metodo definito nell'interfaccia (con le solite segnature), ma trasforma l'invocazione di un metodo in una richiesta ad un host ed ad una porta remoto
- dopo aver eseguito il metodo, un server RMI aspetta invocazioni di metodi remoti su un serverscket legato alla porta specificata
- lo stub generato (da passare al client) contiene indirizzo IP e porta su cui è attivo il server RMI

CREAZIONE DELLO STUB

- Il server deve generare lo stub e renderlo disponibile al client
- NOTA BENE: se si utilizza una versione di JAVA antecedente alla 5, lo stub deve essere
  - generato mediante rmic (rmi compiler) e passato esplicitamente al client
- Nelle ultime versioni di JAVA è invece possibile utilizzare la exportObject per generare lo stub
- Per invocare i metodi dell'oggetto remoto, il client deve avere a disposizione lo stub dell'oggetto
- JAVA mette a disposizione del programmatore un semplice name server (registry) che consente
  - Al server di registrare lo stub con un nome simbolico
  - Al client di reperire lo stub tramite il suo nome simbolico

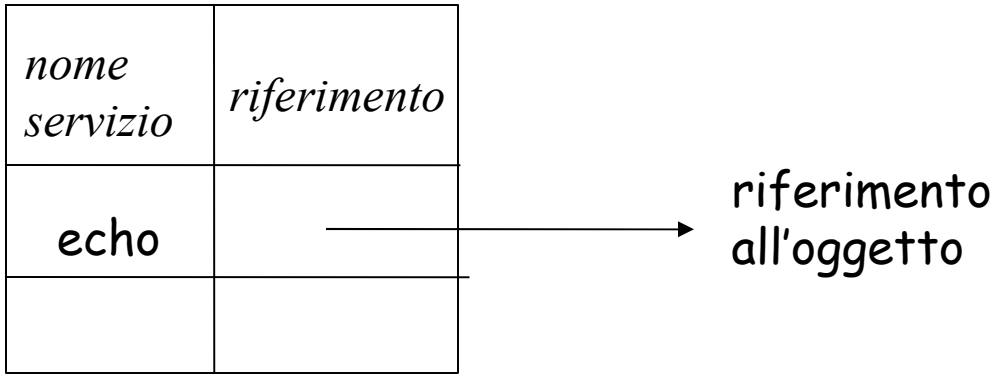
JAVA : ESPORTAZIONE DELLO STUB

Il Server

- per rendere disponibile lo stub creato agli eventuali clients, inserisce un riferimento allo stub creato nel registry locale (che deve essere attivo su local host sulla porta di default 1099)

```
Registry registry = LocateRegistry.getRegistry( );
registry.bind  ("Echo", stub);
```

- Registry = simile ad un DNS per oggetti remoti, contiene legami tra il nome simbolico dell'oggetto remoto ed il riferimento all'oggetto



JAVA: IL REGISTRY

- la classe LocateRegistry contiene metodi per la gestione dei registry
- La getRegistry( ) restituisce un riferimento ad un registry allocato sull'host locale e sulla porta di default 1099
- Si può anche specificare il nome di un host e/o una porta per individuare il servizio di registry su uno specifico host e/o porta
- nel caso più semplice si utilizza un registry locale, attivato sullo stesso host su cui è in esecuzione il server
- se non ci sono parametri oppure se il nome dell'host è uguale a null, allora l'host di riferimento è quello locale

RMI:IL SERVER

Dopo che il server ha registrato lo stub relativo all'oggetto remoto esportato,

- il main termina
- esiste comunque un thread attivo in attesa di invocazione di metodi remoti sul serversocket associato, per cui il programma non termina
- il thread rimane attivo fintanto che
  - Il binding creato per quell'oggetto rimane valido e
  - Esistono riferimenti all'oggetto remoto in clients remoti

JAVA : IL REGISTRY

Supponiamo che registry sia l'istanza di un registro individuato mediante getregistry( )

- registry.bind (...) 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, viene sollevata una eccezione

- registry.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

JAVA: ATTIVAZIONE DEL SERVIZIO

Per rendere disponibile i metodi dell'oggetto remoto, è necessario attivare due tipi di servizi

- il registry che fornisce il servizio di registrazione di oggetti remoti
- Il server implementato fornisce accesso ai metodi remoti

Per attivare il registry in background:

$ rmiregistry & (in LINUX)

$ start rmiregistry (in WINDOWS)

- viene attivato un registry associato per default alla porta 1099
- se la porta è già utilizzata, viene sollevata un'eccezione. Si può anche scegliere esplicitamente una porta

$ rmiregistry 2048 &

RMI REGISTRY

- Il registry viene eseguito per default sulla porta 1099
- Se si vuole eseguire il registry su una porta diversa, occorre specificare il numero di porta da linea di comando, al momento dell'attivazione

  start rmiregistry 2100

- la stessa porta va indicata sia nel client che nel server al momento del reperimento del riferimento al registro, mediante LocateRegistry.getRegistry

  Registry registry = LocateRegistry.getRegistry(2100);

- NOTA BENE: il registry ha bisogno dell'interfaccia e dei .class, per cui attenti a come sono impostati i path!

IL CLIENT RMI

Il client:

- ricerca uno stub per l'oggetto remoto
- invoca i metodi dell'oggetto remoto come fossero metodi locali (l'unica differenza è che occorre intercettare RemoteException)

Per ricercare il riferimento allo stub, il client

- deve accedere al registry attivato sul server.

- il riferimento restituito dal registry è un riferimento ad un oggetto di tipo Object: è necessario effettuare il casting dell'oggetto restituito al tipo definito nell'interfaccia remota

IL CLIENT RMI

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.*;
  public class Client {
    private Client( ) {  }
  public static void main(String[ ] args) throws Exception
   {
    String host = args[0];
            Scanner s = new Scanner(System.in);
            String next = s.next();
```

IL CLIENT RMI

```
try {
      Registry registry = LocateRegistry.getRegistry(host);
      EchoInterface stub = (EchoInterface) registry.lookup("Echo");
      String response = stub.getEcho(next);
      System.out.println("response: " + response);
} catch (Exception e) {
    System.err.println("Client exception: " + e.toString());
    e.printStackTrace();
}}}
```

LOCALIZZARE IL REGISTRY

- Forma generale del metodo LocateRegistry.getRegistry

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

  Restituisce un riferimento (stub) ad un oggetto Registry attivato sull'host e sulla porta specificata

- Il metodo può essere utilizzato dal client per individuare il servizio di registry attivato sul server

L'esecuzione del client richiede

- la classe EchoRMIClient.class, risultante della compilazione del client
- la classe EchoInterface.class

ESERCIZIO

Sviluppare una applicazione RMI per la gestione di un'elezione. Il server esporta un insieme di metodi

- **public void vota (String nome)**. Accetta come parametro il nome del candidato. Non restituisce alcun valore. Registra il voto di un candidato in una struttura dati opportunamente scelta.

- **public int risultato (String nome)** Accetta come parametro il nome di un candidato e restituisce i voti accumulati da tale candidato fino a quel momento.

- un metodo che consenta di ottenere i nomi di tutti i candidati, con i rispettivi voti, ordinati rispetto ai voti ottenuti

PROGRAMMAZIONE CONCORRENTE: IL PROBLEMA DEI 5 FILOSOFI

- cinque filosofi siedono ad una tavola rotonda con un piatto di spaghetti davanti, una forchetta (o bacchette cinesi, a seconda della versione)a destra e una forchetta a sinistra

- la vita di un filosofo alterna periodi in cui mangia a quelli in cui pensa

- ciascun filosofo ha bisogno di due forchette per mangiare, ma che le forchette vengano prese una per volta. Dopo essere riuscito a prendere due forchette il filosofo mangia per un pò, poi lascia le forchette e ricomincia a pensare

IL PROBLEMA DEI 5 FILOSOFI

- Problema proposto da E.Dijkstra nel '65: sviluppo di un algoritmo che impedisca lo stallo (deadlock) o la morte d'inedia (starvation).

- Il deadlock può verificarsi se ciascuno dei filosofi tiene in mano una forchetta senza mai riuscire a prendere l'altra. Il filosofo F1 aspetta di prendere la forchetta che ha in mano il filosofo F2, che aspetta la forchetta che ha in mano il filosofo F3, e così via in un circolo vizioso.

IL PROBLEMA DEI 5 FILOSOFI

- problema proposto da E.Dijkstra nel '65 come problema di sincronizzazione tra più attività concorrenti

- Starvation si verifica quando un processo non riesce mai ad acquisire le risorse di cui ha bisogno

- la situazione di starvation può verificarsi indipendentemente dal deadlock se uno dei filosofi non riesce mai a prendere entrambe le forchette.

FILOSOFI A PRANZO: LE BACCHETTE

```
public class bacchetta {
        private boolean taken;
        private int bacid;
        public bacchetta (int bacid)
            {this.bacid=bacid;
            this.taken = false;    }
```

FILOSOFI A PRANZO: LE BACCHETTE

```
public synchronized void take(int filid) throws InterruptedException
                { while (taken)
                      wait();
                      taken=true;
                      System.out.println(filid+"ho preso la racchetta"+bacid);
                      }
public synchronized void drop(int filid) {
                      taken = false;
                      System.out.println(filid+"ho lasciato la
                        racchetta"+bacid);
                      notifyAll();  } }
```

FILOSOFI A PRANZO: IL FILOSOFO

```
import java.util.*;
public class filosofo implements Runnable {
        private bacchetta left;
        private bacchetta right;
        private int filid;
        public filosofo (bacchetta left, bacchetta right, int filid)   {
                this.filid=filid;
                this.left=left;
                this.right=right;
                }
```

FILOSOFI A PRANZO: IL FILOSOFO

```
public void run( ) {
        try {     for (int i=0; i<2; i++){
                  System.out.println (filid+""+"thinking");
                  Thread.sleep (500);
                  right.take(filid);
                  left.take(filid);
                  System.out.println (filid+""+"Mangio");
                  Thread.sleep(1000);
                  right.drop(filid);
                  left.drop(filid);
            }}catch (InterruptedException e){} }
}
```

FILOSOFI A PRANZO: INIZIALIZZAZIONE

```
import java.util.concurrent.*;
public class filosoficondeadlock {
public static void main (String [ ] args) throws Exception
{int size =5;
if (args.length >1)  size = Integer.parseInt(args[1]);
ExecutorService exec = Executors.newFixedThreadPool(size);
bacchetta [  ] vectbacchetta = new bacchetta[size];
for (int i=0; i < size; i++)   vectbacchetta[i]= new bacchetta(i);
for (int i= 0; i< size; i++)
   {exec.execute(new filosofo(vectbacchetta[i],vectbacchetta[(i+1)%size],i));} }
    }
```

FILOSOFI A PRANZO: ESEMPIO DI ESECUZIONE

1thinking

0thinking

2thinking

4thinking

3thinking

1ho preso la racchetta2

1ho preso la racchetta1

1Mangio

2ho preso la racchetta3

4ho preso la racchetta0

4ho preso la racchetta4

4Mangio

4ho lasciato la racchetta0

4ho lasciato la racchetta4

4thinking

FILOSOFI A PRANZO: ESEMPIO DI ESECUZIONE

3ho preso la racchetta4

1ho lasciato la racchetta2

1ho lasciato la racchetta1

2ho preso la racchetta2

2Mangio

2ho preso la racchetta2

2Mangio

```
0ho preso la racchetta1
0ho preso la racchetta0
0Mangio
1thinking
2ho lasciato la racchetta3
0ho lasciato la racchetta1
0ho lasciato la racchetta0
0thinking.................
```

In questo caso il deadlock non si verifica...

FILOSOFI A PRANZO: ESECUZIONE CON DEADLOCK

Per provocare il deadlock modificare il codice in modo che un thread si sospenda dopo che ha preso la bacchetta di destra

```
public void run(){
try { for (int i=0; i<2; i++){
      System.out.println (filid+""+"thinking");
      Thread.sleep(500);
      right.take(filid);
      Thread.sleep(2500);
      left.take(filid);
      System.out.println (filid+"  "+"Mangio");
      Thread.sleep(1000);
      right.drop(filid);  left.drop(filid);  }}catch (InterruptedException e){ } } }
```

FILOSOFI A PRANZO: ESECUZIONE CON DEADLOCK

1thinking

2thinking

0thinking

3thinking

4thinking

1ho preso la racchetta2

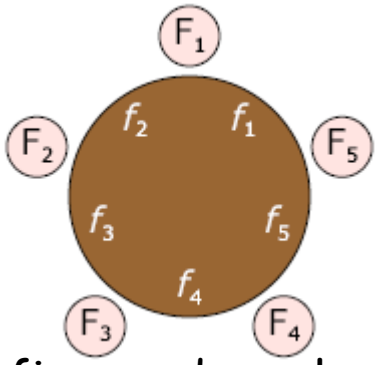
2ho preso la racchetta3

0ho preso la racchetta1

4ho preso la racchetta0

3ho preso la racchetta4

FILOSOFI A PRANZO: COME EVITARE IL DEADLOCK



Possibile soluzione: I filosofi prendono le bacchette in ordine numerico crescente.

F_1 deve prendere la bacchetta f_1 prima di poter prendere la seconda bacchetta f_2,i filosofi F_2, F_3 e F_4 si comportano in modo analogo, prendendo sempre la bacchetta f_i prima della bacchetta f_{i+1}

Invece il filosofo F_5 deve prendere prima la bacchetta b_1 e poi la bacchetta b_5. In questo modo si crea un'asimmetria che evita il deadlock.

FILOSOFI A PRANZO:EVITARE IL DEADLOCK

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class filsofisenzadeadlock {
public static void main (String [] args) throws Exception
{int size =5;
if (args.length >1)
size = Integer.parseInt(args[1]);
ExecutorService exec = Executors.newFixedThreadPool(5);
```

FILOSOFI A PRANZO: EVITARE IL DEADLOCK

```
Bacchetta [ ] vectbacchetta = new bacchetta[size];
for (int i=0; i < size; i++)
vectbacchetta[i]= new bacchetta(i);
for (int i= 0; i< size-1; i++)
   {exec.execute(new filosofo(vectbacchetta[i], vectbacchetta[(i+1)],i));}
    exec.execute(new filosofo(vectbacchetta[0], vectbacchetta[size],size-1));
   }
}
```