

20 APPLICAZIONI CON I SOCKET

PREREQUISITI

- Struttura del software di rete, TCP/IP
- Saper utilizzare una classe
- Conoscere i meccanismi dell'ereditarietà

OBIETTIVI

CONOSCENZE

- Architettura delle applicazioni client/server
- Classe *InetAddress*
- Concetti di porta e socket
- Classi *Socket* e *ServerSocket*

ABILITÀ/CAPACITÀ

- Realizzare applicazioni Java client/server con i socket, anche multithreading

20.1

APPLICAZIONI CLIENT/SERVER

Si chiamano **applicazioni di rete** le applicazioni che si collegano o che comunicano con computer diversi, collegati in rete. Si chiamano **applicazioni distribuite** le applicazioni la cui esecuzione è distribuita tra più computer; parti dell'applicazione si trovano su computer diversi e lavorano insieme.

Una forma molto nota di applicazioni distribuite sono le **applicazioni client/server**.

Un'applicazione client/server è un'applicazione distribuita divisa in due parti: un'applicazione **server** (back-end) che offre dei servizi e una applicazione **client** (front-end) che gestisce l'interfaccia con l'utente e permette di richiedere servizi al server.

Figura 20.1
Il modello
client/server



Le componenti client e server dell'applicazione in genere si trovano su due stazioni diverse (la parte server di solito si trova su un computer molto più potente) ma possono trovarsi anche nella stessa stazione.

L'architettura client/server può essere realizzata anche a **tre livelli**, con un livello di gestione dell'interfaccia utente, uno di elaborazione e uno di gestione delle informazioni (normalmente memorizzate in un database).

È un modello non simmetrico, chiamato anche **request/replay** perché si basa sul susseguirsi di richieste e risposte.

Il server può essere progettato in modo da soddisfare una sola richiesta per volta o più richieste contemporaneamente.

Il client per contattare il server ne deve conoscere l'**indirizzo**.

Per la comunicazione in rete viene usato il protocollo **TCP/IP** (Transmission Control Protocol/Internet Protocol).

Ogni computer è identificato da un numero, chiamato **indirizzo IP**.

L'indirizzo IP comunemente è rappresentato nella notazione decimale puntata cioè da 4 numeri (compresi tra 0 e 255) separati da punti, per esempio 132.45.37.1.

LA CLASSE INETADDRESS

La classe **InetAddress** mette a disposizione metodi per la gestione degli indirizzi IP e dei nomi delle stazioni.

Si può creare un oggetto *InetAddress* usando i metodi statici:

`static InetAddress getLocalHost()` restituisce nome e indirizzo della stazione locale; se la stazione non ha indirizzo o è protetta da firewall l'indirizzo è quello di loopback;
`static InetAddress getByName(String host)` determina l'indirizzo IP di una stazione, dato il nome; se il parametro è *null* dà l'indirizzo di default della stazione locale;

Metodi utili sono:

<code>String getHostName()</code>	dà il nome della stazione rappresentata dall'oggetto <i>InetAddress</i> ;
<code>String getHostAddress()</code>	dà l'indirizzo IP della stazione rappresentata dall'oggetto <i>InetAddress</i> .

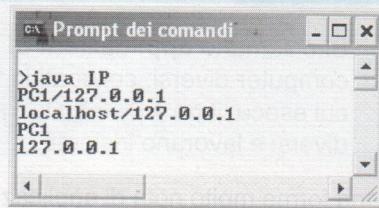
ESEMPIO 20.1

Uso di `InetAddress`.

```
import java.net.*;

public class IP {
    public static void main(String[] args) {
        try {
            System.out.println(InetAddress.getLocalHost());
            System.out.println(InetAddress.getByName(null));
            System.out.println(InetAddress.getLocalHost().getHostName());
            System.out.println(InetAddress.getLocalHost().getHostAddress());
        }
        catch (UnknownHostException e) {System.out.println(e);}
    }
}
```

Figura 20.2
Esecuzione
dell'esempio



20.2 I SOCKET

La tecnica più comune per la realizzazione di applicazioni client/server è l'utilizzo dei socket TCP/IP.

Un socket rappresenta un punto di connessione di una rete TCP/IP; è un punto a cui collegarsi per comunicare con altre applicazioni sulla rete (è analogo a una presa elettrica con cui ci si collega alla rete elettrica).

In pratica un socket identifica un computer e un processo sul computer; è formato da un **indirizzo IP** che identifica un **computer** sulla rete e da un numero di **porta TCP** che identifica un processo o un'**applicazione** sul computer.

A ogni processo o applicazione che usa TCP (o UDP) come protocollo di trasporto viene assegnato un numero identificativo unico chiamato porta. Le porte sono numerate da 0 a 65.535. Su uno stesso computer possono essere attive contemporaneamente più comunicazioni; ognuna è individuata da una porta diversa.

Il socket rappresenta solo uno dei capi di una comunicazione; due computer per comunicare usano ciascuno un socket; tra i **due socket** si crea una **connessione** per il trasferimento dei dati in due direzioni.

Dato che le connessioni sono identificate da entrambi i socket, un socket può essere usato da più di una connessione contemporaneamente, cioè due o più connessioni possono terminare nello stesso socket; per esempio due client possono collegarsi allo stesso server sul suo socket; i client potrebbero usare anche lo stesso numero di porta senza problemi perché comunque la coppia di socket che identifica la connessione è diversa.

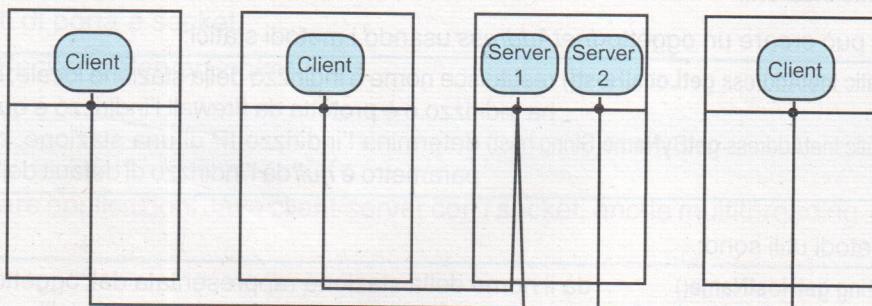


Figura 20.3
Esempi di
connessioni

In una applicazione client/server il **server** apre un socket e resta in attesa di connessioni (**apertura passiva**; la porta si dice in stato di ascolto); il client chiama il socket del server per iniziare la connessione (**apertura attiva**; permette di stabilire la connessione); per potersi connettere il client deve conoscere l'indirizzo e il numero di porta del server di destinazione.

Le applicazioni **server** più diffuse (come i server Web, FTP ecc.) hanno dei **numeri di porta predefiniti** compresi tra 0 e 1.023 assegnati dalla Internet Assigned Numbers Authority (IANA), noti come Well Known Port Number.

Per le applicazioni **client** i numeri di porta di solito sono assegnati **dinamicamente** dal sistema operativo alla richiesta di connessione al server; può essere utilizzato il primo numero libero a partire da 1.024.

Per **creare un'applicazione** con i socket basta utilizzare le **primitive** messe a disposizione dal TCP/IP per richiedere i servizi del protocollo TCP o del protocollo UDP. Il programmatore non si deve preoccupare di nessuna delle problematiche che riguardano il trasporto dei dati; deve solo scegliere quale protocollo di trasporto usare.

CREAZIONE DI APPLICAZIONI CON I SOCKET IN JAVA

Java

CREAZIONE DI UNA APPLICAZIONE CLIENT/SERVER CHE USA TCP COME PROTOCOLLO DI COMUNICAZIONE

20.2

Per creare una applicazione client/server che usa **TCP** si usano le classi *Socket* e *ServerSocket* del package *java.net*.

Per creare un'**applicazione client** bisogna:

- **creare un socket** indicando l'indirizzo (nome o indirizzo IP) del server e la porta su cui il server è in ascolto;

Socket(String host, int port) crea un socket specificando il sever come stringa e il numero di porta;
Socket(InetAddress address, int port) crea un socket specificando il sever come oggetto *InetAddress* e il numero di porta;

```
Socket client = new Socket("127.0.0.1", 1000);
```

- usare i metodi ***getInputStream()*** e ***getOutputStream()*** della classe *Socket* per ottenere dal socket un *InputStream* per ricevere dati e un *OutputStream* per inviare dati;

```
Scanner inStream = new Scanner(client.getInputStream());
```

```
PrintStream outStream = new PrintStream(client.getOutputStream());
```

- usare i metodi degli stream per gestire la comunicazione;

```
String linea = inStream.nextLine();
```

```
String messaggio="testo del messaggio";
outStream.println(messaggio);
```

- al termine chiudere il socket (dopo aver chiuso gli stream).

```
client.close();
```

Per creare un'**applicazione server** bisogna:

- creare una istanza della classe ***ServerSocket*** indicando la porta su cui il server è in ascolto;

```
ServerSocket server = new ServerSocket(1000);
```

- usare il metodo ***accept()*** della classe *ServerSocket* per bloccare il programma in attesa di una richiesta di connessione; quando un client cerca di connettersi il metodo restituisce il *socket* usato per la connessione dal lato del server;

```
Socket socket = server.accept();
```

- procedere come per il client:

- usare i metodi *getInputStream()* e *getOutputStream()* della classe *Socket* per ottenere dal socket un *InputStream* per ricevere dati e un *OutputStream* per inviare dati;
- usare i metodi degli stream per gestire la comunicazione;
- al termine chiudere il socket (dopo aver chiuso gli stream).

ESEMPIO 20.2

Realizzazione di una chat per lo scambio di messaggi tra due utenti.

Bisogna avviare prima il server e poi il client, specificando nel client l'indirizzo del server a cui collegarsi.

Se server e client si avviano sullo stesso computer, come indirizzo si può usare 127.0.0.1 o *localhost*.

Nota

I due utenti devono inviare un messaggio alternativamente (deve iniziare il client). Sia il server che il client possono chiudere la comunicazione inserendo il messaggio "fine".

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

public class Server {
    private ServerSocket server;
    private Socket connessione;
    private Scanner dalClient;
    private PrintStream alClient;

    public Server() {
        try {
            server = new ServerSocket(1000);
            System.out.println("Server attivo");
            connessione = server.accept();
            dalClient= new Scanner(connessione.getInputStream());
            alClient = new PrintStream(connessione.getOutputStream());
        } catch (IOException e) {
            System.out.println(e);
        }
    }

    public void conversazione() {
        alClient.println("Salve");
        Scanner tastiera = new Scanner(System.in);
        String messaggio = " ";
        try {
            while (! messaggio.equals("fine")) {
                messaggio = dalClient.nextLine();
                alClient.println(messaggio);
            }
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

```

        System.out.println(messaggio);
        if (! messaggio.equals("fine")) {
            messaggio = tastiera.nextLine();
            alClient.println(messaggio);
        }
    }
    connessione.close();
}
catch (IOException e) {
    System.out.println("Conversazione interrotta");
}
}

public class ProvaServer {
    public static void main(String[] args) {
        Server server = new Server();
        server.conversazione();
    }
}

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

public class Client {
    private Socket connessione;
    private Scanner dalServer;
    private PrintStream alServer;

    public Client() {
        Scanner tastiera = new Scanner(System.in);
        System.out.println("Inserire l'indirizzo del server");
        String indirizzo = tastiera.next();
        try{
            connessione = new Socket(indirizzo,1000);
            dalServer = new Scanner(connessione.getInputStream());
            alServer = new PrintStream(connessione.getOutputStream());
        }
        catch (IOException e) {
            System.out.println(e);
        }
    }

    public void conversazione() {
        String messaggio = " ";
        Scanner tastiera = new Scanner(System.in);
        try {
            while (! messaggio.equals("fine")) {
                messaggio = dalServer.nextLine();
                System.out.println(messaggio);
                if (! messaggio.equals("fine")) {
                    messaggio = tastiera.nextLine();
                }
            }
        }
    }
}

```

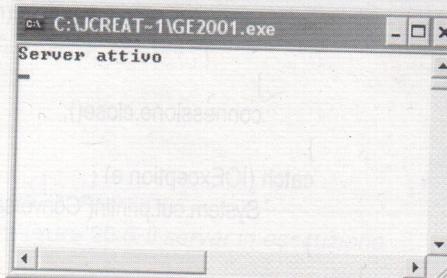


Figura 20.4 Avvio del server

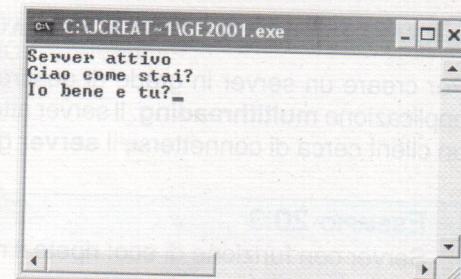
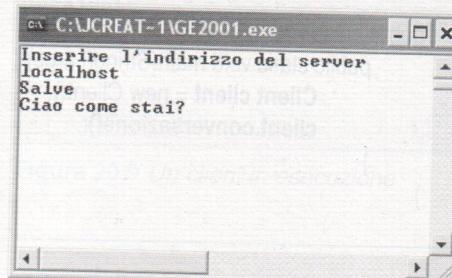


Figura 20.5 Un client si collega al server e inizia un dialogo

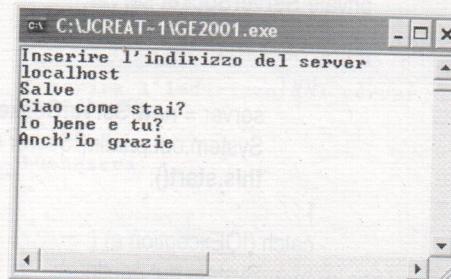


Figura 20.6 Prosecuzione del dialogo tra il client e il server

```

        alServer.println(messaggio);
    }
}
connessione.close();
}
catch (IOException e) {
    System.out.println("Conversazione interrotta");
}
}

public class ProvaClient {
    public static void main(String[] args) {
        Client client = new Client();
        client.conversazione();
    }
}

```

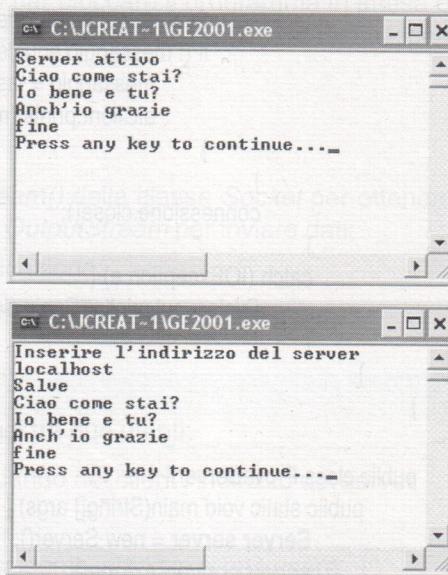


Figura 20.7
Proseguimento
del dialogo
tra il client e
il server

CREAZIONE DI UN SERVER CHE ACCETTA CONNESSIONI MULTIPLE

Per creare un server in grado di gestire più client contemporaneamente bisogna creare una applicazione **multithreading**. Il server attende che arrivino delle richieste di connessione; quando un client cerca di connettersi, il **server** genera un **thread** per gestire quel **client**.

ESEMPIO 20.3

Server con funzione di eco: ripete il messaggio che riceve dal client.

Bisogna avviare prima il server, poi si possono avviare più client, che possono inviare messaggi contemporaneamente.

La classe *Client* usata per inviare i messaggi è quella dell'esempio precedente.

Anche quando si chiudono tutti i client, il server resta attivo; bisogna chiuderlo manualmente (per esempio con *Ctrl-C*).

Nota

```

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

public class Server extends Thread {
    private ServerSocket server;

    public Server() {
        try {
            server = new ServerSocket(1000);
            System.out.println("Server attivo");
            this.start();
        }
        catch (IOException e) {
            System.out.println(e);
        }
    }
}

```

```

public void run() {
    try {
        while(true) {
            Socket richiestaClient = server.accept();
            new Connessione(richiestaClient);
        }
    } catch (IOException e) {
        System.out.println(e);
    }
}

class Connessione extends Thread {
    private Socket connessione;
    private Scanner dalClient;
    private PrintStream alClient;

    public Connessione (Socket richiestaClient) {
        try {
            connessione = richiestaClient;
            dalClient= new Scanner(connessione.getInputStream());
            alClient = new PrintStream(connessione.getOutputStream());
            this.start();
        } catch (IOException e) {
            System.out.println(e);
        }
    }

    public void run() {
        alClient.println("Salve");
        String messaggio = " ";
        try{
            while (! messaggio.equals("fine")) {
                messaggio = dalClient.nextLine();
                alClient.println(messaggio);
            }
            connessione.close();
        } catch (IOException e) {
            System.out.println("Connessione interrotta");
        }
    }
}

public class ProvaServer {
    public static void main(String[] args) {
        Server server = new Server();
    }
}

```

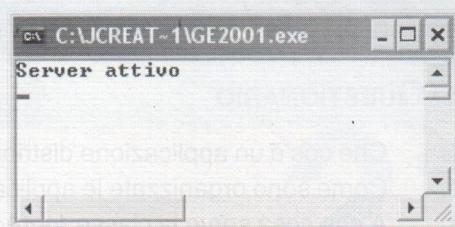


Figura 20.8 Il server in esecuzione

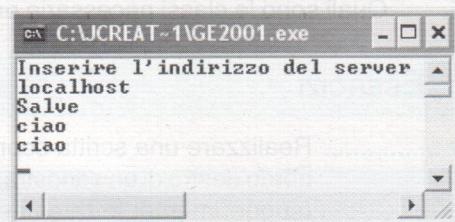


Figura 20.9 Un client in esecuzione

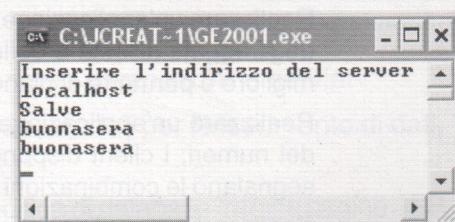
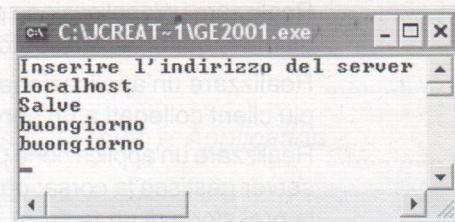


Figura 20.10 Altri due client in esecuzione