



**Lezione n.9**

**LPR-B-09**

**RMI: Remote Method  
Invocation**

**1/12/2009**

**Andrea Corradini**

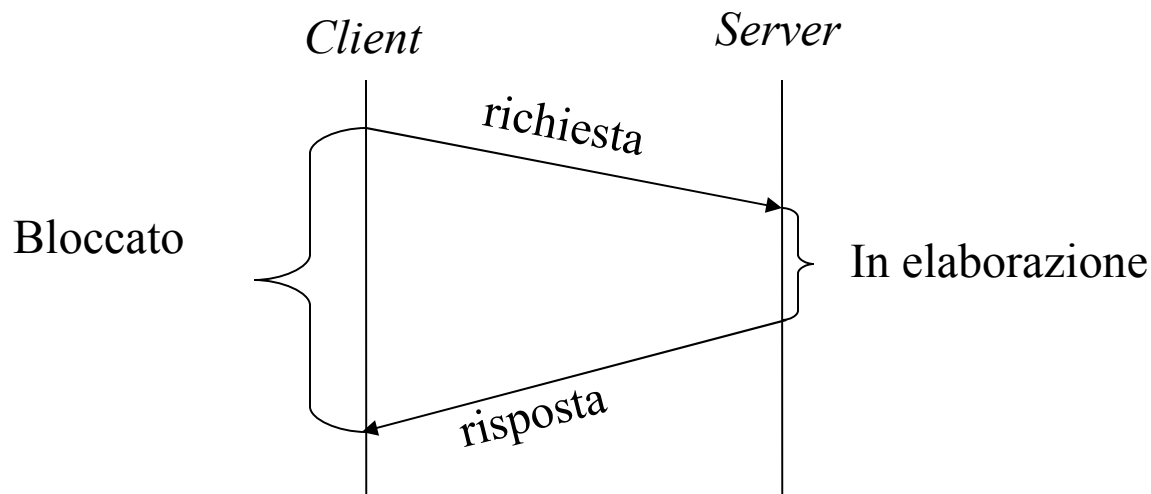
# Da UDP/TCP a RPC/RMI

- I protocolli visti (UDP, TCP) permettono a processi su host diversi di scambiarsi dati, più o meno strutturati:
  - sequenze di byte
  - dati primitivi
  - oggetti (con serializzazione)
- In molte **applicazioni distribuite** il livello di astrazione dei socket, non è adeguato.
- Il paradigma **Remote Procedure Call (RPC)** permette di astrarre da questi concetti: un programma può eseguire del codice su un host remoto con una normale chiamata di procedura.
- Nel contesto Object Oriented una "procedura" corrisponde a un metodo; in Java si parla quindi di **Remote Method Invocation (RMI)**.

# RPC/RMI: PARADIGMA DI INTERAZIONE A DOMANDA/RISPOSTA

Paradigma di interazione basato su richiesta/risposta

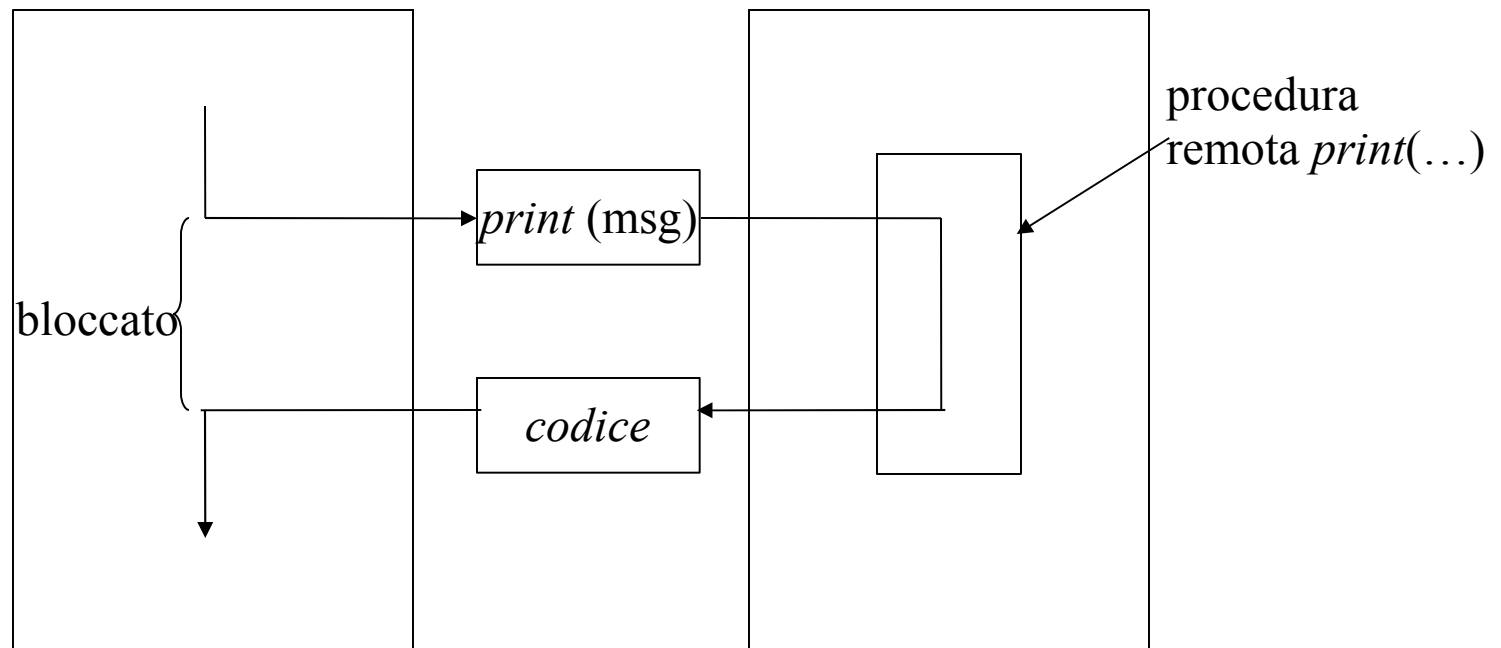
- il client invia ad un server un **messaggio di richiesta** (*invocazione di procedura/metodo*)
- il server risponde con un **messaggio di risposta** (*risultato*)
- il client rimane bloccato finchè non riceve la risposta dal server



# REMOTE PROCEDURE CALL: ESEMPIO

PROCESSO CLIENT

PROCESSO SERVER



Esempio: richiesta stampa di messaggio e restituzione esito operazione

# Esempi di Remote Procedure Call

Possibili esempi in cui RPC è utile:

- Esecuzione su un server molto potente di codice molto complesso
- Esecuzione su host remoto di interrogazioni su database, compreso elaborazione parziale dei risultati (es: media salari impiegati, ...)
- In generale, distribuzione del carico computazionale di un programma CPU-intensive tra più host
- Multiplayer games con stato centralizzato su di un server

# RPC: schema generale di implementazione

- Il server implementa delle procedure (metodi) e li offre come procedure remote tramite un'interfaccia
- Per invocare una procedura remota del server, il client si procura un *handle* ("maniglia"), una specie di *rappresentazione locale* della procedura
- L'invocazione comprende:
  - *marshalling* dei parametri;
  - spedizione al server;
  - *unmarshalling*;
  - invocazione della procedura.
- Il ritorno comprende:
  - *marshalling* del risultato;
  - spedizione al client;
  - *unmarshalling*;
  - consegna del valore di ritorno.

# RPC tra piattaforme eterogenee

- Se **client** e **server** sono scritti in linguaggi arbitrari, eventualmente diversi e su piattaforme diverse, per realizzare RPC bisogna:
  - fissare un formato per lo scambio di dati, per esempio **XDR** (**eXternal Data Representation**)
  - fornire traduzione tra formato nativo e formato di scambio
- Strumenti per il supporto di RPC:
  - **IDL (Interface Description Language)**
  - **Compilatore**
    - IDL -> Stub (lato client)
      - procedura che sostituisce quella del server implementando marshalling e unmarshalling
    - IDL -> Skeleton (lato server)
      - routine che realizza unmarshalling dei parametri, invocazione della procedura, marshalling del risultato

# USER VIEW

- Si crea un handle della procedura:  
`ProcedureHandle = lookup(registro, "nome");`
- Successivamente si invoca la procedura:  
`result = ProcedureHandle(param1, param2, ...);`
- Queste semplici operazioni sostituiscono:
  - apertura di connessione con host remoto
  - spedizione dei parametri
  - ricezione del risultato e sua memorizzazione
  - ... oltre all'implementazione del server remoto...

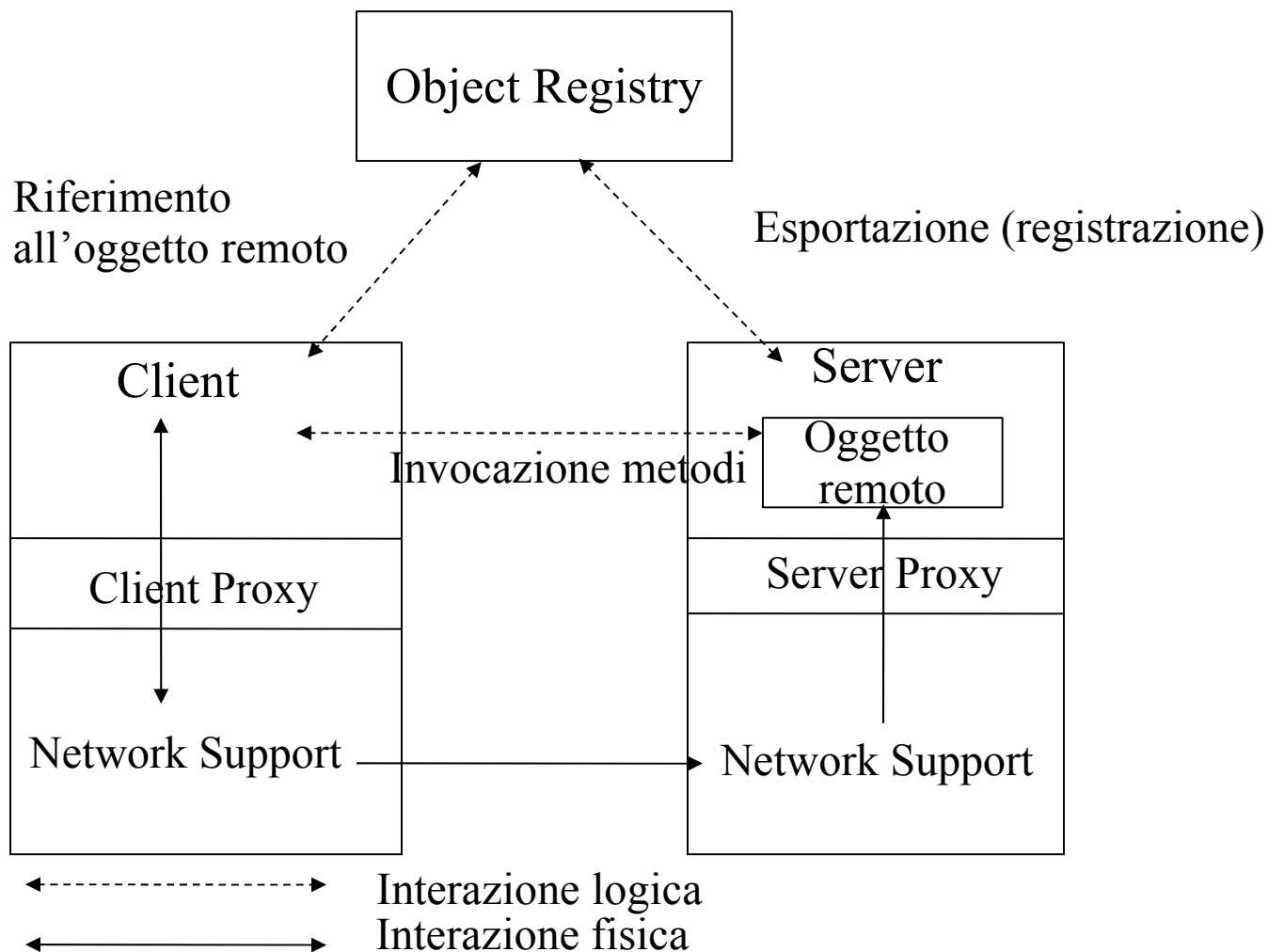


# RPC NEL PARADIGMA ORIENTATO AD OGGETTI

## Implementazioni RPC

- Open Network Computing Remote Procedure Call (Sun)
- Open Group Distributed Computing Environment (DCE)
- ...
- Nel contesto della Programmazione Orientata ad Oggetti, naturale evoluzione di RPC:
  - *procedure remote* => **oggetti remoti**
  - *chiamata di procedura* => **invocazione di metodo**
- **CORBA** (Common Object Request Broker Architecture) è un'architettura che supporta RPC in contesto OO tra sistemi eterogenei (es: Java e C++)
- **JAVA RMI**: API Java per la programmazione distribuita ad oggetti. Sfrutta il fatto che client e server sono entrambi in Java.

# OGGETTI REMOTI: ARCHITETTURA GENERALE



# RMI: ARCHITETTURA LATO SERVER

Il server che definisce l'oggetto remoto:

- definisce un'**interfaccia remota** che contiene i **metodi remoti** che possono essere invocati da parte di processi in esecuzione su hosts remoti
- **crea** un oggetto che implementa l'interfaccia remota, lo **esporta** per farlo diventare un "RMI server", e lo **pubblica** in un **registry** (registro) associandolo a un nome simbolico
- i metodi remoti invocati da client diversi sono eseguiti **concorrentemente**, se non sono sincronizzati.
- Importante la separazione tra interfaccia pubblica e implementazione (privata)

# RMI: ARCHITETTURA LATO CLIENT

- Quando il client vuole accedere all'oggetto remoto ricerca un riferimento all'oggetto remoto mediante i servizi offerti dal registry
  - operazione di **lookup**
  - il cliente deve conoscere
    - host su cui è eseguito il registry
    - nome simbolico pubblicato nel registry per l'oggetto
  - il risultato del lookup è un oggetto (handle) il cui tipo è l'**interfaccia remota** implementata dall'oggetto remoto
  - sullo handle ottenuto il cliente invoca i metodi definiti dall'oggetto remoto (**remote method invocation**).

# RMI: JAVA API lato Server

- Interfaccia remota:
  - extends **java.rmi.Remote** (marker interface)
- Nell'implementazione dell'oggetto remoto
  - i metodi invocabili da remoto devono prevedere il lancio di **java.rmi.RemoteException**
- Per **esportare l'oggetto remoto** e farlo diventare "**sever RMI**", due tecniche:
  - **extends UnicastRemoteObject** nella dichiarazione della classe che definisce l'oggetto remoto
  - oppure prima della pubblicazione si usa il metodo statico **UnicastRemoteObject.exportObject(Object)**:
    - bisogna prima creare la classe dello stub usando
      - **rmic ClasseOggettoRemoto**

# RMI: JAVA API lato Server II

- **Pubblicazione dell'oggetto:**
  - crea un binding (associazione)  
nome simbolico oggetto/ riferimento all'oggetto
  - lo pubblica mediante un servizio di tipo **registry** (registro)  
(simile ad un **DNS** per oggetti distribuiti)
- **Attivazione del RMI registry (porta default 1099):**
  - `rmiregistry &` (Linux)
  - `start rmiregistry` (Windows)

# RMI: JAVA API lato Server II

- Accesso al registro usando la classe `java.rmi.Naming`
  - void Naming.**bind**(String name, Remote obj)
  - void Naming.**rebind**(String name, Remote obj)
  - **Remote** Naming.**lookup**(String name)
    - name ha la forma "**//host:port/nome**", con host e port opzionali e default "**//localhost:1099/**".
- Accesso al registro usando la classe `java.rmi.LocateRegistry`
  - Registry LocateRegistry.**getRegistry**([String host], [int port])
    - restituisce un riferimento al registro, sul quale posso invocare i metodi `bind()`, `rebind()`, `lookup()`, `list()`, `unbind()`

# ESEMPIO DI REMOTE INTERFACE

**Esempio:** 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. L'interfaccia remota potrebbe essere la seguente:

```
import java.rmi.*;
```

```
public interface weather extends Remote{
```

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

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

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

```
}
```



# ESEMPIO RMI: un servizio di ECHO

---

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 firme 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

# ESEMPIO RMI: un servizio di ECHO

## Passo1. Definizione dell'interfaccia

```
import java.rmi.*;

public interface EchoInterface extends Remote {

    String getEcho (String Echo) throws RemoteException; }
```

## Passo2. Implementazione dell'interfaccia

```
public class Server implements EchoInterface {

    public Server( ) { }

    ▪ public String getEcho (String echo) throws RemoteException;
      {return echo; } }

    ▪ la classe può definire ulteriori metodi pubblici, ma solamente quelli
      definiti nella interfaccia remota possono essere invocati da un altro
      host
```

## ESEMPIO RMI: un servizio di ECHO

Passo 3. Creare, esportare e pubblicare 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);
        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)** che
  - **esporta** l'oggetto remoto **obj** creato in modo che le invocazioni ai suoi metodi possano essere ricevute su una porta anonima.
  - **restituisce lo stub** dell'oggetto remoto. Il client deve reperire questo stub per poter invocare i metodi remoti
- **NOTA BENE:** la classe dello stub deve essere creata prima esplicitamente, mediante il comando **rmic (rmi compiler)**: > **rmic Server**
- dopo aver eseguito il metodo, sulla porta anonima individuata, un server RMI aspetta invocazioni di metodi remoti su un **ServerSocket**
- Lo stub generato (da passare al client) contiene indirizzo IP e porta su cui è attivo il server RMI

# CREAZIONE DELLO STUB

- Per invocare i metodi dell'oggetto remoto, il client deve avere a disposizione lo stub dell'oggetto
- Stub = contiene uno scheletro per ogni metodo definito nell'interfaccia (con le solite signature), ma trasforma l'invocazione di un metodo in una richiesta a un host remoto (a una porta opportuna)
- Il client deve reperire lo stub e utilizzarlo per invocare i metodi remoti
- 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

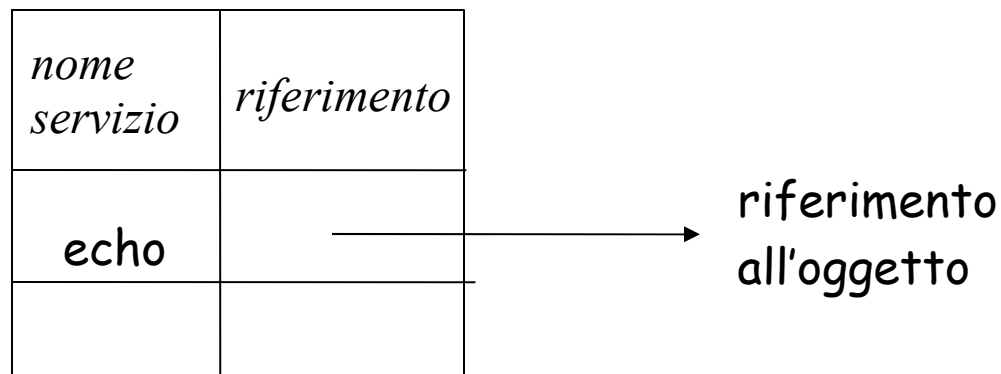
# JAVA : ESPORTAZIONE DELLO STUB

- Il Server per rendere disponibile lo stub creato agli eventuali clients, inserisce un riferimento allo stub creato in un **registry**. Normalmente useremo il **registry locale**, che deve essere attivo sul localhost sulla porta di **default 1099** (altrimenti va attivato come visto prima).

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

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

Attivazione del 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();  
}}
```

# ESEMPIO: FIBONACCI

- Il server calcola il numero di **Fibonacci** per numeri di grandezza arbitraria.
- Rispetto all'esempio **Echo**, questo esempio usa API più semplici per esportare e pubblicare l'oggetto remoto.

## L'Interfaccia Remota

---

```
import java.rmi.*;
import java.math.BigInteger;
public interface RemoteFibonacci extends Remote {
    public BigInteger getFibonacci(BigInteger n)
        throws RemoteException;
}
```

# ESEMPIO: FIBONACCI - L'OGGETTO REMOTO

```
import java.rmi.*; import java.rmi.server.UnicastRemoteObject;
import java.math.BigInteger;
public class FibonacciImpl extends UnicastRemoteObject implements
                                   RemoteFibonacci {
    public FibonacciImpl( ) throws RemoteException {
        super( );
    }
    public BigInteger getFibonacci(BigInteger n)
                                   throws RemoteException {
        System.out.println("Calculating the " + n + "th Fibonacci number");
        //CONTINUA
    }
}
```

# ESEMPIO: FIBONACCI - L'OGGETTO REMOTO

```
// Continua...
BigInteger zero = new BigInteger("0");
BigInteger one  = new BigInteger("1");
if (n.equals(zero)) return one;
if (n.equals(one)) return one;
BigInteger i = one, low = one, high = one;
while (i.compareTo(n) == -1) {
    BigInteger temp = high;
    high = high.add(low);
    low = temp;
    i = i.add(one);
}
return high;    }}
```



# ESEMPIO: FIBONACCI - PUBBLICAZIONE

```
import java.net.*;    import java.rmi.*;
public class FibonacciServer {
    public static void main(String[] args) {
        try {
            FibonacciImpl f = new FibonacciImpl( );
            Naming.rebind("fibonacci", f);
            System.out.println("Fibonacci Server ready.");
        } catch (RemoteException rex) {
            System.out.println("Exception in FibonacciImpl.main: " + rex);
        } catch (MalformedURLException ex) {
            System.out.println("MalformedURLException " + ex);
        }
    }
}
```

# ESEMPIO: FIBONACCI - IL CLIENT

```
import java.rmi.*; import java.net.*; import java.math.BigInteger;
public class RemoteFibonacciClient {
    public static void main(String args[]) {
        if (args.length == 0 || !args[0].startsWith("rmi:")) {
            System.err.println("Usage: java FibonacciClient " +
                " rmi://host.domain:port/fibonacci number"); return; }
        try {Object o = Naming.lookup(args[0]);
            RemoteFibonacci calculator = (RemoteFibonacci) o;
            for (int i = 1; i < args.length; i++) {
                try {BigInteger index = new BigInteger(args[i]);
                    BigInteger f = calculator.getFibonacci(index);
                    System.out.println("The " + args[i] +
                        "th Fibonacci number is " + f); }
            }
        }
    }
}
```

# ESEMPIO: FIBONACCI - IL CLIENT

```
        catch (NumberFormatException e) {
            System.err.println(args[i] + "is not an integer.");
        }
    }
} catch (MalformedURLException ex) {
    System.err.println(args[0] + " is not a valid RMI URL");
} catch (RemoteException ex) {
    System.err.println("Remote object threw exception " + ex);
} catch (NotBoundException ex) {
    System.err.println(
        "Could not find the requested remote object on the server");
}
}}
```

# 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