Programmazione di rete in Java

Reti di calcolatori

- Una rete di calcolatori è un sistema che permette la condivisione di dati informativi e risorse (sia hardware sia software) tra diversi calcolatori.

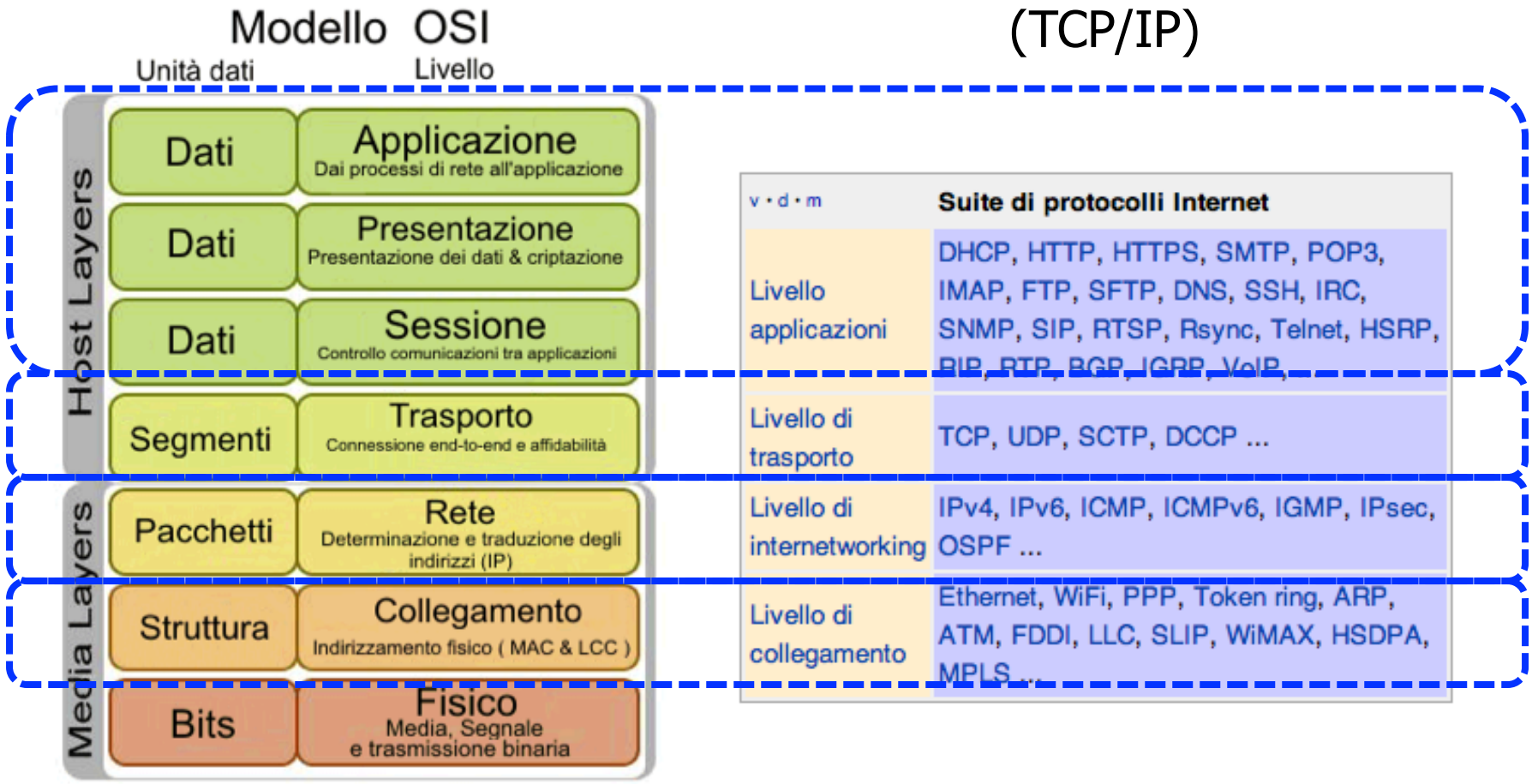
- Lo scopo è di fornire dei servizi aggiuntivi per gli utenti:
  - Condivisione di informazioni e risorse.

- Può essere privata o pubblica:
  - Internet è la più grande rete costituita da miliardi di dispositivi in tutto il mondo.

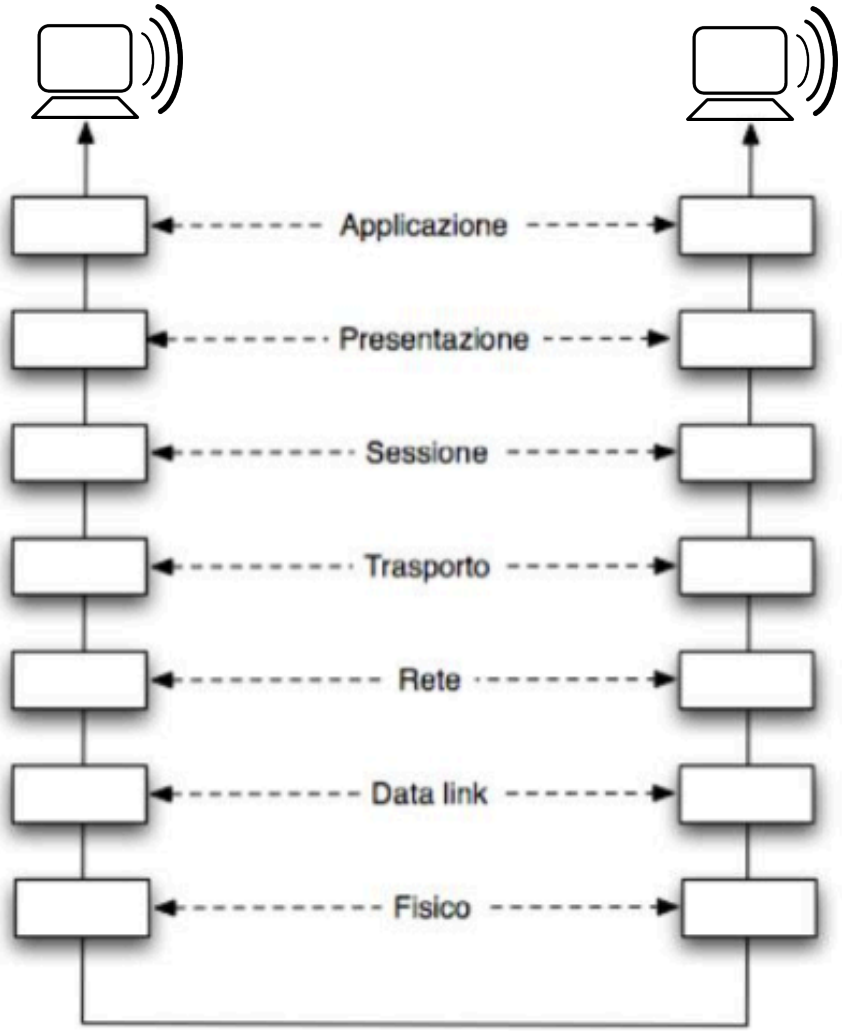
Comunicazione

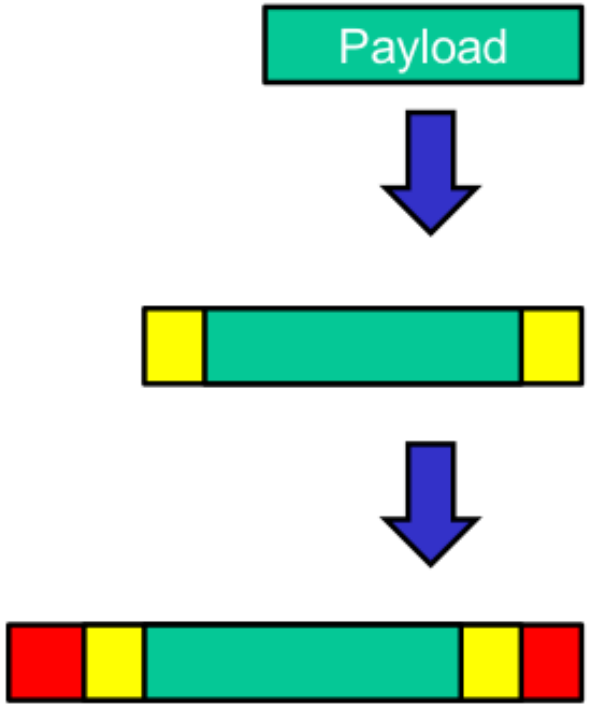
- All'interno di una rete di calcolatori la comunicazione avviene tramite scambio di informazioni (messaggi)
- Per comunicare sono necessari:
  - Un canale fisico;
  - Un linguaggio comune: protocollo di comunicazione.
- Canale fisico:
  - Cavo telefonico, fibra ottica, onde radio, ...
- Protocollo:
  - Insieme di regole formalmente descritte, definite al fine di favorire la comunicazione tra una o più entità.

Pila Protocollare



Comunicazione a livelli





Paradigmi di Comunicazione

- Client-Server
  - Applicazioni di rete formate da due programmi distinti che possono essere in esecuzione in due elaboratori (host) diversi: un server e un client.
  - **Server**: si mette in attesa di una richiesta da servire.
  - **Client**: effettua tale richiesta.
  - Tipicamente il client comunica con un solo server, mentre un server solitamente comunica con più client contemporaneamente.
- Peer-to-peer
  - Più host diversi comunicano tra loro comportandosi sia come client che come server.

Networking per le applicazioni

- Le applicazioni più diffuse per utilizzare le funzionalità di rete si appoggiano direttamente sui protocolli TCP o UDP della suite di protocolli Internet.
  - I livelli (protocolli) sottostanti sono mascherati.

- In alcuni casi si possono usare dei middleware:
  - Aggiungono un altro livello di comunicazione
  - Offrono delle funzionalità di alto livello
  - Esempio: RMI

TCP (Transfer Control Protocol)

- È un protocollo di alto livello:
  - È orientato alla connessione;
  - Esegue controlli sugli errori, congestione e flusso di comunicazione
- Un destinatario TCP è identificato da un indirizzo IP (Internet Protocol) e da una porta di destinazione.
  - L'indirizzo IP identifica la macchina alla quale vogliamo collegarci.
  - Il numero della porta identifica l'applicativo (servizio) al quale vogliamo connetterci.
- Esempio:
  - Un server web all'indirizzo 131.114.11.100 può offrire un servizio di posta elettronica e un servizio http.
  - Il primo risponde sulla porta 110 e il secondo alla porta 80.

Socket

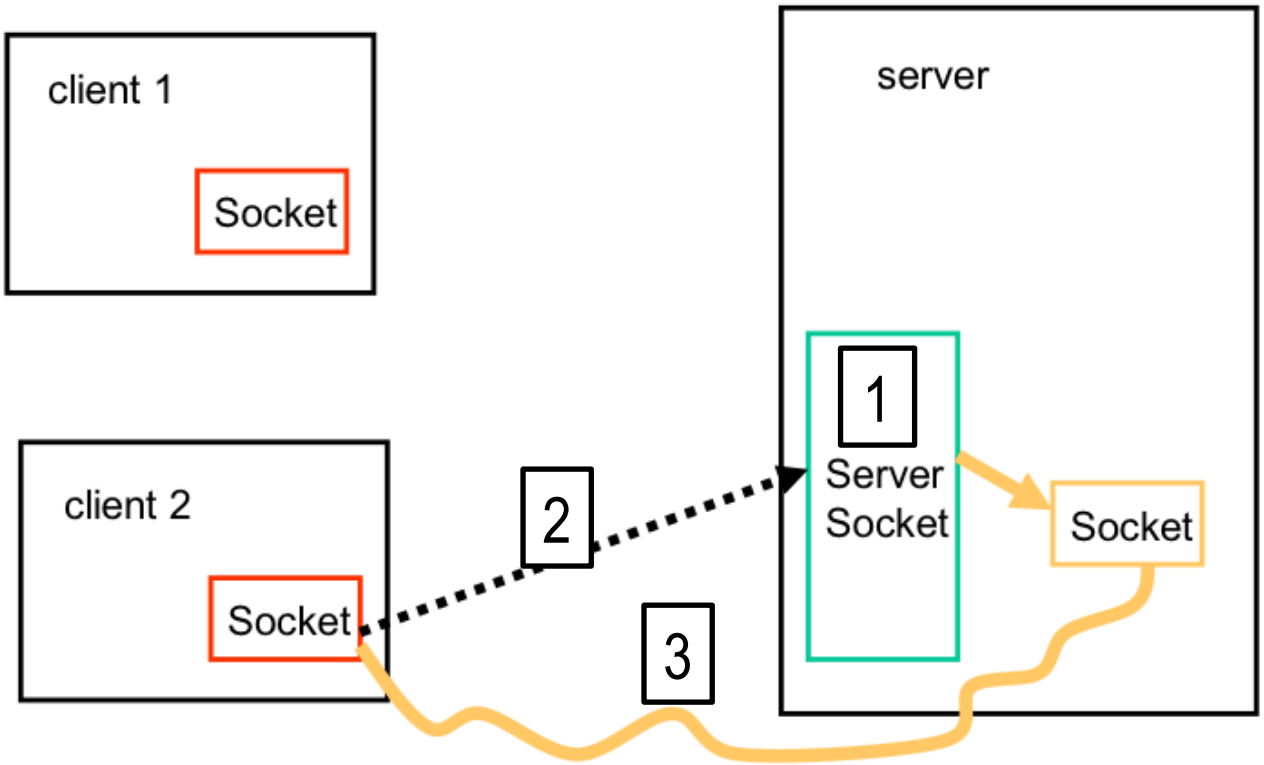
Socket (TCP)

- L'astrazione principale per la programmazione di rete è il **Socket**:
  - Identifica le risorse che permettono di stabilire un canale di comunicazione con un altro processo (eventualmente su un'altra macchina).
  - Tutto quello che viene trasmesso sono pacchetti TCP.

- I partecipanti di una comunicazione tramite socket sono individuati da:
  - indirizzo IP.
  - numero di porta.

- In Java si usa il package java.net
  - classi: **`Socket`**, **`ServerSocket`**

Comunicazione client-server



1. il ServerSocket del server si mette in attesa di una connessione
2. il socket del client si collega al server socket
3. viene creato un socket nel server e quindi stabilito un canale di comunicazione con i l client

Mettersi in attesa di una connessione (lato **Server**)

1. Creare un'istanza della classe **java.net.ServerSocket** specificando il numero di porta su cui rimanere in ascolto.
   - `ServerSocket serverSocket=new ServerSocket(4567);`
2. Chiamare il metodo **accept()** che fa in modo che il server rimanga in ascolto di una richiesta di connessione (la porta non deve essere già in uso)
   - `Socket socket=serverSocket.accept();`
3. Quando il metodo completa la sua esecuzione la connessione col client è stabilita e viene restituita un'istanza di **java.net.Socket** connessa al client remoto.

NB: su alcuni sistemi operativi sono necessari privilegi particolari per poter aprire porte con numeri inferiori a 1024.

Aprire una connessione (lato **Client**)

1. Aprire un socket specificando indirizzo IP e numero di porta del server.
   - `Socket socket = new Socket("127.0.0.1", 4567)`
2. All'indirizzo e numero di porta specificati ci deve essere in ascolto un processo server.
3. Se la connessione ha successo si usano (sia dal lato client che dal lato server) gli stream associati al socket per permettere la comunicazione tra client e server (e viceversa)
   - `Scanner in = new Scanner(socket.getInputStream());`
   - `PrintWriter out = new PrintWriter(socket.getOutputStream());`

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

Esempio Server: EchoServer

- Si crei un server che accetta connessioni TCP sulla porta **1337**.
- Una volta accettata la connessione il server leggerà ciò che viene scritto una riga alla volta e ripeterà nella stessa connessione ciò che è stato scritto.
- Se il server riceve una riga "**quit**" chiuderà la connessione e terminerà la sua esecuzione.

Esempio Server: EchoServer

```
public void startServer() throws IOException {
    // apro una porta TCP
    serverSocket = new ServerSocket(port);
    System.out.println("Server socket ready on port: " + port);

    // resto in attesa di una connessione
    Socket socket = serverSocket.accept();
    System.out.println("Received client connection");

    // apro gli stream di input e output per leggere
    // e scrivere nella connessione appena ricevuta
    Scanner in = new Scanner(socket.getInputStream());
    PrintWriter out = new PrintWriter(socket.getOutputStream());

    // leggo e scrivo nella connessione finche' non
    // ricevo "quit"
    while (true) {
        String line = in.nextLine();
        if (line.equals("quit")) {
            break;
        } else {
            out.println("Received: " + line);
            out.flush();
        }
    }

    // chiudo gli stream e il socket
    System.out.println("Closing sockets");
    in.close();
    out.close();
    socket.close();
    serverSocket.close();
}
```

Esempio Client: LineClient

- Si crei un client che si colleghi, utilizzando il protocollo TCP, alla porta **1337** dell'indirizzo IP **127.0.0.1**.
- Una volta stabilita la connessione il client legge, una riga alla volta, dallo standard input e invia il testo digitato al server.
- Il client inoltre stampa sullo standard output le risposte ottenute dal server.
- Il client deve terminare quando il server chiude la connessione.

Esempio Client: LineClient

```
public void startClient() throws IOException {
        Socket socket = new Socket(ip, port);
        System.out.println("Connection established");
        Scanner socketIn = new Scanner(socket.getInputStream());
        PrintWriter socketOut = new PrintWriter(socket.getOutputStream());
        Scanner stdin = new Scanner(System.in);

        try {
            while (true) {
                String inputLine = stdin.nextLine();
                socketOut.println(inputLine);
                socketOut.flush();
                String socketLine = socketIn.nextLine();
                System.out.println(socketLine);
            }
        } catch(NoSuchElementException e) {
            System.out.println("Connection closed");
        } finally {
            stdin.close();
            socketIn.close();
            socketOut.close();
            socket.close();
        }
}
```

Avviare EchoServer e LineClient

- Per testare il Server e il Client bisogna creare due **main** in due classi diverse.
- Da Eclipse, avviare entrambi i main come due processi diversi attivi contemporaneamente.
- Tramite l'icona "console" di Eclipse è possibile navigare tra le due console dei due processi aperti:

```
public class EchoServer {
  private int port;
  private ServerSocket serverSocket;

  public EchoServer(int port) {
     this.port = port;
  }

  public static void main(String[] args) {
     EchoServer server = new EchoServer(1337);
     try {
        server.startServer();
     } catch (IOException e) {
        System.err.println(e.getMessage());
     }
  }
```

```
public class LineClient {
  private String ip;
  private int port;

  public LineClient(String ip, int port) {
     this.ip = ip;
     this.port = port;
  }

  public static void main(String[] args) {
     LineClient client = new
              LineClient("127.0.0.1", 1337);
     try {
        client.startClient();
     } catch (IOException e) {
        System.err.println(e.getMessage());
     }
  }
```

Servire più client contemporaneamente

- 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".

- Idea:
  - All'interno del processo Server far eseguire l'istruzione **accept()** in un nuovo thread.
  - In questo modo è possibile accettare più client contemporaneamente.

Server Multi-Thread

- Il thread principale del server crea l'istanza di ServerSocket ed esegue l'**accept()**.
- Appena l'**accept()** restituisce un socket viene creato un altro thread che si occuperà del client associato a quel socket.
- Contemporaneamente il thread principale torna ad eseguire l'**accept()** sulla stessa istanza di **ServerSocket**.
- Ogni thread si occuperà di un solo client, senza interferire con gli altri.

Esempio: EchoServer Multi-thread

- Si vuole modificare EchoServer in modo da poter gestire più client contemporaneamente.
- Procedimento:
  - Spostare la logica che gestisce la comunicazione con il client in una nuova classe **ClientHandler** che implementa **Runnable**.
  - La classe principale del server si occuperà solo di istanziare il **ServerSocket**, eseguire la **accept()** e di creare i thread necessari per gestire le connessioni accettate.
  - La classe **ClientHandler** si occuperà di gestire la comunicazione con il client associato al socket assegnato.

EchoServer Multi-thread (Server)

```
public class MultiEchoServer {
    private int port;

    public MultiEchoServer(int port) {
        this.port = port;
    }

    public void startServer() {
        ExecutorService executor = Executors.newCachedThreadPool();
        ServerSocket serverSocket;
        try {
            serverSocket = new ServerSocket(port);
        } catch (IOException e) {
            System.err.println(e.getMessage()); // porta non disponibile
            return;
        }
        System.out.println("Server ready");
        while (true) {
            try {
                Socket socket = serverSocket.accept();
                executor.submit(new EchoServerClientHandler(socket));
            } catch(IOException e) {
                break; // entrerei qui se serverSocket venisse chiuso
            }
        }
        executor.shutdown();
    }

    public static void main(String[] args) {
        MultiEchoServer echoServer = new MultiEchoServer(1337);
        echoServer.startServer();
    }
}
```

EchoServer Multi-thread (ClientHandler)

```
public class EchoServerClientHandler implements Runnable {

    private Socket socket;

    public EchoServerClientHandler(Socket socket) {
        this.socket = socket;
    }

    public void run() {
        try {
            Scanner in = new Scanner(socket.getInputStream());
            PrintWriter out = new PrintWriter(socket.getOutputStream());
            // leggo e scrivo nella connessione finche' non ricevo "quit"
            while (true) {
                String line = in.nextLine();
                if (line.equals("quit")) {
                    break;
                } else {
                    out.println("Received: " + line);
                    out.flush();
                }
            }
            // chiudo gli stream e il socket
            in.close();
            out.close();
            socket.close();
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Chiusura connessioni

- Per chiudere un **ServerSocket** o un **Socket** si utilizza il metodo **close()**.
- Nel caso di **ServerSocket**, l'invocazione di **close()** farà terminare immediatamente la **accept()** con una **IOException**.
- Nel caso di **Socket**, l'invocazione di **close()** farà terminare immediatamente le operazioni di lettura o scrittura del socket con eccezioni che dipendono dal tipo di reader/writer utilizzato.
- Sia **ServerSocket** sia **Socket** hanno un metodo **isClosed()** che restituisce vero se il socket è stato chiuso.

RMI (Remote Method Invocation)

RMI—Remote Method Invocation

- **RMI** è un middleware avente lo scopo di portare i vantaggi della programmazione orientata gli oggetti nel contesto distribuito.
- La tecnologia RMI introduce il concetto di **oggetto remoto**
  - Un oggetto remoto è un oggetto il cui riferimento è disponibile su una JVM diversa da quella in cui risiede l'oggetto.
- Possono essere invocati metodi su oggetti remoti, per i quali si ottiene un riferimento remoto
- Aspetti innovativi
  - passaggio dei parametri per valore (per oggetti NON REMOTI) e indirizzo (oggetti REMOTI)
  - possibilità di scaricare (download) automaticamente le classi necessarie per la valutazione remota

Architettura client-server usando RMI

- Caso tipico
  - Server crea oggetti remoti, li rende visibili e aspetta che i client invochino metodi su di essi
  - Client ottiene riferimenti a oggetti remoti e invoca metodi su di essi
- RMI fornisce il meccanismo con cui server e client comunicano per costituire l'applicazione distribuita

Funzionalità di RMI (1)

- Localizzazione di oggetti remoti
  - Gli oggetti remoti sono registrati presso un registro di nomi (**rmiregistry**) che fornisce una "naming facility", oppure
  - Le operazioni passano come parametri e/o restituiscono riferimenti a oggetti remoti
- Comunicazione con oggetti remoti
  - gestita da RMI, per i programmi accedere a oggetti remoti non fa differenza rispetto agli oggetti "normali"

Funzionalità di RMI (2)

- Dynamic class loading
  - essendo possibile passare oggetti ai metodi di oggetti remoti, oltre a trasmettere i valori dei parametri, RMI consente di trasferire il codice degli oggetti a run time (potrebbe trattarsi di un nuovo sottotipo)

  È un esempio di codice mobile

Definizioni

- *Oggetto remoto*: oggetto i cui metodi possono essere invocati da una *Java Virtual Machine* diversa da quella in cui l'oggetto risiede
- *Interfaccia remota*: interfaccia che dichiara quali sono i metodi che possono essere invocati da una diversa *Java Virtual Machine*
- *Server*: insieme di uno o più oggetti remoti che, implementando una o più interfacce remote, offrono delle risorse (dati e/o procedure) a macchine esterne distribuite sulla rete
- *Remote Method Invocation* (RMI): invocazione di un metodo presente in una interfaccia remota implementata da un oggetto remoto. La sintassi di una invocazione remota è identica a quella locale

Registry

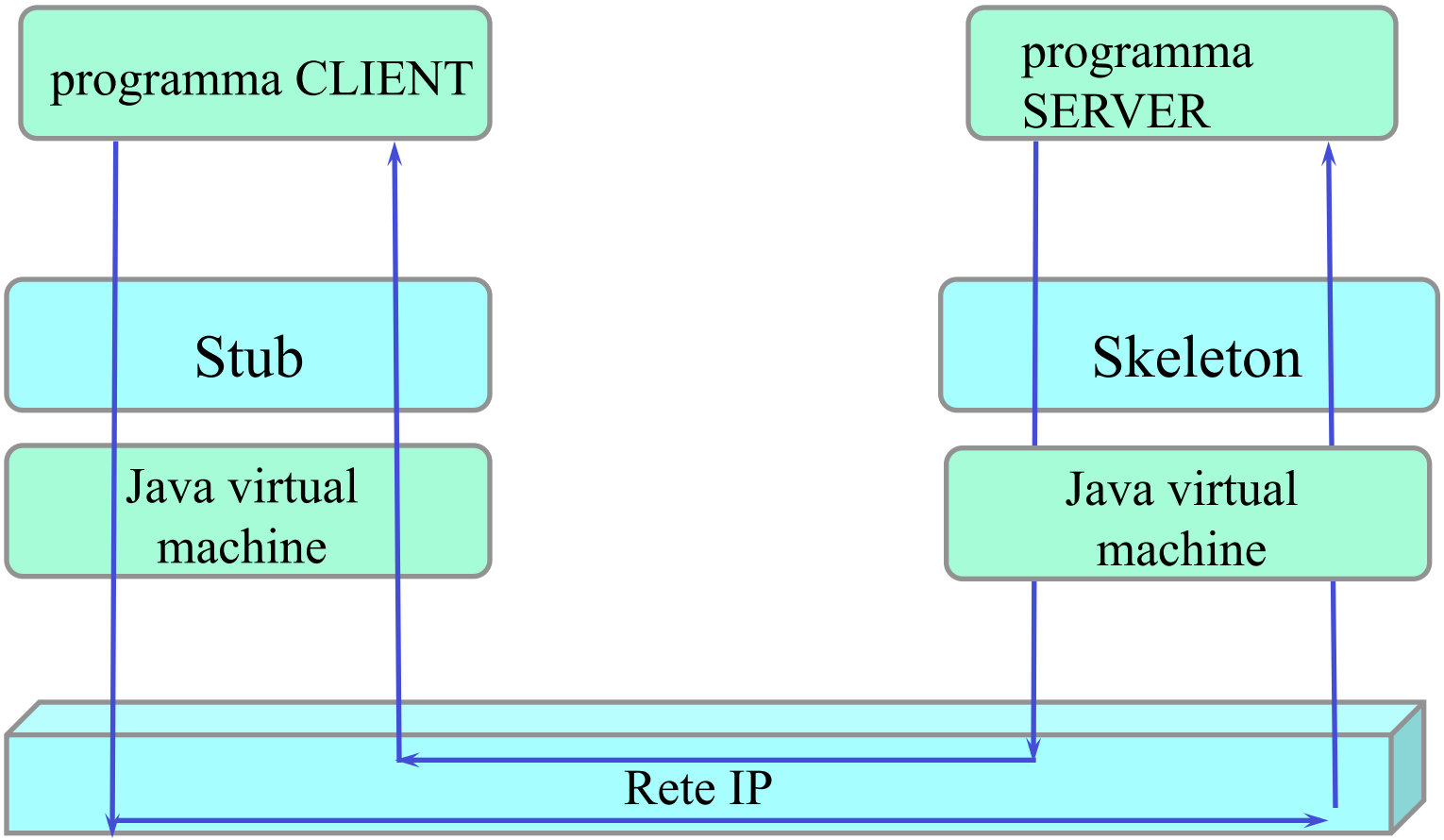
- È un programma che viene lanciato dalla finestra dei comandi.
  - **rmiregistry -J-Djava.rmi.server.hostname=0.0.0.0**
- È un programma di rete che ascolta su una porta, per default "1099".
- Occorre dargli comando per realizzare il binding tra l'oggetto locale esportato come remoto e il nome col quale ad esso si riferiscono i client.

Architettura “interna” (1)

- Il client colloquia con un proxy locale del server, detto **stub**
  - lo stub rappresenta il server sul lato client
  - implementa l'interfaccia del server
  - è capace di fare forward di chiamate di metodi
  - il client ha un riferimento locale all'oggetto stub

- Esiste anche un proxy del client sul lato server, detto **skeleton**
  - è una rappresentazione del client
  - chiama i servizi del server
  - sa come fare forward dei risultati
  - lo skeleton ha un riferimento all'oggetto

Architettura “interna” (2)



skeleton non necessario in una applicazione Java 2

Fasi di creazione di un sistema

1. Definire una INTERFACCIA remota per la comunicazione tra client e server
2. Definire un oggetto remoto che implementa l'interfaccia e un'applicazione server
   convenzione: stesso nome dell'interfaccia e suffisso **Impl**
3. Definire il client che usa un riferimento all'interfaccia remota per accedere all'oggetto remoto
4. Compilare ed eseguire

Creazione di interfaccia remota

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

public interface HelloServer extends Remote {
    String sayHello() throws RemoteException;
}
```

Un'interfaccia remota

1. deve essere pubblica
2. deve estendere **java.rmi.Remote**
3. ogni metodo deve dichiarare **java.rmi.RemoteException**
4. ogni oggetto remoto passato come parametro o valore di ritorno deve essere dichiarato di tipo interfaccia remota

Lato Server

1. Istanziare l'oggetto remoto **obj** che implementa l'interfaccia remota **IR**.
2. Creare l'oggetto **stub** usando la classe **java.rmi.server.UnicastRemoteObject**:
   - `IR stub = (IR) UnicastRemoteObject.exportObject(obj, 0);`
3. Registrare lo stub nel "**registry**"
   - l'oggetto **obj** potrebbe avere metodi che restituiscono riferimenti agli altri oggetti remoti, evitando così di dover passare attraverso il registry per accedere ad essi.
   - `Registry registry = LocateRegistry.getRegistry();`
   - `registry.rebind("NOME", stub);`

Quando si esegue il server bisogna specificare come parametro della JVM: -Djava.rmi.server.codebase=file:***classDir***/ dove **classDir** è la directory che contiene i file compilati (deve essere raggiungibile sia dal client che dal registry!). In alternativa bisogna rendere i file compilati disponibili nel **classpath** del client e del registry.

Lato Server: esempio

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;

    public class HelloServerImpl implements HelloServer {

        public String sayHello() {
            return "Hello, world!";
        }

        public static void main(String args[]) {

            try {
                HelloServerImpl helloServer = new HelloServerImpl();
                HelloServer helloServerStub = (HelloServer)
                                UnicastRemoteObject.exportObject(helloServer, 0);

                // Registra il riferimento remoto (stub) dell'helloServer nel
                // registry e lo identifica con la stringa "Hello".
                Registry registry = LocateRegistry.getRegistry();
                registry.rebind("Hello", helloServerStub);

                System.err.println("Server ready");
            } catch (Exception e) {
                System.err.println("Server exception: " + e.toString());
                e.printStackTrace();
            }
        }
}
```

Lato Client

1. Ottenee un riferimento al registry specificandone indirizzo IP:
   - `Registry registry = LocateRegistry.getRegistry(ip);`

2. Ottenere un riferimento all'oggetto **stub** usando il registry:
   - `IR stub = (IR) registry.lookup("NOME");`

Lato Client: esempio

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

  public static void main(String[] args) {

    final String REGISTRY_ADDRESS = "127.0.0.1";

    try {
        Registry registry = LocateRegistry.getRegistry(REGISTRY_ADDRESS);
        HelloServer helloServerStub = (HelloServer) registry.lookup("Hello");
        String response = helloServerStub.sayHello();
        System.out.println("response: " + response);

    } catch (Exception e) {
        System.err.println("Client exception: " + e.toString());
        e.printStackTrace();
    }
  }
}
```

Semantica Parametri

- Semantica dell'invocazione di un metodo remoto **m(obj)**
  - se l'oggetto **`obj`** è **remoto**, ed è stato esportato con **UnicastRemoteObject.exportObject(obj,0)**:
    - normale semantica
  - se no, **`m`** opera su una copia **serializzata** di **`obj`**; eventuali modifiche allo stato di **`obj`** hanno effetto solo sulla copia, non sull'originale
    - in pratica il passaggio di oggetti non remoti è per valore, mentre per gli oggetti remoti è per riferimento

Serializzazione

- E` la trasformazione di un oggetto in un byte stream
- E` possibile effettuarla su oggetti che implementano l'interfaccia **Serializable**
  - Oggetti passati a, o restituiti da, un oggetto remoto devono implementare l'interfaccia **Serializable**
  - se invece si passano riferimenti remoti, basta implementare **Remote**
- **Stubs** e **skeletons** effettuano automaticamente la serializzazione e deserializzazione
  - si dice che fanno il "*marshal*" (schieramento) e "*unmarshal*" di argomenti e risultati

Vita degli oggetti remoti

- Se **main()** termina nel server che ha registrato l'oggetto, l'oggetto remoto registrato continua ad esistere finchè **rmiregistry** resta in esecuzione.
- Ecco perché è necessario fare **rebind()** anziché **bind()** (potrebbe esistere un oggetto con lo stesso nome)
- Si può anche fare:
  - **`Naming.unbind() ("…");`**

Ricapitolando

1. Eseguire il comando:
   rmiregistry -J-Djava.rmi.server.hostname=0.0.0.0 nel server.
2. Eseguire il **server** passando alla JVM il seguente parametro aggiuntivo:
   -Djava.rmi.server.codebase=file:***classDir***/
   dove **classDir** è la directory che contiene i file compilati (deve essere accessibile sia dal **client** che dal **rmiregistry,** in alternativa si possono rendere i file compilati disponibili nel **classpath** del client e del registry usando la variabile d'ambiente **CLASSPATH**).
   - NB: esistono anche altri modi per mettere disponibile la codebase (ad esempio attraverso http, ftp, ecc).
3. Eseguire i **client** fornendo a ciascuno l'**indirizzo** della macchina che ha **rmiregistry** e il server in esecuzione.

Esercizio 1

- Si scriva un server che accetta connessioni via socket nella porta 4747.
- Il testo ricevuto da uno dei client deve essere visualizzato immediatamente a tutti gli altri.
- Se uno dei client digita “off” il server chiuderà la connessione a tutti i client, chiuderà il socket in attesa, e infine terminerà

Esercizio 2

- Si svolga l'esercizio 1 utilizzando RMI e aggiungendo la logica del client.
- Per svolgere questo esercizio bisogna ricordare che anche il client deve esportare un oggetto remoto per poter ricevere le notifiche dal server (che avrà bisogno di invocare metodi remoti del client).
- NB: il client non deve registrare lo stub del suo oggetto remoto nel registry.

Riferimenti

- Socket:
  - http://docs.oracle.com/javase/tutorial/networking/sockets/
- RMI:
  - http://docs.oracle.com/javase/tutorial/rmi/index.html