

Programmazione Distribuita in Java

Lorenzo Bettini

Dip. Sistemi e Informatica, Univ. Firenze
<http://www.dsi.unifi.it/~bettini>

Ottobre 2005

Strumenti per il networking in Java

- Libreria `java.net`
 - classi `Socket`, `ServerSocket`, `DatagramSocket`, `DatagramPacket`
- Serializzazione
 - scrivere/leggere un oggetto di una qualsiasi classe (che implementi l'interfaccia `java.io.Serializable`)
- Caricamento dinamico delle classi
 - possibilità di caricare una nuova classe scaricata dalla rete dinamicamente (es. le applet)
- RMI (Remote Method Invocation)

Programmazione Client–Server

- *Server*: Entità che mette a disposizione delle risorse e dei servizi

Programmazione Client–Server

- *Server*: Entità che mette a disposizione delle risorse e dei servizi
- *Client*: Entità che richiede risorse e servizi al server

Programmazione Client–Server

- *Server*: Entità che mette a disposizione delle risorse e dei servizi
- *Client*: Entità che richiede risorse e servizi al server
- le richieste e le risposte seguono un determinato protocollo di comunicazione, comune sia al client che al server

Programmazione Client–Server

- *Server*: Entità che mette a disposizione delle risorse e dei servizi
- *Client*: Entità che richiede risorse e servizi al server
- le richieste e le risposte seguono un determinato protocollo di comunicazione, comune sia al client che al server

Web

Il server web mette a disposizione le pagine HTML (ed altri dati multimediali).

Il client (browser web) richiede le pagine HTML e le visualizza per l'utente.

Il protocollo di comunicazione è l'HTTP.

Internet programming

- Il protocollo principale è il *TCP/IP* (che è in realtà una suite di protocolli)

Internet programming

- Il protocollo principale è il *TCP/IP* (che è in realtà una suite di protocolli)
- L'astrazione principale per la programmazione è la *Socket*:

Internet programming

- Il protocollo principale è il *TCP/IP* (che è in realtà una suite di protocolli)
- L'astrazione principale per la programmazione è la *Socket*:
 - permette di stabilire un canale di comunicazione con un'altro processo (eventualmente su un'altra macchina)

Internet programming

- Il protocollo principale è il *TCP/IP* (che è in realtà una suite di protocolli)
- L'astrazione principale per la programmazione è la *Socket*:
 - permette di stabilire un canale di comunicazione con un'altro processo (eventualmente su un'altra macchina)
 - tutto quello che viene trasmesso sono pacchetti TCP/IP

Internet programming

- Il protocollo principale è il *TCP/IP* (che è in realtà una suite di protocolli)
- L'astrazione principale per la programmazione è la *Socket*:
 - permette di stabilire un canale di comunicazione con un'altro processo (eventualmente su un'altra macchina)
 - tutto quello che viene trasmesso sono pacchetti TCP/IP
- I partecipanti di una comunicazione tramite socket sono individuati da

Internet programming

- Il protocollo principale è il *TCP/IP* (che è in realtà una suite di protocolli)
- L'astrazione principale per la programmazione è la *Socket*:
 - permette di stabilire un canale di comunicazione con un'altro processo (eventualmente su un'altra macchina)
 - tutto quello che viene trasmesso sono pacchetti TCP/IP
- I partecipanti di una comunicazione tramite socket sono individuati da
 - indirizzo IP

Internet programming

- Il protocollo principale è il *TCP/IP* (che è in realtà una suite di protocolli)
- L'astrazione principale per la programmazione è la *Socket*:
 - permette di stabilire un canale di comunicazione con un'altro processo (eventualmente su un'altra macchina)
 - tutto quello che viene trasmesso sono pacchetti TCP/IP
- I partecipanti di una comunicazione tramite socket sono individuati da
 - indirizzo IP
 - numero di porta

Internet programming

- Il protocollo principale è il *TCP/IP* (che è in realtà una suite di protocolli)
- L'astrazione principale per la programmazione è la *Socket*:
 - permette di stabilire un canale di comunicazione con un'altro processo (eventualmente su un'altra macchina)
 - tutto quello che viene trasmesso sono pacchetti TCP/IP
- I partecipanti di una comunicazione tramite socket sono individuati da
 - indirizzo IP
 - numero di porta
- anche il concetto di porta è astratto e NON corrisponde ad una porta fisica del computer.

Porte

- Il numero di porta è compreso fra 1 e 65535
- Le porte inferiori a 1024 sono riservate a servizi standard:
 - 80: web
 - 21: FTP
 - 25: posta elettronica
 - 119: news group

Stabilire una comunicazione (lato client)

- Aprire una socket specificando indirizzo IP e numero di porta del server (remoto)

Stabilire una comunicazione (lato client)

- Aprire una socket specificando indirizzo IP e numero di porta del server (remoto)
- All'indirizzo e numero di porta specificati ci deve essere in ascolto un processo server

Stabilire una comunicazione (lato client)

- Aprire una socket specificando indirizzo IP e numero di porta del server (remoto)
- All'indirizzo e numero di porta specificati ci deve essere in ascolto un processo server
- se la connessione ha successo si usano gli stream (di lettura e scrittura) associati alla socket, per comunicare col server remoto

Stabilire una comunicazione (lato client)

- Aprire una socket specificando indirizzo IP e numero di porta del server (remoto)
- All'indirizzo e numero di porta specificati ci deve essere in ascolto un processo server
- se la connessione ha successo si usano gli stream (di lettura e scrittura) associati alla socket, per comunicare col server remoto

Localhost

Per fare test in locale si può utilizzare l'indirizzo (riservato) di localhost, 127.0.0.1, che corrisponde al computer locale stesso.

Accettare una connessione (lato server)

- Creare un'istanza della classe `java.net.ServerSocket` specificando il numero di porta su cui rimanere in ascolto

Accettare una connessione (lato server)

- Creare un'istanza della classe `java.net.ServerSocket` specificando il numero di porta su cui rimanere in ascolto
- Chiamare il metodo `accept()` che rimane in ascolto di una richiesta di connessione (la porta non deve essere già in uso)

Accettare una connessione (lato server)

- Creare un'istanza della classe `java.net.ServerSocket` specificando il numero di porta su cui rimanere in ascolto
- Chiamare il metodo `accept()` che rimane in ascolto di una richiesta di connessione (la porta non deve essere già in uso)
- Quando il metodo ritorna la connessione col client è stabilita

Accettare una connessione (lato server)

- Creare un'istanza della classe `java.net.ServerSocket` specificando il numero di porta su cui rimanere in ascolto
- Chiamare il metodo `accept()` che rimane in ascolto di una richiesta di connessione (la porta non deve essere già in uso)
- Quando il metodo ritorna la connessione col client è stabilita
- Il metodo restituisce un'istanza di `java.net.Socket` connessa al client remoto.

Accettare una connessione (lato server)

- Creare un'istanza della classe `java.net.ServerSocket` specificando il numero di porta su cui rimanere in ascolto
- Chiamare il metodo `accept()` che rimane in ascolto di una richiesta di connessione (la porta non deve essere già in uso)
- Quando il metodo ritorna la connessione col client è stabilita
- Il metodo restituisce un'istanza di `java.net.Socket` connessa al client remoto.

Porte privilegiate

Il sistema operativo potrebbe proibire agli utenti standard di utilizzare le porte con numero inferiore a 1024 per attendere richieste di connessione.

Esempio: echo server

```
import java.io.*;
import java.net.*;

public class EchoServer {
    public EchoServer(int port) throws IOException {
        ServerSocket serverSocket = new ServerSocket(port);
        Socket socket = serverSocket.accept();
        System.out.println("ricevuta connessione: " + socket.getInetAddress() +
            ":" + socket.getPort());
        DataInputStream dataInputStream =
            new DataInputStream(socket.getInputStream());
        DataOutputStream dataOutputStream =
            new DataOutputStream(socket.getOutputStream());
        while (true) {
            char c = dataInputStream.readChar();
            dataOutputStream.writeChar(c);
        }
    }
}
```

Client? telnet

Non importa scriversi un client per testare questo server: si può utilizzare **telnet** per comunicare col server.

Ogni volta che premiamo INVIO quello che abbiamo scritto viene spedito al server, e quello che viene inviato dal server viene scritto a schermo:

```
telnet localhost 9999
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
ciao mondo!
ciao mondo!
```

Efficienza

- Il programma precedente legge e scrive un carattere alla volta
- Questo può essere inefficiente nelle applicazioni distribuite
- È meglio bufferizzare soprattutto le scritture
 - In questo modo si possono inserire più informazioni in un singolo pacchetto TCP/IP

Esempio: echo server, una linea alla volta

```
BufferedReader bufferedReader =  
    new BufferedReader(new InputStreamReader(socket.getInputStream()));  
BufferedWriter bufferedWriter =  
    new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));  
while (true) {  
    String string = bufferedReader.readLine();  
    bufferedWriter.write(string);  
    bufferedWriter.newLine();  
    bufferedWriter.flush();  
}
```

Flush the stream

È fondamentale richiamare `flush()` per assicurarsi di scaricare il buffer (il suo contenuto verrà effettivamente spedito al destinatario).

Line client

```

public class LineClient {
    public LineClient(String host, int port)
        throws UnknownHostException, IOException {
        System.out.println("connecting to " + host + ":" + port + " ...");
        Socket socket = new Socket(host, port);
        System.out.println("connected");
        BufferedReader keyboardReader =
            new BufferedReader(new InputStreamReader(System.in));
        BufferedReader bufferedReader =
            new BufferedReader(new InputStreamReader(socket.getInputStream()));
        BufferedWriter bufferedWriter =
            new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
        while (true) {
            System.out.print("client: ");
            String string = keyboardReader.readLine();
            bufferedWriter.write(string);
            bufferedWriter.newLine();
            bufferedWriter.flush();
            System.out.println("server: " + bufferedReader.readLine());
        }
    }
}

```

Server multi threaded

- Il server che abbiamo visto accetta una sola connessione (da un solo client)

Server multi threaded

- Il server che abbiamo visto accetta una sola connessione (da un solo client)
- Tipicamente, invece, un server dovrebbe essere in grado di accettare connessioni da diversi client e di dialogare con questi “contemporaneamente”:

Server multi threaded

- Il server che abbiamo visto accetta una sola connessione (da un solo client)
- Tipicamente, invece, un server dovrebbe essere in grado di accettare connessioni da diversi client e di dialogare con questi “contemporaneamente”:
 - Un thread esegue l'accept()

Server multi threaded

- Il server che abbiamo visto accetta una sola connessione (da un solo client)
- Tipicamente, invece, un server dovrebbe essere in grado di accettare connessioni da diversi client e di dialogare con questi “contemporaneamente”:
 - Un thread esegue l'accept()
 - Appena riceve una connessione fa partire un altro thread che si occuperà di quella connessione

Server multi threaded

- Il server che abbiamo visto accetta una sola connessione (da un solo client)
- Tipicamente, invece, un server dovrebbe essere in grado di accettare connessioni da diversi client e di dialogare con questi “contemporaneamente”:
 - Un thread esegue l'accept()
 - Appena riceve una connessione fa partire un altro thread che si occuperà di quella connessione
 - E torna ad eseguire l'accept()

Server multi threaded

- Il server che abbiamo visto accetta una sola connessione (da un solo client)
- Tipicamente, invece, un server dovrebbe essere in grado di accettare connessioni da diversi client e di dialogare con questi “contemporaneamente”:
 - Un thread esegue l'accept()
 - Appena riceve una connessione fa partire un altro thread che si occuperà di quella connessione
 - E torna ad eseguire l'accept()
 - Ogni thread si occuperà di un solo client, senza interferire con gli altri.

Server multi threaded

- Il server che abbiamo visto accetta una sola connessione (da un solo client)
- Tipicamente, invece, un server dovrebbe essere in grado di accettare connessioni da diversi client e di dialogare con questi “contemporaneamente”:
 - Un thread esegue l'accept()
 - Appena riceve una connessione fa partire un altro thread che si occuperà di quella connessione
 - E torna ad eseguire l'accept()
 - Ogni thread si occuperà di un solo client, senza interferire con gli altri.
- La stessa istanza di `ServerSocket` può essere usata per ricevere connessioni differenti.

Server multi threaded

- Il server che abbiamo visto accetta una sola connessione (da un solo client)
- Tipicamente, invece, un server dovrebbe essere in grado di accettare connessioni da diversi client e di dialogare con questi “contemporaneamente”:
 - Un thread esegue l'accept()
 - Appena riceve una connessione fa partire un altro thread che si occuperà di quella connessione
 - E torna ad eseguire l'accept()
 - Ogni thread si occuperà di un solo client, senza interferire con gli altri.
- La stessa istanza di `ServerSocket` può essere usata per ricevere connessioni differenti.
- Ogni istanza `Socket` restituita da `accept()` può essere utilizzata per dialogare solo con quel particolare client.

Echo server (multi threaded)

```
public class EchoMultiServer {
    public EchoMultiServer(int port) {
        try {
            ServerSocket serverSocket = new ServerSocket(port);
            while (true) {
                System.out.println("in attesa su " + port);
                Socket socket = serverSocket.accept();
                System.out.println("ricevuta connessione: "
                    + socket.getInetAddress() + ":" +
                    socket.getPort());
                (new ClientHandler(socket)).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Echo server (multi threaded)

```
class ClientHandler extends Thread {
    Socket socket;
    public ClientHandler(Socket socket) { this.socket = socket; }

    public void run() {
        try {
            BufferedReader bufferedReader =
                new BufferedReader(new InputStreamReader(socket.getInputStream()));
            BufferedWriter bufferedWriter =
                new BufferedWriter(new OutputStreamWriter
                    (socket.getOutputStream()));

            while (true) {
                String string = bufferedReader.readLine();
                bufferedWriter.write(string);
                bufferedWriter.newLine();
                bufferedWriter.flush();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Interruzione thread

- Le operazioni di lettura da stream non gestiscono le `InterruptedException`

Interruzione thread

- Le operazioni di lettura da stream non gestiscono le `InterruptedException`
- Quindi non serve richiamare `interrupt()` su un thread che legge da uno stream

Interruzione thread

- Le operazioni di lettura da stream non gestiscono le `InterruptedException`
- Quindi non serve richiamare `interrupt()` su un thread che legge da uno stream
- Si può però chiudere lo stream dal quale il thread legge, oppure direttamente la `Socket`, richiamando il metodo `close()`

Interruzione thread

- Le operazioni di lettura da stream non gestiscono le `InterruptedException`
- Quindi non serve richiamare `interrupt()` su un thread che legge da uno stream
- Si può però chiudere lo stream dal quale il thread legge, oppure direttamente la `Socket`, richiamando il metodo `close()`
- In tal caso il thread riceverà una `java.io.IOException`

Interruzione thread

- Le operazioni di lettura da stream non gestiscono le `InterruptedException`
- Quindi non serve richiamare `interrupt()` su un thread che legge da uno stream
- Si può però chiudere lo stream dal quale il thread legge, oppure direttamente la `Socket`, richiamando il metodo `close()`
- In tal caso il thread riceverà una `java.io.IOException`
- Per far questo, il thread principale deve tenersi memorizzati i riferimenti alle socket o agli stream

Interruzione thread

- Le operazioni di lettura da stream non gestiscono le `InterruptedException`
- Quindi non serve richiamare `interrupt()` su un thread che legge da uno stream
- Si può però chiudere lo stream dal quale il thread legge, oppure direttamente la `Socket`, richiamando il metodo `close()`
- In tal caso il thread riceverà una `java.io.IOException`
- Per far questo, il thread principale deve tenersi memorizzati i riferimenti alle socket o agli stream
- Oppure il thread che si occupa del client deve implementare un metodo che chiude la socket, richiamabile dal thread principale

Serializzazione

- Scrivere un oggetto in uno stream
- Leggere un oggetto da uno stream
- La classe dell'oggetto deve implementare l'interfaccia `java.io.Serializable`
 - Questa interfaccia NON contiene metodi
- Usare stream
 - `ObjectOutputStream` (metodo `writeObject()`)
 - `ObjectInputStream` (metodo `readObject()`)
- La maggior parte delle classi della libreria standard implementano questa interfaccia

Date server

```
import java.io.*;
import java.net.*;
import java.util.Date;

public class DateServer {
    public DateServer(int port) throws IOException {
        ServerSocket serverSocket = new ServerSocket(port);
        System.out.println("in attesa di connessioni...");
        Socket socket = serverSocket.accept();
        ObjectOutputStream objectOutputStream =
            new ObjectOutputStream(socket.getOutputStream());
        objectOutputStream.writeObject(new Date());
        objectOutputStream.flush();
        System.out.println("server terminato");
        socket.close();
    }
}
```

Date client

Utilizzare telnet in questo caso non va bene perché si riceve la rappresentazione di un oggetto Java.

```
import java.io.*;
import java.net.*;
import java.util.Date;

public class DateClient {
    public DateClient(String host, int port)
        throws UnknownHostException, IOException, ClassNotFoundException {
        Socket socket = new Socket(host, port);
        ObjectInputStream objectInputStream =
            new ObjectInputStream(socket.getInputStream());
        Date date = (Date)objectInputStream.readObject();
        System.out.println(date.toString());
        socket.close();
    }
}
```

RMI (Remote Method Invocation)

- Implementa l'RPC (Remote Procedure Call) nel contesto object-oriented:

RMI (Remote Method Invocation)

- Implementa l'RPC (Remote Procedure Call) nel contesto object-oriented:
 - Invece di una procedura remota si può invocare un metodo di un oggetto remoto

RMI (Remote Method Invocation)

- Implementa l'RPC (Remote Procedure Call) nel contesto object-oriented:
 - Invece di una procedura remota si può invocare un metodo di un oggetto remoto
- Il client può invocare un metodo su un oggetto remoto trasparentemente

RMI (Remote Method Invocation)

- Implementa l'RPC (Remote Procedure Call) nel contesto object-oriented:
 - Invece di una procedura remota si può invocare un metodo di un oggetto remoto
- Il client può invocare un metodo su un oggetto remoto trasparentemente
 - Allo stesso modo dell'invocazione su un oggetto locale

RMI (Remote Method Invocation)

- Implementa l'RPC (Remote Procedure Call) nel contesto object-oriented:
 - Invece di una procedura remota si può invocare un metodo di un oggetto remoto
- Il client può invocare un metodo su un oggetto remoto trasparentemente
 - Allo stesso modo dell'invocazione su un oggetto locale
 - Tutte le procedure per la realizzazione dell'RMI sono gestite automaticamente da Java

RMI (Remote Method Invocation)

- Implementa l'RPC (Remote Procedure Call) nel contesto object-oriented:
 - Invece di una procedura remota si può invocare un metodo di un oggetto remoto
- Il client può invocare un metodo su un oggetto remoto trasparentemente
 - Allo stesso modo dell'invocazione su un oggetto locale
 - Tutte le procedure per la realizzazione dell'RMI sono gestite automaticamente da Java
- l'RMI si affida alla serializzazione per il passaggio dei parametri e per la restituzione del risultato.

Architettura dell'RMI

L'architettura dell'RMI è basata su alcuni layer

- 1 Lo *Stub* è la simulazione locale dell'oggetto remoto

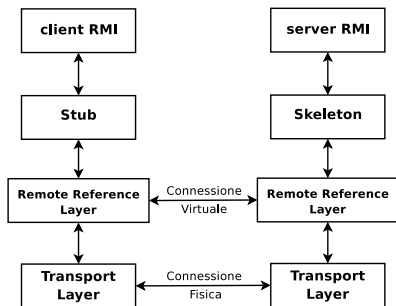


Figure: RMI

Registro di nomi: rmiregistry

- Gli oggetti i cui metodi possono essere richiamati da remoto devono essere registrati presso un **rmiregistry** che è in esecuzione sulla stessa macchina del server (di default utilizza la porta 1099)

Registro di nomi: rmiregistry

- Gli oggetti i cui metodi possono essere richiamati da remoto devono essere registrati presso un **rmiregistry** che è in esecuzione sulla stessa macchina del server (di default utilizza la porta 1099)
 - Operazioni
 - `Naming.bind("nome", oggetto)`
 - `Naming.rebind("nome", oggetto)`

Registro di nomi: rmiregistry

- Gli oggetti i cui metodi possono essere richiamati da remoto devono essere registrati presso un **rmiregistry** che è in esecuzione sulla stessa macchina del server (di default utilizza la porta 1099)
 - Operazioni
 - `Naming.bind("nome", oggetto)`
 - `Naming.rebind("nome", oggetto)`
- I client ottengono un riferimento ad un oggetto remoto interrogando un rmiregistry remoto:

Registro di nomi: rmiregistry

- Gli oggetti i cui metodi possono essere richiamati da remoto devono essere registrati presso un **rmiregistry** che è in esecuzione sulla stessa macchina del server (di default utilizza la porta 1099)
 - Operazioni
 - `Naming.bind("nome", oggetto)`
 - `Naming.rebind("nome", oggetto)`
- I client ottengono un riferimento ad un oggetto remoto interrogando un rmiregistry remoto:
 - `Remote remote = Naming.lookup("nome")`

Skeleton & Stub

- Vengono creati automaticamente utilizzando il compilatore `rmic`

Skeleton & Stub

- Vengono creati automaticamente utilizzando il compilatore `rmic`
- Gli si deve passare il nome completo di una classe

Skeleton & Stub

- Vengono creati automaticamente utilizzando il compilatore `rmic`
- Gli si deve passare il nome completo di una classe
- Se gli si passa la classe C, genera i file:

Skeleton & Stub

- Vengono creati automaticamente utilizzando il compilatore `rmic`
- Gli si deve passare il nome completo di una classe
- Se gli si passa la classe C, genera i file:
 - `C_Skel.class`

Skeleton & Stub

- Vengono creati automaticamente utilizzando il compilatore `rmic`
- Gli si deve passare il nome completo di una classe
- Se gli si passa la classe C, genera i file:
 - `C_Skel.class`
 - `C_Stub.class`

Skeleton & Stub

- Vengono creati automaticamente utilizzando il compilatore `rmic`
- Gli si deve passare il nome completo di una classe
- Se gli si passa la classe C, genera i file:
 - `C_Skel.class`
 - `C_Stub.class`
- Con la versione 1.2 del protocollo RMI (di default nel jdk 1.5) lo skeleton non è più necessario

Skeleton & Stub

- Vengono creati automaticamente utilizzando il compilatore `rmic`
- Gli si deve passare il nome completo di una classe
- Se gli si passa la classe C, genera i file:
 - `C_Skel.class`
 - `C_Stub.class`
- Con la versione 1.2 del protocollo RMI (di default nel jdk 1.5) lo skeleton non è più necessario
- Utilizzando l'opzione `-vcompat` verranno generati skeleton e stub compatibili con tutte le versioni del protocollo.

Lato server

- Dichiarare un'interfaccia (coi metodi che devono essere richiamabili da remoto) che estende `java.rmi.Remote`

Lato server

- Dichiarare un'interfaccia (coi metodi che devono essere richiamabili da remoto) che estende `java.rmi.Remote`
- È necessario che i metodi dichiarino di lanciare l'eccezione `java.rmi.RemoteException`

Lato server

- Dichiarare un'interfaccia (coi metodi che devono essere richiamabili da remoto) che estende `java.rmi.Remote`
- È necessario che i metodi dichiarino di lanciare l'eccezione `java.rmi.RemoteException`
- Definisce una classe che implementa tale interfaccia ed estende `java.rmi.server.UnicastRemoteObject`

Lato server

- Dichiarare un'interfaccia (coi metodi che devono essere richiamabili da remoto) che estende `java.rmi.Remote`
- È necessario che i metodi dichiarino di lanciare l'eccezione `java.rmi.RemoteException`
- Definisce una classe che implementa tale interfaccia ed estende `java.rmi.server.UnicastRemoteObject`
- Utilizza il compilatore `rmic` che genera stub e skeleton

Lato server

- Dichiarare un'interfaccia (coi metodi che devono essere richiamabili da remoto) che estende `java.rmi.Remote`
- È necessario che i metodi dichiarino di lanciare l'eccezione `java.rmi.RemoteException`
- Definisce una classe che implementa tale interfaccia ed estende `java.rmi.server.UnicastRemoteObject`
- Utilizza il compilatore `rmic` che genera stub e skeleton
- Crea un'istanza dell'oggetto e la registra presso il `rmiregistry`

Lato server

- Dichiarare un'interfaccia (coi metodi che devono essere richiamabili da remoto) che estende `java.rmi.Remote`
- È necessario che i metodi dichiarino di lanciare l'eccezione `java.rmi.RemoteException`
- Definisce una classe che implementa tale interfaccia ed estende `java.rmi.server.UnicastRemoteObject`
- Utilizza il compilatore `rmic` che genera stub e skeleton
- Crea un'istanza dell'oggetto e la registra presso il `rmiregistry`
 - Prima deve esser già stato lanciato l'`rmiregistry` e deve essere in grado di trovare le classi degli oggetti che vengono registrati

RemoteHello

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
import java.util.Date;  
  
public interface RemoteHello extends Remote {  
    public String sayHello(String name) throws RemoteException;  
    public Date getDate() throws RemoteException;  
}
```

RemoteHello

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
import java.util.Date;  
  
public interface RemoteHello extends Remote {  
    public String sayHello(String name) throws RemoteException;  
    public Date getDate() throws RemoteException;  
}
```

Serializzazione

Per spedire i parametri ed il risultato verrà utilizzata la serializzazione, quindi è necessario che le classi coinvolte siano serializzabili.

RemoteHelloImpl

```
public class RemoteHelloImpl extends UnicastRemoteObject
    implements RemoteHello {
    public String sayHello(String name) throws RemoteException {
        return "ciao " + name + "!";
    }

    public Date getDate() throws RemoteException {
        return new Date();
    }

    public static void main(String[] args) throws RemoteException,
        MalformedURLException, AlreadyBoundException {
        RemoteHello remoteHello = new RemoteHelloImpl();
        Naming.bind("remote_hello", remoteHello);
        System.out.println("oggetto registrato");
    }
}
```

Lato client

- Il client deve per prima cosa ottenere un riferimento all'oggetto remoto utilizzando `Naming.lookup(url)`

Lato client

- Il client deve per prima cosa ottenere un riferimento all'oggetto remoto utilizzando
`Naming.lookup(url)`
- L'URL sarà della forma
`"rmi://<server>/<nome oggetto>"`

Lato client

- Il client deve per prima cosa ottenere un riferimento all'oggetto remoto utilizzando
`Naming.lookup(url)`
- L'URL sarà della forma
`"rmi://<server>/<nome oggetto>"`
- Il riferimento dovrà esser convertito al tipo giusto

Lato client

- Il client deve per prima cosa ottenere un riferimento all'oggetto remoto utilizzando
`Naming.lookup(url)`
- L'URL sarà della forma
`"rmi://<server>/<nome oggetto>"`
- Il riferimento dovrà esser convertito al tipo giusto
- E potrà essere utilizzato per richiamare i metodi come si fa con un oggetto locale

RemoteHelloClient

```
public class RemoteHelloClient {  
    public static void main(String[] args) throws MalformedURLException,  
        RemoteException, NotBoundException {  
        RemoteHello remoteHello;  
        String url = "rmi://127.0.0.1/remote_hello";  
  
        remoteHello = (RemoteHello) Naming.lookup(url);  
        System.out.println(remoteHello.sayHello("cliente"));  
        System.out.println(remoteHello.getDate().toString());  
    }  
}
```