

ESERCITAZIONI DI LABORATORIO 1

JAVA SOCKET

Java socket: caratteristiche della comunicazione

Si è detto che un **socket** è come una porta di comunicazione e tutto ciò che è in grado di comunicare tramite il protocollo standard **TCP/IP** può collegarsi a un **socket** e comunicare attraverso di esso; sappiamo anche che le informazioni “spedite” da un **socket** a un altro prendono il nome di pacchetti **TCP/IP**.

Riassumiamo le caratteristiche fondamentali della comunicazione:

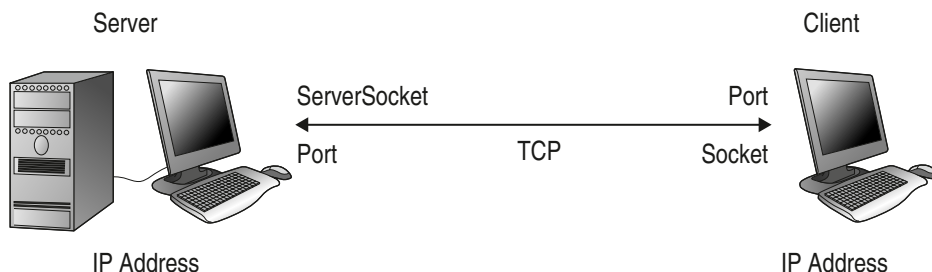
- ▶ tipicamente, la comunicazione è punto-punto (**unicast**), ma esistono estensioni che permettono comunicazioni multi-punto (**multicast**);
- ▶ la sorgente della comunicazione deve conoscere l'identità (indirizzo **IP** e numero di porta) del destinatario;
- ▶ la **serializzazione** (trasformazione tra dati strutturati e sequenze di byte) e il **marshalling** (conversione tra rappresentazioni di dati diversi) sono a carico delle applicazioni.

◀ **Marshalling** is the process of transforming the memory representation of an object to a data format suitable for storage or transmission, and it is typically used when data must be moved between different parts of a computer program or from one program to another (Wikipedia). ▶



Le operazioni necessarie per un trasferimento di dati tra due host sono le seguenti:

- 1 un host che si comporta da **server** apre un canale di comunicazione su una determinata porta e rimane in ascolto, in attesa di una richiesta di connessione (**ServerSocket**);
- 2 un host **client** fa una richiesta di connessione a un **server** (con indirizzo **IP** e address port conosciuti: **socket** sul **client**);
- 3 il **server** accetta la connessione del **client** e così viene instaurato un canale di comunicazione tra i due host.



Java utilizza la classe `Socket` per la creazione degli oggetti che permettono di utilizzare i `socket` e quindi di stabilire un canale di comunicazione tra un `client` e un `server` attraverso il quale si comunica utilizzando due stream particolari, specializzati per i `socket`, uno per l'input e l'altro per l'output.

Vediamo dapprima singolarmente le classi che Java mette a disposizione per l'utilizzo dei `socket` e successivamente implementiamo un `server` e un `client`:

- classe `InetAddress`;
- classe `ServerSocket`;
- classe `Socket`.

Classe `InetAddress`

Un indirizzo di rete **IP v4** è costituito da 4 numeri (da 0 a 255) separati ciascuno da un punto e il **DNS** permette di utilizzare in alternativa il nome simbolico per individuare un particolare host. La classe `InetAddress` mette a disposizione diversi metodi per astrarre dal particolare tipo di indirizzo specificato (a numeri o a lettere), occupandosi essa stessa di effettuare le dovute traduzioni.

Il diagramma completo della classe è il seguente:

Classe <code>InetAddress</code>
Metodi modificatori
<pre> InetAddress[] getAllByName(String host) InetAddress getByAddress(byte[] addr) InetAddress getByAddress(String host, byte[] addr) InetAddress getByName(String host) InetAddress getLocalHost()</pre>
<pre> byte[] getAddress() String getCanonicalHostName() String.getHostAddress() String.getHostByAddr(byte[] addr) boolean isAnyLocalAddress() boolean isLinkLocalAddress() boolean isLoopbackAddress() boolean isMCGlobal() boolean isMCLinkLocal() boolean isMCNodeLocal() boolean isMCOrgLocal() boolean isMCSiteLocal() boolean isMulticastAddress() boolean isSiteLocalAddress()</pre>
<pre> boolean equals(Object obj) int hashCode() String toString()</pre>
<pre> byte[][] lookupAllHostAddr(String host) Object run()</pre>

Non sono previsti costruttori e l'unico modo per creare un oggetto di classe `InetAddress` prevede l'utilizzo di *metodi statici*, e in particolare:

- `public static InetAddress getByName(String host)`
restituisce un oggetto `InetAddress` rappresentante l'host specificato nel parametro `host`; l'host può essere specificato sia col nome sia con l'indirizzo numerico e se si specifica `null` come parametro, ci si riferisce all'indirizzo di default della macchina locale;

```
public static InetAddress getLocalHost()
```

viene restituito un `InetAddress` corrispondente alla macchina locale; se tale macchina non è registrata, oppure è protetta da un firewall, l'indirizzo è quello di **loopback**: 127.0.0.1.

Se l'indirizzo specificato non può essere risolto tramite il **DNS**; questi metodi possono sollevare l'eccezione **UnknownHostException**.

Di particolare utilità sono i tre metodi illustrati di seguito:

```
1 public String getHostName()
```

restituisce il nome **dell'host** che corrisponde all'indirizzo **IP** dell'`InetAddress`; se il nome non è ancora noto (per esempio se l'oggetto è stato creato specificando un indirizzo **IP** numerico), verrà cercato tramite il **DNS**; se tale ricerca fallisce, verrà restituito l'indirizzo **IP** numerico sotto forma di stringa.

```
2 public String getHostAddress()
```

simile al precedente, restituisce però l'indirizzo **IP** numerico, sotto forma di stringa, corrispondente all'oggetto `InetAddress`.

```
3 public byte[] getAddress()
```

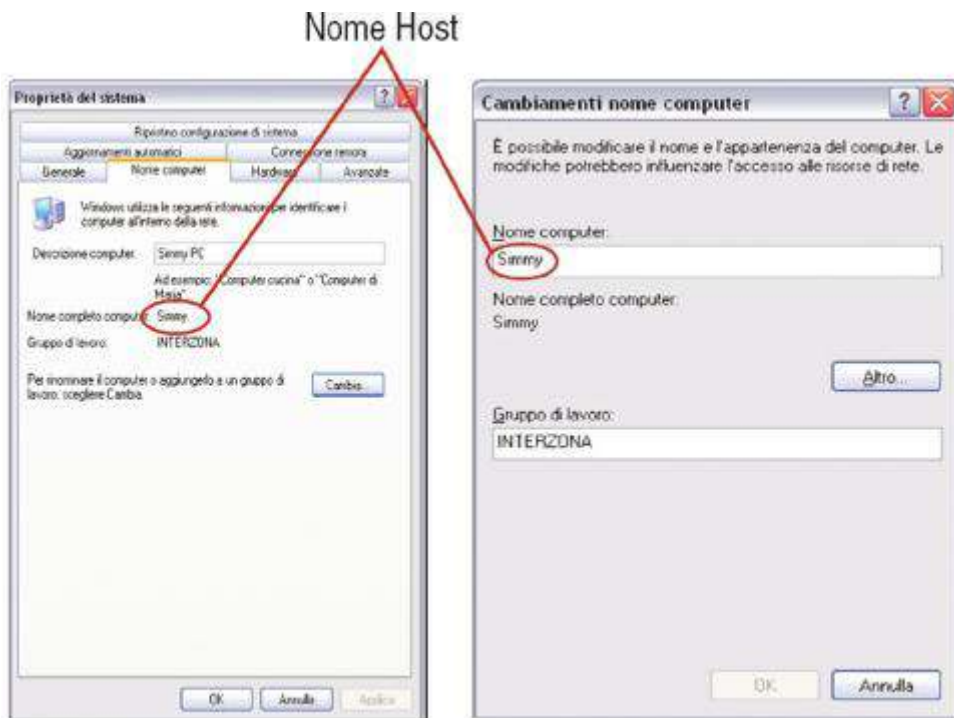
l'indirizzo **IP** numerico restituito sarà sotto forma di matrice di byte; l'ordinamento dei byte è **high byte first** (che è l'ordinamento tipico della rete).

Per esempio, con:

```
String indirizzo = InetAddress.getLocalHost().getHostAddress();
```

l'oggetto *indirizzo* conterrà l'indirizzo **IP** della macchina locale.

La figura che segue rappresenta un esempio del nome dell'host riconosciuto dalla classe `InetAddress` e dal **DNS**.



Classe ServerSocket

La classe `ServerSocket`, il cui diagramma è riportato di seguito, viene utilizzata per accettare connessioni da parte del `client`.

Classe ServerSocket	
Metodi costruttori	Metodi modificatori
<code>ServerSocket()</code> <code>ServerSocket(int port)</code> <code>ServerSocket(int port, int backlog)</code> <code>ServerSocket(int port, int backlog, InetAddress bindAddr)</code>	<code>void setSpcketFactory(SocketImplFactory fac)</code> <code>ServerSocketChannel getChannel()</code> <code>InetAddress getInetAddress()</code> <code>int getLocalPort()</code> <code>SocketAddress getLocalSocketAddress()</code> <code>int get/setReceiveAddress()</code> <code>boolean get/setReuseAddress()</code> <code>int get/setSoTimeout()</code> <code>boolean isBound()</code> <code>boolean isClosed()</code> <code>String toString()</code> <code>Socket accept()</code> <code>void bind(SocketAddress endpoint)</code> <code>void bind(SocketAddress endpoint, int backlog)</code> <code>void close()</code> <code>void implAccept(Socket s)</code>

Questa classe deve essere istanziata passando come parametro il numero della porta su cui il `server` sarà in ascolto: `ServerSocket(int port)`. L'unico metodo realmente necessario è `accept()`: mediante tale metodo l'oggetto rimane in attesa di richiesta di connessioni da parte di un `client` sulla porta specificata nel costruttore.

Quando una richiesta di connessione va a buon fine viene creato il canale di collegamento e il metodo restituisce un oggetto `Socket` connesso con il `client`.

Per esempio:

```
// creo un server sulla porta 6789
ServerSocket server = new ServerSocket(6789);
// rimane in attesa di un client
Socket client = server.accept();
// chiudo il server per inibire altri client
server.close();
```

Vediamo nel dettaglio le singole istruzioni:

- il `server` si mette in ascolto sulla porta 6789;
- quando riceve una richiesta di connessione, viene creato un oggetto `client` di classe `Socket` (che vedremo in seguito); questo oggetto rappresenta il canale di comunicazione `server-client`;
- con la terza istruzione si chiude il `server`: ciò implica una comunicazione di tipo `Unicast`, dato che, dopo avere instaurato una connessione, il `server` non rimane più in ascolto per alcuna ulteriore richiesta di connessione.

Per poter utilizzare una connessione `Multicast` (che tratteremo più avanti), si dovrebbe istanziare un thread per ogni connessione effettuata, in modo da lasciare il `server` sempre in ascolto.

Classe Socket

La classe **Socket** permette di definire una connessione **client-server** via **TCP** su entrambi i lati, sia **client** che **server**.

La differenza tra **client** e **server** sta nella modalità di creazione di un oggetto di questo tipo:

- ▶ nel **server** l'oggetto **Socket** viene creato dal metodo **accept()** della classe **ServerSocket**;
- ▶ il **client** dovrà provvedere a creare un'istanza di **Socket**: per creare un **socket** con un **server** in esecuzione su un certo **host** è sufficiente creare un oggetto di classe **Socket** specificando nel costruttore l'*indirizzo Internet dell'host* e il *numero di porta*.

Dopo che l'oggetto **Socket** è stato istanziato, è possibile ottenere (tramite appositi metodi) due **stream** (uno di input e uno di output): tramite essi è possibile comunicare con l'**host**, e riceverne messaggi. Il diagramma della classe **Socket** è il seguente.

Classe Socket	
Metodi costruttori	Metodi modificatori
Socket() Socket(SocketImpl port) Socket(String host, int port) Socket(InetAddress address, int port) Socket(String host, int port, InetAddress localAddr, int localPort) Socket(InetAddress address, int port, InetAddress localAddr, int localPort)	void setSpcketImplFactory(SocketImplFactory fac) SocketChannel getChannel() InetAddress getInetAddress() InputStream getInputStream() boolean get/setKeepAlive() InetAddress getLocalAddress() int getLocalPort() SocketAddress getLocalSocketAddress() boolean get/setOOBInline() OutputStream getOutputStream() int getPort() int get/setReceiveBufferSize() SocketAddress get/getRemoteSocketAddress() boolean get/setReuseAddress() int get/setSendBufferSize() int getSoLinger() int get/setSoTimeout() boolean get/setTopNoDelay() int get/setTrafficClass() boolean isBound() boolean isClosed() boolean isConnected() boolean isInputShutdown() boolean isOutputShutdown() void setSoLinger(boolean on, int linger) String toString() void bind(SocketAddress bindpoint) void close() void connect(SocketAddress endpoint) void connect(SocketAddress endpoint, int timeout) void sendUrgentData(int data) void shutdownInput() void shutdownOutput()

Qualsiasi metodo che prenda in ingresso un **InputStream** o un **OutputStream** può comunicare con l'host in rete: una volta creato il **Socket** si può comunicare in rete tramite l'utilizzo degli stream.

I costruttori della classe `Socket` sono i seguenti:

```
public Socket (String host, int port) throws IOException;  
public Socket (InetAddress address, int port) throws IOException.
```

Viene creato un oggetto `Socket` connettendosi con l'host specificato (sotto forma di stringa o di `InetAddress`) alla porta specificata. Se sull'host e sulla porta specificata non c'è un `server` in ascolto, viene generata un'`IOException` e viene specificato il messaggio `connection refused`.

Alcuni metodi tra i più utilizzati sono i seguenti:

```
public InetAddress getInetAddress(): restituisce un oggetto InetAddress corrispondente  
all'indirizzo dell'host con il quale il socket è connesso;  
public InetAddress getLocalAddress(): restituisce un oggetto InetAddress corrispondente  
all'indirizzo locale al quale il socket è collegato;  
static InetAddress getByName(String hostname): restituisce una istanza di InetAddress rap-  
presentante l'host specificato;  
public int getPort(): restituisce il numero della porta dell'host remoto con il quale il socket è  
collegato;  
public int getLocalPort(): restituisce il numero di porta locale con la quale il socket è collegato.
```

Quando si crea un `socket`, come già detto, ci si collega con un `server` su un determinato host, che è in ascolto su una certa porta; sulla macchina locale viene creato il `socket` su una determinata porta assegnata dal sistema operativo (il primo numero libero):

```
public InputStream getInputStream() throws IOException  
public OutputStream getOutputStream() throws IOException
```

Quando si comunica attraverso connessioni `TCP`, i dati vengono suddivisi in pacchetti (`IP packet`), quindi è consigliabile utilizzare degli stream "bufferizzati" evitando così di avere pacchetti contenenti poche informazioni.

La definizione dei due stream, rispettivamente di input e di output, da parte di un `server` verso un `client`, è per esempio la seguente:

```
BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));  
DataOutputStream out = new DataOutputStream(client.getOutputStream());
```

Dopo questa definizione l'uso dei `socket` diventa quindi trasparente: su questi oggetti si utilizzano i metodi `readLine()` e `println()`!

L'ultimo metodo che descriviamo è:

```
public synchronized void close() throws IOException
```

Con questo metodo viene chiuso il `socket` (e quindi la connessione) e tutte le risorse che erano in uso vengono rilasciate.

I dati bufferizzati verranno comunque spediti prima della chiusura del `socket`, chiusura anche solo di uno dei due stream associati ai `socket` che comporterà automaticamente la chiusura del `socket` stesso.

Nei metodi precedenti può essere lanciata un'`IOException`, a significare che ci sono stati problemi sulla connessione: questo succede quando uno dei due programmi che utilizza il `socket` chiude la connessione e quindi l'altro programma potrà ricevere una tale eccezione.

La realizzazione di un client TCP in Java

Possiamo ora realizzare un **client**: per comunicare con un host remoto usando il protocollo **TCP/IP**, si deve creare, per prima cosa, un oggetto **Socket** con tale host, specificando l'indirizzo **IP** dell'host e il numero di porta (naturalmente sull'host remoto dovrà essere presente un **server** in "ascolto" su tale porta).

Possiamo utilizzare indistintamente uno dei due costruttori della classe **Socket** sopra descritta:

```
public Socket (String host, int port) throws IOException;
public Socket (InetAddress address, int port) throws IOException.
```

Con questa istruzione viene creata una connessione con un'applicazione **server** (che nel frattempo è in attesa in una **accept()**) e restituisce il relativo **socket**.

Definiamo le variabili della nostra classe:

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

public class Client2 {
    String nomeServer = "nomeServer"; // indirizzo del server
    int portaServer = 6789; // porta x servizio
    DataInputStream in; // stream di input
    DataOutputStream out; // stream di output
    ...
}
```

Realizziamo ora la parte di un metodo che esegue le operazioni per aprire un **socket** con un **server** che si trova a un certo indirizzo (**nomeServer**) ed è in ascolto su una certa porta (**portaServer**) restituendo come parametro di ritorno un oggetto della classe **Socket**.

Una volta creato un oggetto **Socket** otteniamo gli stream a esso associati tramite i due metodi prima descritti della classe **Socket**: a questo punto, la comunicazione può avere inizio, cioè il **client** può scrivere sull'Output-Stream, come si fa con un normale stream, e ricevere dati dal **server** leggendo dall'InputStream.

```
protected Socket connetti () throws IOException
{
    Socket socket = new Socket (nomeServer, portaServer);

    out = new DataOutputStream (socket.getOutputStream ());
    in = new DataInputStream (socket.getInputStream ());
    ...
    return socket;
}
```

L'esempio seguente permette di leggere una stringa di testo dalla tastiera di un **client**. Esso invia la stringa a un **server**, il quale la modifica trasformandola in maiuscolo e la restituisce al **client**. La comunicazione avviene sulla porta 6789 sul **server** locale localhost.

Naturalmente, non possiamo mandarlo in esecuzione senza aver realizzato il **server**, ma per ora lo scriviamo e lo commentiamo, realizzando il **server** successivamente.

Definiamo le variabili da utilizzare nella classe.

```

1 import java.io.*;
2 import java.net.*;
3 public class ClientStr {
4     String nomeServer = "localhost";           // indirizzo server locale
5     int portaServer = 6789;                     // porta x servizio data e ora
6     Socket miosocket;
7     BufferedReader tastiera;                  // buffer per l'input da tastiera
8     String stringaUtente;                       // stringa inserita da utente
9     String stringaRicevutaDalServer;            // stringa ricevuta dal server
10    DataOutputStream outVersoServer;           // stream di output
11    BufferedReader inDalServer;                 // stream di input

```

Quindi definiamo il metodo che stabilisce la connessione con il **server**:

```

35 public Socket connetti(){
36     System.out.println("2 CLIENT partito in esecuzione ...");
37     try
38     {
39         // per l'input da tastiera
40         tastiera = new BufferedReader(new InputStreamReader(System.in));
41         // creo un socket
42         miosocket = new Socket(nomeServer,portaServer);
43         // miosocket = new Socket(InetAddress.getLocalHost(), 6789);
44         // associo due oggetti al socket per effettuare la scrittura e la lettura
45         outVersoServer = new DataOutputStream(miosocket.getOutputStream());
46         inDalServer = new BufferedReader(new InputStreamReader (miosocket.getInputStream()));
47     }
48     catch (UnknownHostException e){
49         System.err.println("Host sconosciuto"); }
50     catch (Exception e)
51     {
52         System.out.println(e.getMessage());
53         System.out.println("Errore durante la connessione!");
54         System.exit(1);
55     }
56     return miosocket;
57 }

```

Abbiamo creato un **socket** mediante il costruttore:

```
miosocket = new Socket(nomeServer,portaServer);
```

Trattandosi di **server** locale avremmo anche potuto utilizzare la seguente istruzione:

```
miosocket = new Socket(InetAddress.getLocalHost(), 6789);
```

Per le comunicazioni **I/O** abbiamo definito tre oggetti, il primo della classe **BufferedReader** per effettuare l'input da tastiera e gli altri due, rispettivamente, della classe **DataOutputStream** e **BufferedReader** per scrivere e leggere sul **socket**.

Definiamo un secondo metodo che effettua la conversazione:

- leggiamo la stringa inserita dall'utente e la scriviamo sullo stream del miosocket: quindi ci mettiamo in attesa della risposta del **server** mediante la **inDalServer.readLine()**;

- ▶ quando riceviamo la risposta la visualizziamo sullo schermo e terminiamo l'elaborazione chiudendo la porta con `miosocket.close()`.

```

13 public void comunica() {
14     try                // legge una riga
15     {
16         System.out.println("4 ... inserisci la stringa da trasmettere al server:"+'\n');
17         stringaUtente = tastiera.readLine();
18         //la spedisco al server
19         System.out.println("5 ... invio la stringa al server e attendo ...");
20         outVersoServer.writeBytes( stringaUtente+'\n');
21         //leggo la risposta dal server
22         stringaRicevutaDalServer=inDalServer.readLine();
23         System.out.println("8 ... risposta dal server "+'\n'+stringaRicevutaDalServer );
24         // chiudo la connessione
25         System.out.println("9 CLIENT: termina elaborazione e chiude connessione" );
26         miosocket.close();
27     }
28     catch (Exception e)
29     {
30         System.out.println(e.getMessage());
31         System.out.println("Errore durante la comunicazione col server!");
32         System.exit(1);
33     }
34 }

```

Concludono il metodo le istruzioni del costrutto `catch` che ci segnalano gli eventuali errori.

Il main è il seguente:

```

58 public static void main(String args[]) {
59     ClientStr cliente = new ClientStr();
60     cliente.connetti();
61     cliente.comunica();
62 }

```



Prova adesso!

Sulla base del **client** sopra descritto, scrivi un tuo **client** predisposto per connettersi a un **server** sempre presente sul tuo pc che trasmette e riceve una stringa.

La verifica della connessione verrà fatta al termine della successiva esercitazione di laboratorio oppure utilizzando come **server** `ServerStr.class` che puoi scaricare dal sito www.hoepliscuola.it nella cartella materiali nella sezione riservata al presente volume (file `UD-SOCKET.rar`).

ESERCITAZIONI DI LABORATORIO 2

JAVA SOCKET: REALIZZAZIONE DI UN SERVER TCP

Realizzazione di un server

In questa esercitazione realizzeremo il **server** corrispondente al **client** definito nella precedente esercitazione: dopo aver definito le variabili:

```

1 import java.io.*;
2 import java.net.*;
3 import java.util.*;
4
5 public class ServerStr
6 {
7     ServerSocket server    = null;
8     Socket client         = null;
9     String stringaRicevuta = null;
10    String stringaModificata = null;
11    BufferedReader inDalClient;
12    DataOutputStream outVersoClient;
13
14

```

Definiamo un metodo che effettua la connessione: per prima cosa si deve creare un oggetto **ServerSocket** che indichi il numero di porta sulla quale si mette in ascolto (nel nostro esempio, 6789) e mediante il metodo **accept()** aspetta un **client**.

```

17
18
19 public Socket attendi()
20 {
21     try
22     {
23         System.out.println("Il SERVER partito in esecuzione ...");
24         // creo un server sulla porta 6789
25         server = new ServerSocket(6789);
26         // rimane in attesa di un client
27         client = server.accept();
28         // chiudo il server per inibire altri client
29         server.close();
30         //associo due oggetti al socket del client per effettuare la scrittura e la lettura
31         inDalClient = new BufferedReader(new InputStreamReader (client.getInputStream()));
32         outVersoClient = new DataOutputStream(client.getOutputStream());
33     }
34 }

```

L'esecuzione dell'istruzione successiva avviene solamente se si è instaurata la connessione col **client** su tale porta: prima di iniziare la comunicazione, il **server** deve chiudere il **ServerSocket** per inibire ad altri **client** il collegamento, in quanto quell'indirizzo è già utilizzato (e ora noi stiamo implementando una comunicazione **Unicast**).

Conclude il metodo la gestione dell'errore di connessione:

```

27 catch (Exception e)
28 {
29     System.out.println(e.getMessage());
30     System.out.println("Errore durante l'istanza del server !");
31     System.exit(1);
32 }
33 return client;
34 }

```

Il metodo che effettua la comunicazione è molto simile a quello descritto per il **client**: dopo il saluto di benvenuto, il **server** si pone in attesa di lettura di una stringa dal canale di “input dal **client**”; al suo arrivo, la converte in maiuscolo e la trasmette al **client**.

Quindi si commiata dal **client** e chiude la connessione terminando la sua elaborazione.

```

38 public void comunica()
39 {
40     try
41     {
42         // rimango in attesa della riga trasmessa dal client
43         System.out.println("3 benvenuto client, scrivi una frase e la trasformo in maiuscolo. Attendo ...");
44         stringaRicevuta = inDalClient.readLine();
45         System.out.println("5 ricevuta la stringa dal cliente : "+stringaRicevuta);
46
47         //la modifico e la rispedisco al client
48         stringaModificata=stringaRicevuta.toUpperCase();
49         System.out.println("7 invio la stringa modificata al client ...");
50         outVersoClient.writeBytes(stringaModificata+'\n');
51
52         //termina elaborazione sul server : chiude la connessione del client
53         System.out.println("9 SERVER: fine elaborazione ... buona notte!");
54         client.close();
55     }

```

Il main è il seguente:

```

62
63 public static void main(String args[]) {
64     ServerStr servente = new ServerStr();
65     servente.attendi ();
66     servente.comunica();
67 }

```

Mandiamo ora in esecuzione le due classi: dato che utilizziamo la medesima macchina per entrambe le funzioni, dovremo attivare due istanze di **BlueJ**: una che ci permetterà di mandare in esecuzione il **server** e l'altra per il **client**.

L'output si presenta in due finestre diverse: per meglio comprendere la corretta sequenza di esecuzione delle operazioni queste sono state numerate in ordine progressivo. Ricordiamo di mandare in esecuzione prima il **server** altrimenti il **client** termina immediatamente con il seguente messaggio:



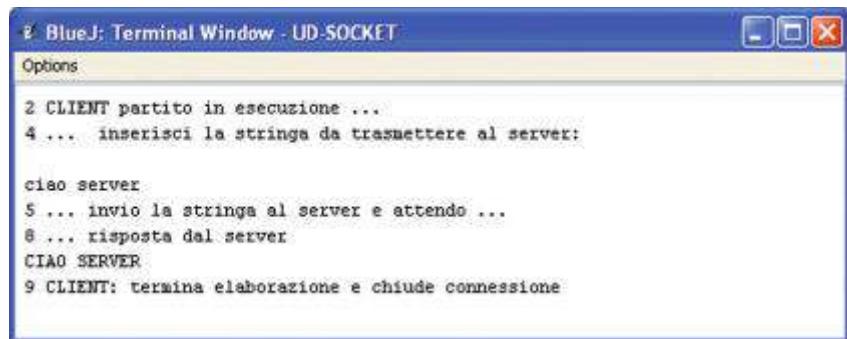
Effettuiamo una sequenza completa di operazioni, digitando come stringa da tradurre “ciao server”. ►



```

1 SERVER partito in esecuzione ...
3 benvenuto client, scrivi una frase e la trasforma in maiuscolo. Attendo ...
6 ricevuta la stringa dal cliente : ciao server
7 invio la stringa modificata al client ...
9 SERVER: fine elaborazione ... buona notte!
  
```

Nella finestra precedente sono riportate tutte le operazioni effettuate dal **server** e nella finestra seguente dal **client**. ►



```

2 CLIENT partito in esecuzione ...
4 ... inserisci la stringa da trasmettere al server:

ciao server
5 ... invio la stringa al server e attendo ...
6 ... risposta dal server
CIAO SERVER
9 CLIENT: termina elaborazione e chiude connessione
  
```



Prova adesso!

Realizza un'applicazione **client-server** dove il **client** invia un messaggio al **server**, questo conta il numero di vocali e di consonanti, e gli ritorna tali valori: il **client** continua a inviare frasi fino a che il numero di consonanti sia esattamente la metà del numero di vocali. In questo caso termina l'applicazione.

Esercizi proposti

- 1 Realizza una semplice calcolatrice: il client invia gli operandi e l'operatore al server, il quale esegue l'operazione e restituisce il risultato.
- 2 Realizza un sistema in cui il client riceve dal server un numero progressivo (tipo dispenser dei numeri per la coda dal panettiere).
- 3 Bomba a orologeria: un server spedisce a un client una bomba innescata con una miccia con valore random() e il client la rispedisce successivamente al server; a ogni operazione, entrambi riducono la miccia finché... la bomba scoppia.
- 4 Realizza il noto gioco della battaglia navale tra due giocatori che si trovano su due PC connessi in rete: il server provvede alle operazioni di avvio del gioco per poi diventare a tutti gli effetti il secondo giocatore.
- 5 Realizza il gioco dello "spara all'orso": un orso si muove orizzontalmente in modo casuale e un cacciatore gli spara spostandosi anch'esso orizzontalmente (il cacciatore è mosso dal giocatore mediante i tasti freccia).
- 6 Gioco del Tris: realizza il gioco del tris tra due giocatori connessi in rete locale.

ESERCITAZIONI DI LABORATORIO 3

REALIZZAZIONE DI UN SERVER MULTIPLO IN JAVA

Nella definizione del **server** nella esercitazione precedente abbiamo visto che all'inizio della comunicazione con il **client** viene chiusa la porta sulla quale il **server** era in attesa (cioè il **ServerSocket**, inizializzato sull'indirizzo 6789): infatti, tale **server** deve essere fermato per inibire ad altri **client** il collegamento contemporaneo sul medesimo indirizzo:

```
System.out.println("I SERVER partito in esecuzione ...");
// creo un server sulla porta 6789
server = new ServerSocket(6789);
// rimane in attesa di un client
client = server.accept();
// chiudo il server per ammettere altri client
server.close();
```

Spesso al **server** devono però connettersi più **client** contemporaneamente, a volte anche in numero non predeterminato (per esempio in una chat tra un **server** e un molteplice numero di **client**). La realizzazione di questo sistema avviene mediante un meccanismo **multithread** per la gestione della ricezione e dell'invio dei messaggi, in quanto non è prevedibile il numero di **client** da servire e il momento in cui ogni singolo **client** fa richiesta di invio di un nuovo messaggio.

I **server multithreaded** vengono tipicamente realizzati seguendo uno dei seguenti schemi:

- **on demand**: viene creato un **thread** per ogni **client** al momento della richiesta di connessione e viene terminato alla chiusura della comunicazione;
- **thread pool**: viene predeterminato e avviato un numero fisso di **thread** e ciascuno di essi viene assegnato alla richiesta di connessione da parte dei **client**; alla chiusura di una connessione il corrispondente **thread** non viene terminato ma viene sospeso per essere riutilizzato in una nuova connessione.

La scelta dello schema più consono varia in funzione delle situazioni, che quindi vanno accuratamente analizzate prima dell'implementazione. Infatti, ciascuno degli schemi sopra descritti presenta pregi e difetti; per esempio: il primo schema è più semplice ma più costoso (in termini di tempo, in quanto ogni volta deve essere creato un thread) mentre il secondo richiede un dimensionamento preventivo del pool (che potrebbe essere errato, o sottostimato e sovradimensionato).

Una soluzione ottimale può essere ottenuta con una forma intermedia, ovvero adottando uno schema misto. Modifichiamo ora l'esempio precedente per realizzare una situazione **on demand**: riscriviamo quindi il **server**.

Definiamo la classe con le necessarie variabili.

```

1  import java.net.*;
2  import java.io.*;
3  class ServerThread extends Thread {
4      ServerSocket server      = null;
5      Socket client            = null;
6      String stringaRicevuta    = null;
7      String stringaModificata = null;
8      BufferedReader inDalClient;
9      DataOutputStream outVersoClient;

```

Modifichiamo il costruttore in questo modo:

```

10
11  public ServerThread (Socket socket){
12      this.client = socket;
13  }

```

Anche il metodo `run()` viene limitato alla chiamata di un nuovo metodo `comunica()` e alla gestione delle eccezioni:

```

14
15  public void run(){
16      try{
17          comunica();
18      }catch (Exception e){
19          e.printStackTrace(System.out);
20      }

```

Il metodo che effettua la comunicazione è costituito da un ciclo infinito dal quale si esce quando il **client** ha trasmesso la stringa **FINE**: per il resto, si limita ad acquisire una riga dallo stream di input e ritrasmetterla al **client**.

La numerazione dei messaggi aiuta a seguire il flusso della comunicazione.

```

21  public void comunica ()throws Exception{
22      inDalClient      = new BufferedReader(new InputStreamReader (client.getInputStream()));
23      outVersoClient   = new DataOutputStream(client.getOutputStream());
24      for (;;){
25          stringaRicevuta = inDalClient.readLine();
26          if (stringaRicevuta == null || stringaRicevuta.equals("FINE")){
27              outVersoClient.writeBytes(stringaRicevuta+" (=server in chiusura...)"+ '\n');
28              System.out.println("Echo sul server in chiusura :"+ stringaRicevuta);
29              break;
30          }
31          else{
32              outVersoClient.writeBytes(stringaRicevuta+" (ricevuta e ritrasmessa)"+ '\n');
33              System.out.println("6 Echo sul server :"+ stringaRicevuta);
34          }
35      }
36      outVersoClient.close();
37      inDalClient.close();
38      System.out.println("9 Chiusura socket" + client);
39      client.close();
40  }
41  }
42  }

```


Realizziamo ora la classe più importante, cioè quella che rimane in attesa che un **client** si connetta alla sua porta (come nell'esempio precedente la 6789), mediante la:

```
21 // rimane in attesa di un client
22 client = server.accept();
```

Al momento della connessione, facciamo effettuare un'echo del **socket** ora istanziato che ci permette di osservare come a ogni nuovo **client** che si connette viene mantenuta attiva la *local port* 6789 mentre troveremo volta per volta un valore diverso per la port dove viene "trasferito".

Quindi viene creato un thread passando il **socket** al costruttore e successivamente viene avviato.

```
44 public class MultiServer{
45     public void start(){
46         try{
47             ServerSocket serverSocket = new ServerSocket(6789);
48             for (;;)
49             {
50                 System.out.println("1 Server in attesa ... ");
51                 Socket socket = serverSocket.accept();
52                 System.out.println("3 Server socket " + socket);
53                 ServerThread serverThread = new ServerThread(socket);
54                 serverThread.start();
55             }
56         }
57         catch (Exception e){
58             System.out.println(e.getMessage());
59             System.out.println("Errore durante l'istanza del server !");
60             System.exit(1);
61         }
62     }
}
```

In questo esempio il **server** non termina mai la propria esecuzione: il corpo del metodo **start()** è costituito da un ciclo infinito d'attesa sulla **accept()** per la generazione di un numero indefinito di **client**.

Il **main** si riduce alla seguente istruzione:

```
63
64 public static void main (String[] args){
65     MultiServer tcpServer = new MultiServer();
66     tcpServer.start();
67 }
```

Facciamo alcune modifiche anche al **client** per realizzare il metodo che permette di inviare più messaggi al **server**: il metodo modificato è **comunica()**.

In questo metodo realizziamo un ciclo infinito mediante l'istruzione **for(;;)**, dove ogni iterazione di tale ciclo esegue:

- ▶ input di una riga da parte dell'utente;
 - ▶ trasmissione al **server**;
 - ▶ ricezione dell'echo da parte del **server**;
- e termina quando l'utente inserisce la parola **FINE** chiudendo la connessione.

```

13 public void comunica() {
14     for (;;) // ciclo infinito: termina con FINE
15     try{
16         System.out.println("4 ... utente, inserisci la stringa da trasmettere al server:");
17         stringaUtente = tastiera.readLine();
18         //la spedisco al server
19         System.out.println("5 ... invio la stringa al server e attendo ...");
20         outVersoServer.writeBytes( stringaUtente+'\n');
21         //leggo la risposta dal server
22         stringaRicevutaDalServer=inDalServer.readLine();
23         System.out.println("7 ... risposta dal server "+'\n'+stringaRicevutaDalServer );
24         if (stringaUtente.equals("FINE")) {
25             System.out.println("8 CLIENT: termina elaborazione e chiude connessione" );
26             miosocket.close(); // chiudo la connessione
27             break;
28         }
29     }

```

E in caso di errore viene catturata l'eccezione:

```

28     }
29     catch (Exception e)
30     {
31         System.out.println(e.getMessage());
32         System.out.println("Errore durante la comunicazione col server!");
33         System.exit(1);
34     }

```

Il metodo `connetti()`, non molto diverso da quello precedentemente descritto, nell'esercitazione di laboratorio 1, è il seguente:

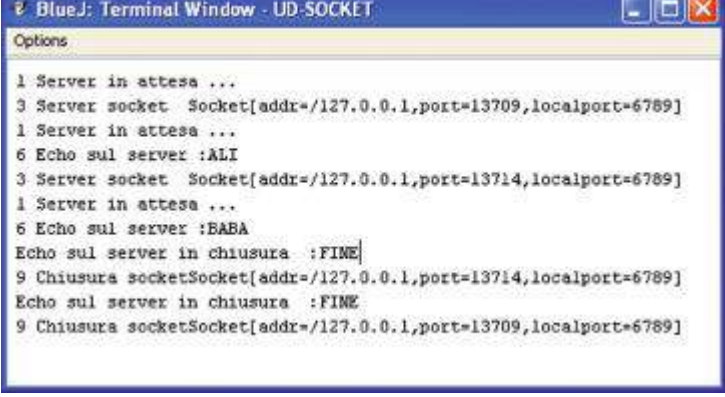
```

28 public Socket connetti(){
29     System.out.println("2 CLIENT partito in esecuzione ...");
30     try{
31         // input da tastiera
32         tastiera = new BufferedReader(new InputStreamReader(System.in));
33         // miosocket = new Socket(InetAddress.getLocalHost(), 6789);
34         miosocket = new Socket(nomeServer,portaServer);
35         // associo due oggetti al socket per effettuare la scrittura e la lettura
36         outVersoServer = new DataOutputStream(miosocket.getOutputStream());
37         inDalServer = new BufferedReader(new InputStreamReader (miosocket.getInputStream()));
38     }
39     catch (UnknownHostException e){
40         System.err.println("Host sconosciuto"); }
41     catch (Exception e){
42         System.out.println(e.getMessage());
43         System.out.println("Errore durante la connessione!");
44         System.exit(1);
45     }
46     return miosocket;
47 }

```

Il resto della classe (chiamata public class `Multiclient()`) è identico al `client` descritto nel paragrafo precedente.

Vediamo gli output generati da una esecuzione: mandiamo dapprima in esecuzione il **server** e successivamente due **client**.



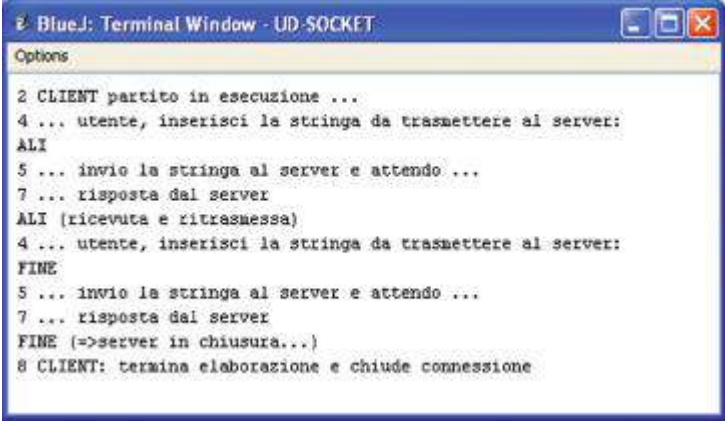
```

2 Server in attesa ...
3 Server socket Socket[addr=/127.0.0.1,port=13709,localport=6789]
1 Server in attesa ...
6 Echo sul server :ALI
3 Server socket Socket[addr=/127.0.0.1,port=13714,localport=6789]
1 Server in attesa ...
6 Echo sul server :BABA
Echo sul server in chiusura :FINE
9 Chiusura socketSocket[addr=/127.0.0.1,port=13714,localport=6789]
Echo sul server in chiusura :FINE
9 Chiusura socketSocket[addr=/127.0.0.1,port=13709,localport=6789]

```

I **client** comunicano al **server** solamente una parola (rispettivamente **ALI** e **BABA**): nel primo (**ALI**) digitiamo **FINE** mentre apriamo una nuova finestra e mandiamo in esecuzione un terzo **client**.

Di seguito, è riprodotto l'**echo()** sul **client ALI**:

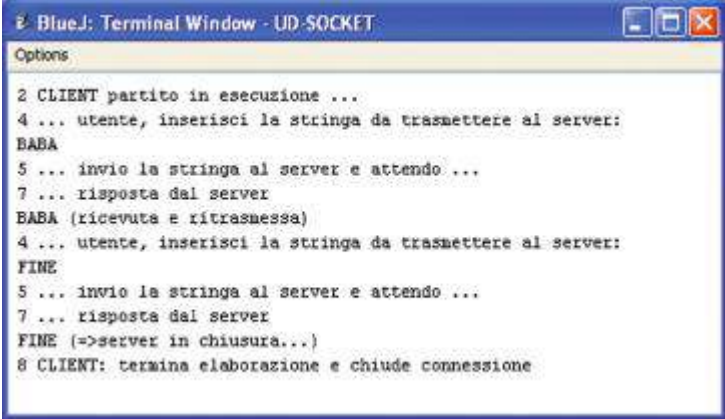


```

2 CLIENT partito in esecuzione ...
4 ... utente, inserisci la stringa da trasmettere al server:
ALI
5 ... invio la stringa al server e attendo ...
7 ... risposta dal server
ALI (ricevuta e ritrasmessa)
4 ... utente, inserisci la stringa da trasmettere al server:
FINE
5 ... invio la stringa al server e attendo ...
7 ... risposta dal server
FINE (=>server in chiusura...)
8 CLIENT: termina elaborazione e chiude connessione

```

e l'**echo()** sul **client BABA**:



```

2 CLIENT partito in esecuzione ...
4 ... utente, inserisci la stringa da trasmettere al server:
BABA
5 ... invio la stringa al server e attendo ...
7 ... risposta dal server
BABA (ricevuta e ritrasmessa)
4 ... utente, inserisci la stringa da trasmettere al server:
FINE
5 ... invio la stringa al server e attendo ...
7 ... risposta dal server
FINE (=>server in chiusura...)
8 CLIENT: termina elaborazione e chiude connessione

```



Prova adesso!

Realizza un'applicazione che simula una lotteria di 90 numeri; alcuni giocatori si collegano e "acquistano" alcuni numeri (per esempio 5) estratti casualmente dal **server**: quando si raggiunge un numero adeguato di giocatori (almeno 4) e non avvengono successive connessioni per almeno 60 secondi il **server** provvede alla estrazione di 5 premi, comunicando ai giocatori l'esito della estrazione; a loro volta i **client**, se hanno un numero vincente, lo segnalano al **server** e ritirano la vincita.

Esercizi proposti

- 1 Realizza l'agenzia stampa PANZA: ogni redazione locale invia a un server una notizia catalogata secondo lo schema Settore/Argomento/Area geografica; l'agenzia server le deve smistare in base a tale classificazione e inviare come risposta tutte le notizie analoghe.
- 2 Realizza un sistema in cui il server sia il banditore di un'asta: accoglie le offerte e comunica se sono accettabili o meno, comunica a richiesta la migliore offerta corrente; i client sono i partecipanti all'asta che possono richiedere quale sia l'offerta migliore (e di chi) e possono effettuare rilanci se il loro budget a disposizione lo consente.
- 3 Realizza un sistema che simuli la Borsa Valori: il server comunica ciclicamente un listino azionario con la quotazione dei singoli titoli a tutti i client a lui connessi, i quali possono comunicare una variazione (verso l'alto o verso il basso di una certa quantità fissa) del valore di un titolo del listino.
- 4 Realizza un sistema per la gestione del gioco della tombola dove il server gestisce il tabellone e assegna a ogni singolo client che si connette una cartella casuale; ogni client controlla i numeri estratti e segnala al server le eventuali vincite.
- 5 Realizza la simulazione di una corsa dei cavalli: quattro cavalli percorrono uno o più giri di un ippodromo ovale fino al termine della gara. Aggiungi successivamente la possibilità di effettuare puntate sui cavalli.
- 6 Realizza un sistema che gestisca il movimento di automobili in rete: il server attende la connessione di N giocatori che muovono la propria auto mediante i tasti freccia.
- 7 Simula il gioco della tombola: realizza un sistema di estrazioni del lotto: N giocatori scelgano tre numeri e li comunichino alla ricevitoria. Quindi effettua le estrazioni del lotto comunicando eventuali vincite.

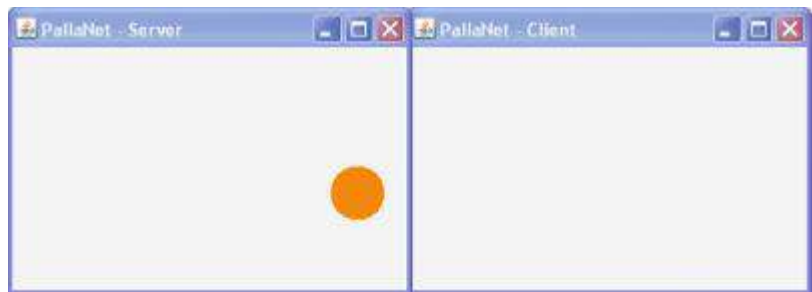
ESERCITAZIONI DI LABORATORIO 4

JAVA SOCKET: UNA ANIMAZIONE CLIENT-SERVER

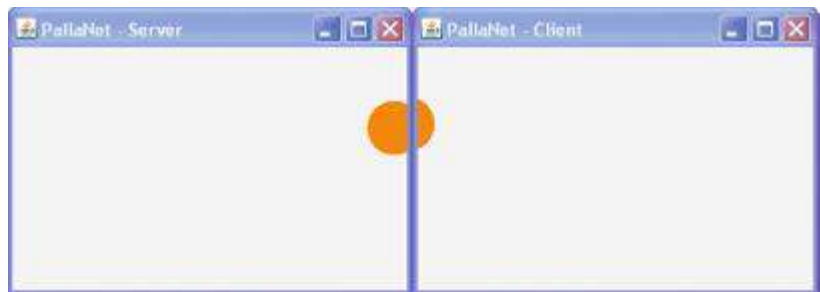
Una palla che “attraversa” le finestre

Realizziamo ora un'applicazione costituita da due finestre indipendenti affiancate e una pallina in movimento che si sposta nella prima finestra: quando raggiunge il bordo destro, anziché rimbalzare, “esce” dalla prima finestra ed “entra” nella seconda (come mostrato nella sequenza di immagini).

Si avvicina al bordo destro della finestra prima finestra: ►



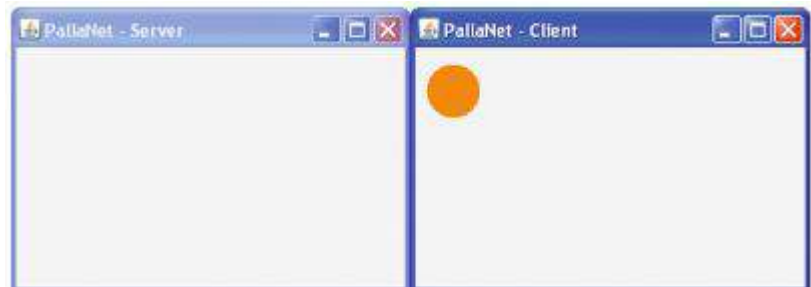
inizia l'attraversamento ►



procede nella seconda finestra ►



e termina l'attraversamento passando completamente nella seconda finestra dove continua il suo movimento, rimbalza sui suoi bordi e ritorna nella prima finestra. ►



Descrizione della soluzione

Realizziamo il sistema con una applicazione **client-server**, quindi definiamo due classi e facciamo comunicare tra loro.

La pallina viene generata dal **server** non appena si accorge che il **client** si è connesso.

Dovendo gestire un'animazione, abbiamo problemi di sfarfallio che annulliamo con la tecnica del **doppio buffering**: realizziamo l'animazione gestendo l'evento generato dall'oggetto **Timer** e, dato che è proprio il metodo che gestisce tale evento a incaricarsi del movimento della pallina, e quindi riconoscere il momento in cui "sta per lasciare" la finestra, demandiamo a tale metodo la realizzazione della comunicazione tra **client** e **server** per ricevere la segnalazione.

Implementiamo due classi:

```
class PannelloAnimazione extends JPanel implements ActionListener{...}
class PannelloClient extends JPanel implements ActionListener{ ...}
```

rispettivamente per il **server** e per il **client** che si occupano, quindi, di:

- animare la pallina;
- eliminare lo sfarfallio;
- comunicare tra loro le situazioni di transazione.

Nel complesso le due classi sono abbastanza simili: descriviamo solamente la classe **server** dove evidenzieremo le differenze con la classe del **client**.

La classe "starter" è la seguente, cioè **PallaNetServer**, che definisce la finestra e avvia un thread che attende la connessione del **client**:

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import java.io.*;
4 import java.net.*;
5 import javax.swing.*;
6 public class PallaNetServer extends JFrame{
7     private Socket connessione = null;
8     private DataOutputStream out = null;
9     private DataInputStream input = null;
10    public PallaNetServer() {
11        super("PallaNet - Server");
12        this.setSize(500,400);           //setto la grandezza della finestra
13        this.setLocationRelativeTo(null); //setto la posizione al centro dello schermo
14        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
15        //istanzio un oggetto per attendere la connessione di un client
16        ThreadConnessione attendiConnessione = new ThreadConnessione(this);
17        this.setVisible(true);
18    } //fine costruttore
```


Unica osservazione nella definizione di classe è che estende la classe **JFrame**: nulla da sottolineare nel costruttore tranne che per l'utilizzo di un oggetto della classe **ThreadConnessione()** che rimane in attesa sulla porta **6789** mediante il metodo **setConnessione()**, dopo aver ricevuto dal costruttore il riferimento alla finestra.

Nel caso di singolo **client** non è necessario utilizzare i **thread**, ma questo **server** è stato così impostato a titolo di esempio, in modo da poter essere utilizzato intervenendo con "piccole modifiche" anche nel caso di connessione tra più **client**, come viene richiesto nell'esercizio proposto al termine della spiegazione di questo esempio.

```

61 class ThreadConnessione implements Runnable{
62     private PallaNetServer finestra;
63     private Thread me;
64     public ThreadConnessione(PallaNetServer finestra){
65         //ottengo il riferimento alla JFrame
66         this.finestra = finestra;
67         me = new Thread(this);           //istanzio il Thread
68         me.start();                       //attivo il Thread
69     }
70     public void run(){
71         try{                             //istanzio un oggetto in ascolto sulla porta 6789
72             ServerSocket server = new ServerSocket(6789);
73             finestra.setConnessione(server.accept()); //attesa connessione
74             server.close();               //chiudo il server
75         }catch(Exception e){
76             JOptionPane.showMessageDialog(null,e.toString());
77             System.exit(-1);
78         }
79     } //fine metodo run del Thread
80 }

```

Non appena il **client** si connette viene "chiuso" il **server** oppure si segnala un errore: naturalmente nel caso si dovessero attendere più **client** è necessario intervenire in questa prima parte di codice.

Con l'istruzione 73 l'avvenuta connessione col **client** richiama il metodo **setConnessione()** che, utilizzando il riferimento alla connessione con il **client**, crea gli stream di input e output e definisce e istanzia un **PannelloAnimazione()** avviando, di fatto, il movimento della pallina.

```

29 public void setConnessione(Socket connessione){
30     //ricevo il riferimento per la connessione con il client
31     this.connessione = connessione;
32     //ricevo lo stream di output e di input
33     try{
34         out = new DataOutputStream(connessione.getOutputStream());
35         input = new DataInputStream(connessione.getInputStream());
36     }catch(Exception e){
37         JOptionPane.showMessageDialog(null,e.toString());
38         System.exit(-1);
39     }
40     //inizio l'animazione su un oggetto di classe PannelloAnimazione
41     PannelloAnimazione pan = new PannelloAnimazione(this,this.getSize());
42     //lo aggiungo alla JFrame
43     this.add(pan);
44 }

```

Completano la classe due metodi definiti per rispettare l'incapsulamento. ►

```

45
46 public DataInputStream getInput(){
47     return input;
48 }
49
50 public DataOutputStream getOutputStream(){
51     return out;
52 }
53
54 } /* --Fine classe PallaNetServer */

```

Le differenze con la classe del **client**, la **PallaNetClient**, si limitano alle istruzioni di connessione. Al posto del thread che effettua la connessione con le seguenti istruzioni:

```

67 ServerSocket server = new ServerSocket(6789);
68 finestra.setConnessione(server.accept()); //attesa connessione
69 server.close(); //chiudo il server

```

nel **client** mettiamo semplicemente la seguente istruzione che crea la connessione:

```

21 //mi connetto al server sul socket(<nomecomputer>,<porta>)
22 connessione = new Socket("localhost",6789);

```

La definizione della classe e del suo costruttore è la seguente:

```

1 import java.awt.*;
2 import java.awt.event.*;
3 import java.io.*;
4 import java.net.*;
5 import javax.swing.*;
6 public class PallaNetClient extends JFrame{
7     private Socket connessione = null;
8     private DataOutputStream out = null;
9     private DataInputStream input = null;
10    public PallaNetClient(){
11        super("PallaNet - Client"); // costruttore della super classe
12        this.setSize(500,400); // grandezza della finestra
13        this.setLocationRelativeTo(null); // posizione al centro schermo
14        this.setDefaultCloseOperation(EXIT_ON_CLOSE); //modalità di chiusura
15        connetti(); //mi connetto al server
16        this.setVisible(true); //rendo visibile la finestra
17    } //fine costruttore
18
19

```

Il metodo che effettua la connessione e avvia l'animazione della pallina è il seguente:

```

18
19 public void connetti(){
20     try {
21         //mi connetto al server sul socket(<nomecomputer>,<porta>)
22         connessione = new Socket("localhost",6789);
23         //ottengo lo stream di input dal server e di output verso il server
24         out = new DataOutputStream(connessione.getOutputStream());
25         input = new DataInputStream(connessione.getInputStream());
26     } catch (Exception e){
27         JOptionPane.showMessageDialog(null,e.toString());
28         System.exit(-1);
29     }
30     //inizio l'animazione
31     Container contenitore = this.getContentPane();
32     PannelloClient pan = new PannelloClient(this,contenitore.getSize());
33     contenitore.add(pan); //lo aggiungo alla JFrame
34 }

```

Conclude la classe la definizione dei due metodi getter: ►

```

16 public DataInputStream getInput(){
17     return input;
18 }
19 public DataOutputStream getOutput(){
20     return out;
21 }

```

I pannelli che gestiscono l'animazione sono simili: descriviamo quindi solo quello del **server**, cioè la classe **PannelloAnimazione**.

Iniziamo definendo la classe con le variabili necessarie per l'animazione:

```

67 class PannelloAnimazione extends JPanel implements ActionListener{
68     private PallaNetServer finestra;
69     private Dimension dimensioni;
70     private Image buffer Virtuale;           // per il doppio buffering
71     private Graphics offScreen;
72     private Timer tim = null;
73     private int xPallina = 0;               // coordinate iniziali
74     private int yPallina = 0;
75     private int diametroPallina = 40;
76     private int spostamento = 3;
77     private int timerDelay = 10;
78     private boolean destra,basso,pallinaPresente,arrivoComunicato;
79 }

```

Gli attributi sono “parlanti” e non necessitano di spiegazioni aggiuntive. Anche il suo costruttore non presenta particolarità degne di nota:

```

89 public PannelloAnimazione(PallaNetServer finestra,Dimension dimensioni){
90     super();
91     this.finestra = finestra;           // riferimento alla finestra corrente
92     this.setSize(dimensioni);          // setto la grandezza
93     destra = true;                     // direzioni iniziali della pallina
94     basso = true;
95     pallinaPresente = true;            // la pallina parte dal questo schermo
96     arrivoComunicato = false;
97     tim = new Timer(timerDelay,this); // instancio il Timer per il movimento
98     tim.start();                       // attivo il Timer
99 }

```

Completano la classe i due metodi che si occupano del disegno evitando lo sfarfallio mediante la tecnica del **doppio buffering**:

```

101 public void update(Graphics g){        // aggiorno il pannello
102     paint(g);                          // eseguo il disegno in Doppio Buffering
103 }
104 public void paint(Graphics g){
105     super.paint(g);
106     //definisco lo spazio esterno per il doppioBuffering
107     bufferVirtuale = createImage(getWidth(),getHeight());
108     offScreen = bufferVirtuale.getGraphics();
109     Graphics2D screen = (Graphics2D) g;
110     offScreen.setColor(new Color(254,138,22)); // setto il colore della palla
111     if(pallinaPresente){                // disegno la palla
112         offScreen.fillOval(xPallina,yPallina,diametroPallina,diametroPallina);
113     }
114     //disegno l'immagine modificata "bufferVirtuale" sul Component
115     screen.drawImage(bufferVirtuale,0,0,this);
116     offScreen.dispose();
117 }

```

Più articolato è il metodo che gestisce l'evento generato dal **time** e che quindi si occupa del movimento della pallina ed eventualmente della “migrazione” tra le due finestre. Lo analizziamo in tre segmenti:

- ▶ aggiornamento dello spostamento della palla in verticale;
- ▶ aggiornamento dello spostamento della palla in orizzontale;
- ▶ palla assente nella finestra.

Nulla da segnalare quando aggiorniamo il moto verticale in quanto dobbiamo solo controllare quando avviene la collisione con i bordi per invertire la direzione:

```

123 public void actionPerformed(ActionEvent e){
124     if(pallinaPresente){
125         /*direzione in verticale*/
126         if(basso){
127             //si muove verso il basso
128             if(yPallina > (this.getHeight()-diametroPallina)){// urto inferiore
129                 basso = false;                                //cambia direzione
130                 yPallina -= spostamento;
131             }else{
132                 yPallina += spostamento;
133             }
134         }else{
135             //si muove verso l'alto
136             if(yPallina <= 0){                                // urto superiore
137                 basso = true;                                //cambia direzione
138                 yPallina += spostamento;
139             }else{
140                 yPallina -= spostamento;
141             }
142         }
143     }
144 }

```

Durante la direzione orizzontale abbiamo due diverse situazioni:

- ▶ nel caso di *movimento verso destra*, quando la pallina tocca il bordo destro, inizia la “migrazione” nella finestra del **client**; dobbiamo quindi:
 - ▶ comunicare al **client** che inizia il trasferimento;
 - ▶ comunicare al **client** la coordinata verticale di “passaggio”;
 - ▶ comunicare al **client** la direzione verticale.

Quindi impostiamo una variabile booleana (**arrivoComunicato**) utilizzata dal **server** per non ripetere la comunicazione al **client** delle successive posizioni della pallina durante la transazione:

```

143     /*direzione orizzontale*/
144     if(destra){
145         if(!arrivoComunicato && (xPallina > (this.getWidth()-diametroPallina))){
146             /* comunica il verso di direz. verticale e la coordinata verticale
147              * in cui la pallina si trova attualmente */
148             try{
149                 finestra.getOutputStream().writeBoolean(basso);
150                 finestra.getOutputStream().writeInt(yPallina);
151                 arrivoComunicato = true;
152             }catch(Exception ecc){
153                 JOptionPane.showMessageDialog(null,ecc.toString());
154                 System.exit(-1);
155             }
156         }else{
157             xPallina += spostamento;
158             if(xPallina > this.getWidth()){
159                 pallinaPresente = false;
160                 arrivoComunicato = false;
161             }
162         }
163     }
164 }

```


Durante la transazione avremo due palline presenti, una nel **client** che va “scomparendo” e una nel **server** che si sta “formando”.

Quando la pallina ha lasciato completamente la finestra del **server** vengono modificati i valori delle due variabili booleane **pallinaPresente** e **arrivoComunicato** che vengono posti al valore di falso.

- B** Nel caso di *movimento verso sinistra* è sufficiente controllare se si è raggiunto il bordo laterale dello schermo per invertire la direzione:

```

163 }else{                                     // la pallina sta andando a sinistra
164     if(xPallina <=0 ){                     // fine finestra a sinistra
165         destra = true;                    // cambio direzione
166         xPallina += spostamento;
167     }else{
168         xPallina -= spostamento;
169     }
170 }

```

Quando la pallina non è presente, il **server** rimane in attesa con l'istruzione 177 di una eventuale comunicazione da parte del **client** dell'inizio di “ritorno pallina”: in questa occasione vengono aggiornate le coordinate e la variabile di stato **pallinaPresente = true**.

```

171 }else{                                     // la pallina non è presente
172     // rimango in attesa del ritorno della pallina nel mio schermo
173     boolean direzione = false;
174     int y = 0;
175     try{
176         direzione = finestra.getInput().readBoolean();
177         y = finestra.getInput().readInt(); // attendo che arriva la pallina
178         basso = direzione;                // direzione di ingresso
179         destra = false;                   // si muove verso sinistra
180         yPallina = y;                     // coord. iniziali della pallina
181         xPallina = this.getWidth();
182         pallinaPresente = true;
183     }
184     catch(Exception ecc){
185         JOptionPane.showMessageDialog(null,ecc.toString());
186         System.exit(-1);
187     }
188 }

```

L'ultima operazione del metodo effettua il ridisegno invocando il metodo **repaint()**.

```

189     repaint();
190 }
191 }

```

Il codice del **client** è simile: sono naturalmente invertite le situazioni di movimento della pallina, cioè se dal **server** esce dal lato destro, nel **client** entrerà dal lato sinistro e da questo lato successivamente uscirà per rientrare nel **server** dallo stesso lato da cui è uscita.

Anche le classi di prova sono simili, per cui riportiamo solo quella del **server** a titolo di esempio. ►

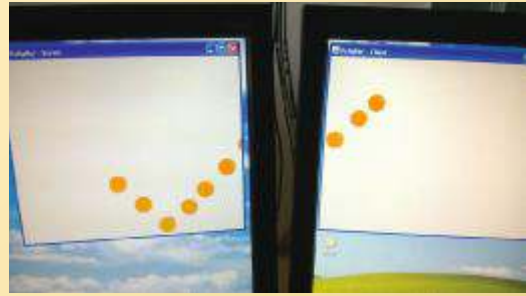
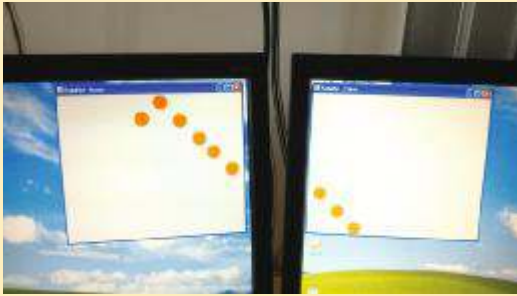
```

1 public class ProvaPallaServer{
2     public static void main(String args[]){
3         PallaNetServer server = new PallaNetServer();
4     }
5 }

```

Maggior effetto scenografico si ottiene mandando in esecuzione i due thread su due macchine diverse, connesse in rete, ponendo i due monitor vicini: unica modifica da apportare alla classe **client** è l'aggiornamento del nome del PC su cui viene mandato in esecuzione il thread **server**.

La fotografia seguente ne riporta una esecuzione



```
class PannelloAnimazione extends JPanel implements ActionListener {...}
class PannelloClient extends JPanel implements ActionListener{ ...}
```



Prova adesso!

Modifica il programma precedente aggiungendo un secondo **client**, cioè una terza finestra, facendo in modo che la pallina esca prima a destra e successivamente a sinistra dalla videata centrale.

ESERCITAZIONI DI LABORATORIO 5

IL PROTOCOLLO UDP NEL LINGUAGGIO JAVA

Client e server UDP in Java

Il protocollo **UDP** ha caratteristiche diverse dal protocollo **TCP**: **UDP** non è orientato alla connessione e quindi tra due host non si crea uno stream stabile.

L'avvio di una comunicazione avviene senza handshaking e la consegna non è garantita; al contrario i controlli sull'integrità vengono eseguiti come in **TCP**.

Uno dei principali vantaggi di **UDP** è che, grazie alla sua semplicità offre un servizio molto rapido e non richiede **handshaking** (per questo viene spesso utilizzato per **DNS**).

In questa esercitazione realizzeremo la comunicazione mediante protocollo **UDP** dove un mittente (**UDPClient.java**) trasmette un messaggio (datagram) a un destinatario (**UDPServer.java**).

Ricordiamo che in una comunicazione dati “connectionless o datagram” il canale:

- trasporta messaggi;
- non è affidabile;
- il **socket** è condiviso;
- non preserva l'ordine delle informazioni.

Inoltre l'ordine di arrivo dei pacchetti non è necessariamente lo stesso di invio.

Realizzare un sistema **client-server UDP** consiste nell'implementare un programma che effettui i seguenti passi fondamentali.

Per il **client**:

- crea il **socket**;
- manda richiesta sul **socket** (composto da *indirizzo*, *porta* e *messaggio*);
- riceve dati dal **socket**;
- chiude il **socket**.

Per il **server**:

- crea il **socket**
- ripete le seguenti operazioni:
 - si mette in attesa delle richieste in arrivo;
 - riceve il *messaggio* dal **socket**;
 - invia la *risposta* sul **socket** al **client** che ha fatto una richiesta;
- chiude il **socket**.

La comunicazione **UDP** viene realizzata in **Java** mediante la classe **DatagramSocket**.

Classe DatagramSocket

La classe `Java DatagramSocket` usa il protocollo `UDP` per trasmettere dati attraverso i `socket`: permette a un `client` di connettersi a una determinata porta di un host per leggere e scrivere dati impacchettati attraverso la classe `DatagramPacket`.

L'indirizzo dell'host destinatario è sul `DatagramPacket`.

I principali metodi della `DatagramSocket(..)` sono i seguenti.

► Metodi costruttori:

```
void DatagramSocket() throws SocketException
void DatagramSocket(int porta) throws SocketException
```

► Metodi per ricevere/trasmettere:

```
void send(DatagramPacket pacchettoDati) throws IOException
void receive(DatagramPacket pacchettoDati) throws IOException
```

Il metodo `receive()` blocca il chiamante fino a quando un pacchetto è ricevuto

```
void setSoTimeout(int timeout) throws SocketException
```

usando `setSoTimeout()` il chiamante di `receive()` si blocca al dopo timeout millisecondi.

Infine, il “solito” metodo per chiudere la connessione:

```
void close()
```

Classe DatagramPacket

La classe `DatagramPacket` è la classe `Java` che permette di rappresentare i pacchetti `UDP` da inviare e ricevere sui `socket` di tipo `datagram`.

Il costruttore della classe è il seguente:

```
public DatagramPacket(byte buffer[], int lunghezza, InetAddress indirizzo, int porta)
```

Il `client` crea un oggetto di `DatagramPacket` passando al costruttore:

- il contenuto del messaggio: un array di `lunghezza` caratteri;
- l'`indirizzo IP` del `server`;
- il numero di `porta` su cui il `server` è in ascolto.

Il `server`, per ricevere un messaggio, crea un oggetto di `DatagramPacket` definendone la lunghezza massima:

```
public DatagramPacket(byte buffer[], int lunghezza)
```

I principali metodi della classe `DatagramPacket` sono:

Ⓐ Per la gestione dell'indirizzo `IP`:

```
void setAddress(InetAddress indirizzo): setta il valore dell'indirizzo IP;
InetAddress getAddress(): restituisce l'indirizzo IP del'host da cui il pacchetto è stato ricevuto.
```

B Per la gestione della porta:

`void setPort(int porta):` setta il valore della porta;
`int getPort():` restituisce la porta della macchina remota da cui il pacchetto è stato ricevuto.

C Per la gestione dei dati:

`void setData(byte[] buffer) :` inserisce i dati nel pacchetto;
`byte[] getData() :` restituisce i dati del pacchetto.

Server UDP

Scriviamo il **server UDP**: creiamo un `DatagramSocket()` sulla solita porta **6789** e definiamo i due buffer per i dati, rispettivamente per la ricezione e per la trasmissione, per esempio della medesima dimensione di 1024 byte.

Una variabile booleana viene definita per essere utilizzata come condizione di ripetizione/uscita dal ciclo.

```
1 import java.io.*;
2 import java.net.*;
3
4 class UDPServer{
5     public static void main(String args[]) throws Exception {
6         DatagramSocket serverSocket = new DatagramSocket(6789);
7         boolean attivo = true;           // per la ripetizione del servizio
8         byte[] bufferIN = new byte[1024]; // buffer spedizione e ricezione
9         byte[] bufferOUT = new byte[1024];
10
11         System.out.println("SERVER avviato...");
12         while(attivo)
```

Definiamo ora il datagramma e ci poniamo in attesa di ricevere dati da qualche **client**:

```
14 // definizione del datagramma
15 DatagramPacket receivePacket =
16     new DatagramPacket(bufferIN,bufferIN.length);
17 //attesa della ricezione dato dal client
18 serverSocket.receive(receivePacket);
19
```

Quando viene ricevuto un pacchetto, viene analizzato estraendo il messaggio, individuando la lunghezza impostata alla trasmissione per eliminare i caratteri superflui presenti nel buffer, che ricordiamo essere di 1024 byte;

```
20 // analisi del pacchetto ricevuto
21 String ricevuto = new String(receivePacket.getData());
22 int numCaratteri = receivePacket.getLength();
23 ricevuto=received.substring(0,numCaratteri); //elimina i caratteri in eccesso
24 System.out.println("RICEVUTO: " + ricevuto);
```

Sempre dal pacchetto ricevuto vengono individuati i parametri per la trasmissione della risposta, cioè l'indirizzo e il numero di porta del **client**:

```
25
26 // riconoscimento dei parametri del socket del client
27 InetAddress IPClient = receivePacket.getAddress();
28 int portaClient = receivePacket.getPort();
29
```

Viene preparata la risposta, che in questo caso non è altro che il messaggio ricevuto trasformato in maiuscolo, viene creato il datagramma in uscita e viene inviato al **client**:

```

30 // preparo il dato da spedire
31 String daSpedire = ricevuto.toUpperCase();
32 bufferOUT = daSpedire.getBytes();
33 // invio del Datagramma
34 DatagramPacket sendPacket =
35     new DatagramPacket(bufferOUT, bufferOUT.length, IPClient, portaClient);
36 serverSocket.send(sendPacket);

```

Se il **client** ha inviato la parola “fine”, viene conclusa l’elaborazione del **server**, si esce quindi dal ciclo e si chiude il **socket**.

```

38 // controllo termine esecuzione del server
39 if (ricevuto.equals("fine"))
40 {
41     System.out.println("SERVER IN CHIUSURA. Buona serata.");
42     attivo=false;
43 }
44 }
45 serverSocket.close();
46 }

```

Client UDP

Scriviamo ora il **client UDP**: come per il **server** definiamo i buffer per la comunicazione e indichiamo i riferimenti del **socket** del **server**:

```

1 import java.io.*;
2 import java.net.*;
3
4 class UDPClient
5 {
6     public static void main(String args[]) throws Exception
7     {
8         int portaServer = 6789; // porta del server
9         InetAddress IPServer = InetAddress.getByName("localhost");
10
11         byte[] bufferOUT = new byte[1024]; // buffer di spedizione e ricezione
12         byte[] bufferIN = new byte[1024];
13         BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
14

```

Creiamo un **socket** e invitiamo l’utente a inserire un dato:

```

15 // creazione del socket
16 DatagramSocket clientSocket = new DatagramSocket();
17 System.out.println("Client pronto - inserisci un dato da inviare:");

```

Leggiamo la stringa inserita dall’utente e predisponiamo il buffer di uscita:

```

19 // preparazione del messaggio da spedire
20 String daSpedire = input.readLine();
21 bufferOUT = daSpedire.getBytes();

```

Creiamo ora un **DatagramPacket** con il messaggio da inviare e i parametri del **server** e lo inviamo a destinazione:

```

23 // trasmissione del dato al server
24 DatagramPacket sendPacket =
25     new DatagramPacket(bufferOUT, bufferOUT.length, IPServer, portaServer);
26 clientSocket.send(sendPacket);

```

Quindi ci mettiamo in attesa della risposta, predisponendo il **DatagramPacket** con il buffer per la ricezione:

```

27
28 // ricezione del dato dal server
29 DatagramPacket receivePacket = new DatagramPacket(bufferIN, bufferIN.length);
30 clientSocket.receive(receivePacket);
31 String ricevuto = new String(receivePacket.getData());

```

Alla ricezione del messaggio da parte del **server**, questo viene analizzato, vengono tolti i caratteri eccedenti presenti nel buffer di ricezione e viene visualizzato il risultato:

```

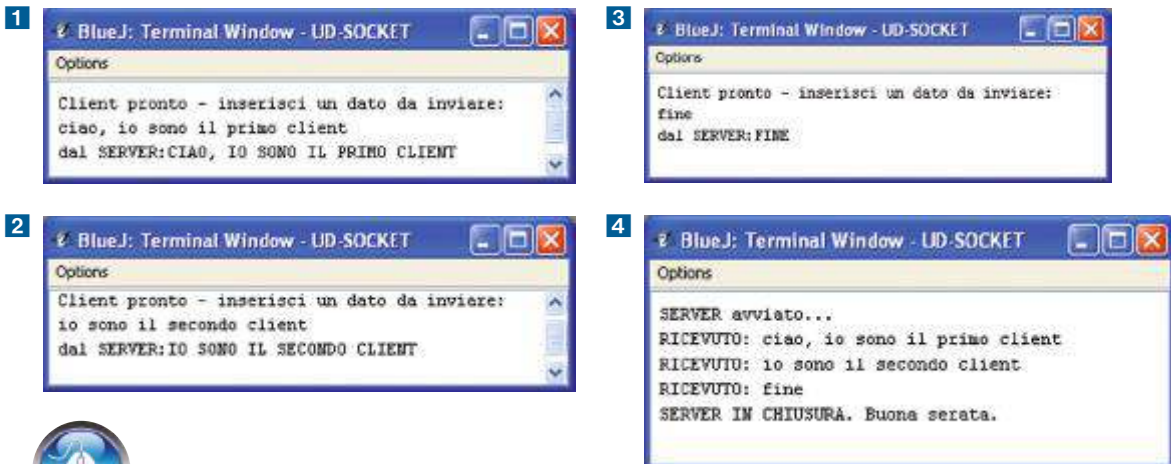
// elaborazione dei dati ricevuti
int numCaratteri = receivePacket.getLength();
ricevuto=ricevuto.substring(0,numCaratteri); //elimina i caratteri in eccesso
System.out.println("dal SERVER:" + ricevuto);

// termine elaborazione
clientSocket.close();

```

Il **client** termina l'elaborazione chiudendo la connessione.

Un'esecuzione della applicazione è la seguente, dove sono stati mandati in esecuzione tre **client** e l'ultimo ha comunicato la stringa "fine":



Prova adesso!

Il servizio daytime (udp/13) fornisce data e ora correnti in formato comprensibile da un essere umano: scrivi un **server** che invia data e ora in un pacchetto **UDP** a ogni **client** che gli invia un qualsiasi pacchetto **UDP**, anche vuoto.

Successivamente effettua il conteggio delle volte che un medesimo **client** si è connesso memorizzandolo su di un file: alla decima connessione segnala al **client** che ha "esaurito i bonus" gratuiti e da questo momento il servizio "orario" è a pagamento.

ESERCITAZIONI DI LABORATORIO 6

APPLICAZIONI MULTICAST IN JAVA

Client e server multicast

La comunicazione tra gruppi di processi realizzata mediante **multicasting** (one to many communication):

- un insieme di processi formano un **gruppo di multicast**;
- un messaggio **spedito** da un **processo** a quel gruppo viene recapitato **a tutti gli altri** partecipanti appartenenti al gruppo.

L'implementazione del **multicast** richiede:

- uno schema di indirizzamento dei gruppi;
- un supporto che registri la corrispondenza tra un gruppo e i partecipanti;
- un'implementazione che ottimizzi l'uso della rete nel caso di invio di pacchetti a un gruppo di multicast.

Le **API Multicast** contengono le primitive per far unire un **client** a un gruppo di **multicast** (**join**) e per lasciare il gruppo (**leave**) quando non è più interessato a ricevere i messaggi relativi a quel gruppo.

La libreria **java.net** mette a disposizione la classe **MulticastSocket** per inviare messaggi **multicast** e con i **socket datagram** si può usare il **multicast** per mandare un pacchetto a un insieme di processi con una sola invocazione a **send()**.

I principali metodi della classe **MulticastSocket** sono:

```
joinGroup(InetAddress addr) throws IOException : per unirsi al gruppo
leaveGroup(InetAddress addr) throws IOException : per lasciare il gruppo
send(DatagramPacket p)                        : per inviare un pacchetto
setTimeToLive(int tlive) throws IOException   : per settare il tempo di vita dei pacchetti
                                                inviati sul socket
int getTimeToLive() throws IOException         : ritorna il tempo restante (tra 0 e 255)
```

Client multicast

I **client** che vogliono ricevere messaggi **multicast** devono unirsi a un gruppo specifico e per far questo devono conoscere:

- la porta del **socket** usata dal **server** (per noi la solita **6789**);
- l'indirizzo del gruppo a cui vengono inviati messaggi (per esempio **"225.4.5.6"**).

Vediamo nello specifico come realizzare una classe **client** che si connette a un gruppo e riceve i messaggi a esso destinati:

```

1 import java.net.*;
2 import java.io.*;
3
4 public class MulticastClient
5 {
6     public static void main(String[] args) throws IOException
7     {
8         byte[] bufferIN = new byte[1024];           // buffer di ricezione
9         //parametri del server
10        int porta = 6789;
11        String gruppo = "225.4.5.6";

```

Creiamo un **socket multicast** e uniamo il **client** nello specifico gruppo definito sul **server**:

```

12        // creazione del socket sulla porta
13        MulticastSocket socket = new MulticastSocket(porta);
14        // mi aggiungo al gruppo Multicast
15        socket.joinGroup(InetAddress.getByAddress(gruppo));

```

Creiamo un oggetto datagramma e ci poniamo in attesa dei messaggi del **server**:

```

16        // creo il DatagramPacket e mi metto in ricezione
17        DatagramPacket packetIN = new DatagramPacket(bufferIN, bufferIN.length);
18        socket.receive(packetIN);

```

Sullo schermo visualizziamo il messaggio ricevuto e alcuni parametri della porta del **socket**:

```

19        // Visualizzo i parametri ed i dati ricevuti
20        System.out.println("Ho ricevuto dati di lunghezza: "+packetIN.getLength()
21        + " da : " + packetIN.getAddress().toString()
22        + " porta : " + packetIN.getPort());
23        System.out.write(packetIN.getData(),0,packetIN.getLength());
24        System.out.println();

```

Al termine delle operazioni il **client** si scollega dal gruppo e chiude il **socket**.

```

25        // al termine della ricezione lascio il gruppo
26        socket.leaveGroup(InetAddress.getByAddress(gruppo));
27        // chiudo il socket
28        socket.close();

```

Naturalmente un **client** può continuare a ricevere quanti pacchetti vuole: è sufficiente includere la **receive()** in un ciclo ed eseguirla periodicamente.

Server multicast

Il processo **server** crea il **socket** per una porta e invia, quando vuole, pacchetti **datagram** al gruppo: nel nostro esempio invieremo la data e l'ora a intervalli di un secondo per un determinato tempo prefissato nella variabile conta (per esempio per 20 secondi).

Definiamo “225.4.5.6” come indirizzo di gruppo e apriamo un **socket** sulla porta 6789:

```

1 import java.net.*;
2 import java.io.*;
3 import java.util.*;
4
5 public class MulticastServer
6 {
7     public static void main(String[] args) throws IOException
8     {
9         boolean attivo = true;           // per la ripetizione del servizio
10        byte[] bufferOUT = new byte[1024]; // buffer di spedizione e ricezione
11        int conta = 20;                   // secondi di attività del server
12        int porta = 6789;
13        InetAddress gruppo = InetAddress.getByName("225.4.5.6");

```

Creiamo il **socket**

```

14 // creo il socket multicast
15 MulticastSocket socket = new MulticastSocket();

```

e prepariamo la variabile che conterrà il dato da trasmettere.

```

16 // contenitore per il dato da trasmettere
17 String dString = null;

```

Il ciclo di trasmissione genera il dato, crea un datagramma e lo spedisce al gruppo:

```

18 // ciclo di trasmissione
19 while (attivo)
20 {
21     // come messaggio viene inviata la data e l'ora di sistema
22     dString = new Date().toString();
23     bufferOUT = dString.getBytes();
24     // creo il DatagramPacket
25     DatagramPacket packet;
26     packet = new DatagramPacket(bufferOUT, bufferOUT.length, gruppo, porta);
27     // invio il dato
28     socket.send(packet);

```

Il ciclo di trasmissione viene ripetuto ogni secondo grazie all'attesa del metodo **sleep()** e a ogni iterazione aggiorna il contatore che modifica la condizione di terminazione:

```

29 // introduco un ciclo di attesa di 1 secondo
30 try {
31     Thread.sleep(1000); //attesa di 1000 millisecondi
32 } catch (InterruptedException ie) {
33     ie.printStackTrace();
34 }
35 conta--;
36 if (conta==0){
37     System.out.println("SERVER IN CHIUSURA. Buona serata.");
38     attivo=false;
39 }else{
40     System.out.println("SERVER attivo per altri secondi "+conta);
41 }

```

Alla fine, come ultima istruzione, il **server** chiude la connessione.

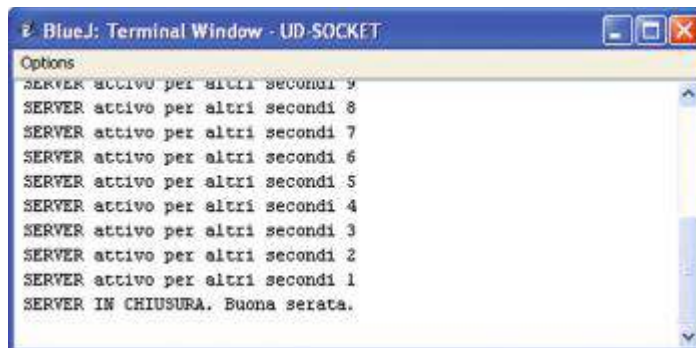
```

46 // alla fine chiudo il socket
47 socket.close();
48 )

```

Una possibile esecuzione è la seguente:

A per il **server**:



```

Options
SERVER attivo per altri secondi 9
SERVER attivo per altri secondi 8
SERVER attivo per altri secondi 7
SERVER attivo per altri secondi 6
SERVER attivo per altri secondi 5
SERVER attivo per altri secondi 4
SERVER attivo per altri secondi 3
SERVER attivo per altri secondi 2
SERVER attivo per altri secondi 1
SERVER IN CHIUSURA. Buona serata.

```

B per il **client**:



```

Options
Ho ricevuto dati di lunghezza: 28 da : /192.168.1.2 porta :15271
Mon Feb 25 17:55:05 CET 2013

```



Prova adesso!

Realizza una applicazione di diffusione in **multicast** della "Divina Commedia": il **server** inizia a trasmettere a intervalli di 1 secondo pacchetti contenenti ciascuno una riga di un file di testo così strutturata:

cantica # numero canto # numero riga # una riga del canto

Il **client** che riceve i datagrammi aspetta l'inizio di un nuovo canto e successivamente scrive un file per ogni canto, con la seguente struttura:

cantica_nrcanto.txt (per esempio *inferno_canto5.txt*).

Il file da trasmettere è anche disponibile nella cartella materiali della sezione del sito www.hoepliscuola.it dedicata a questo volume.

ESERCITAZIONI DI LABORATORIO 7

UN ESEMPIO COMPLETO CON LE JAVA SOCKET: "LA CHAT"

Come esempio completo di applicazione Web realizziamo una semplice chat tra un **server** e un numero N di **client** (per esempio 10 **client**).

Per la gestione della ricezione e dell'invio dei messaggi utilizziamo quindi il meccanismo **multithread**, in quanto non si può prevedere il momento in cui ogni singolo **client** fa richiesta di invio di un nuovo messaggio.

Definiamo un **thread** con il compito di rimanere in ascolto su una determinata porta (la nota 6789) in attesa di connessioni da parte di un **client**; il **server**, a ogni nuova connessione, provvede a istanziare un nuovo oggetto dedicato con il compito di mantenere la connessione **client-server** per tutta la durata del suo collegamento e di occuparsi di ricevere e spedire i messaggi a quel **client**.

Vediamo ora il **thread** che si occupa di ricevere le connessioni dei **client**:

```

1 public class ThreadGestioneServizioChat implements Runnable
2 {
3     private int nrMaxConnessioni;
4     private List lista;
5     private ThreadChatConnessioni[] listaConnessioni;
6     Thread me;
7     private ServerSocket serverChat;
8
9     public ThreadGestioneServizioChat(int numeroMaxConnessioni, List lista)
10    {
11        this.nrMaxConnessioni = nrMaxConnessioni-1;
12        this.lista = lista;
13        this.listaConnessioni = new ThreadChatConnessioni[this.nrMaxConnessioni];
14        me = new Thread(this);
15        me.start();
16    }
17
18
19

```

Al costruttore passiamo come parametro il numero massimo di connessioni e la lista dei messaggi. Vediamo ora il **thread** che si occupa di ricevere le connessioni dei **client**: per prima cosa viene creato un **socket** sulla porta 6789:

```

21 public void run()
22 {
23     boolean continua = true;
24     //instancio la connessione del server per la chat
25     try{
26         serverChat = new ServerSocket(6789);
27     }catch(Exception e){
28         JOptionPane.showMessageDialog(null, "Impossibile instanziare il server");
29         continua = false;
30     }
31

```

Se questa operazione va a buon fine, si procede con la creazione di un **thread** per ogni connessione:

```

23
24     if(continua){
25         //accetto le connessioni chat
26         try {
27             for(int xx=0;xx<nrMaxConnessioni;xx++){
28                 Socket tempo=null;
29                 tempo = serverChat.accept();
30                 listaConnessioni[xx] = new ThreadChatConnessioni(this,tempo);
31             }
32             serverChat.close();
33         }catch(Exception e){
34             JOptionPane.showMessageDialog(null,"Impossibile instanziare server chat");
35         }
36     }
37 } //fine metodo "run"

```

L'elenco delle connessioni viene memorizzato nell'array **listaConnessioni**.
La classe viene completata con un metodo che invia un messaggio a tutti i **client**:

```

47
48     public void spedisciMessaggio(String mex)
49     {
50         //scrivo il mex nella mia lista
51         lista.add(mex);
52         lista.select(lista.getItemCount()-1);
53         //mando il messaggio agli altri
54         for(int xx=0; xx<this.nrMaxConnessioni;xx++){
55             if(listaConnessioni[xx] != null){
56                 listaConnessioni[xx].spedisciMessaggioChat(mex);
57             }
58         }
59     }
60 } //fine classe ThreadGestioneServizioChat

```

Descriviamo ora la classe che viene associata a ogni **client** connesso con il **server**: si ottengono gli stream di I/O e si crea un metodo che permette di spedire i messaggi al **client**.
(Le variabili non sono commentate in quanto i loro identificativi "parlanti" indicano a che cosa servono.)

```

4     public class ThreadChatConnessioni implements Runnable
5     {
6         private ThreadGestioneServizioChat gestoreChat;
7         private Socket client = null;
8         private BufferedReader input = null;
9         private PrintWriter output = null;
10        Thread me;
11
12        public ThreadChatConnessioni(ThreadGestioneServizioChat gestoreChat,Socket client)
13        {
14            this.gestoreChat = gestoreChat;
15            this.client= client;
16            try{
17                this.input = new BufferedReader(new InputStreamReader(client.getInputStream()));
18                this.output = new PrintWriter(this.client.getOutputStream(),true);
19            }catch(Exception e){
20                output.println("Errore nella lettura.");
21            }
22            me = new Thread(this);
23            me.start();
24        }

```


Le operazioni eseguite nel costruttore sono comunque note:

- creazione e inizializzazione dello stream input come oggetto **BufferedReader**;
- creazione e inizializzazione dello stream output come oggetto **PrintWriter**;
- inizializzazione di **gestoreChat** ricevuto come parametro;
- inizializzazione di **Socket client** ricevuto come parametro.

Quindi viene creato un **thread** e mandato in esecuzione.

Vediamo di seguito l'implementazione del metodo `run()`.

```

46 public void run()
47 {
48     while(true){
49         try
50         {
51             String mex=null;
52             //rimango in attesa dei messaggi mandati dal client
53             while({mex = input.readLine()}==null)
54             { }
55             //invoco il metodo del gestoreChat per ripetere a tutti il messaggio ricevuto
56             gestoreChat.spedisciMessaggio(mex);
57         }catch(Exception e)
58         {
59             output.println("Errore nella spedizione del messaggio a tutti.");
60         }
61     }
62 }

```

Completa la classe un semplice metodo che spedisce un messaggio a un singolo **client**.

```

44 public void spedisciMessaggioChat(String messaggio)
45 {
46     try{
47         output.println(messaggio);
48     }catch(Exception e){
49         output.println("Errore nella spedizione del singolo messaggio.");
50     }
51 }
52 }

```

Vediamo ora l'implementazione del servizio **client** della chat.

Il **client** ha solamente il compito di connettersi con il **server**, di rimanere in attesa di messaggi inviati dal **server** e, su richiesta dell'utente, deve inviare nuovi messaggi verso il **server**.

Analizziamo in dettaglio il codice: la classe definisce le variabili private utilizzate da ogni singolo **thread** ► e riceve nel costruttore la lista dei **thread** e i parametri del **server** al quale collegarsi. ▼

```

6 public class ThreadChatClient implements Runnable
7 {
8     private List lista;
9     Thread me;
10    private Socket client;
11    private BufferedReader input=null;
12    private PrintWriter output=null;
13    ...

```

```

14 public ThreadChatClient(List lista, String ipServer, int porta)
15 {
16     this.lista = lista;
17     try{
18         client = new Socket(ipServer,porta);
19         this.input = new BufferedReader(new InputStreamReader(client.getInputStream()));
20         this.output = new PrintWriter(client.getOutputStream(),true);
21     }catch(Exception e){
22         JOptionPane.showMessageDialog(null,"Impossibile connettersi al server");
23     }
24     me = new Thread(this);
25     me.start();
26 }

```

Il metodo `run()` rimane in attesa che l'utente scriva un messaggio, quindi invoca il metodo del `server` affinché il messaggio venga inoltrato a tutti coloro che sono connessi alla chat.

```

26 public void run()
27 {
28     while(true){
29         try
30         {
31             String mex=null;
32             //rimango in attesa dei messaggi mandati dal client
33             while{(mex = input.readLine())==null}
34             { }
35             //invoco il metodo del gestoreChat per ripetere a tutti il messaggio ricevuto
36             gestoreChat.spedisciMessaggio(mex);
37         }catch(Exception e)
38         {
39             output.println("Errore nella spedizione del messaggio a tutti.");
40         }
41     }
42 }

```

Conclude la classe un metodo che invia il messaggio al singolo `client`: ►

```

44 public void spedisciMessaggioChat(String messaggio){
45     try{
46         output.println(messaggio);
47     }catch(Exception e){
48     }
49 }

```

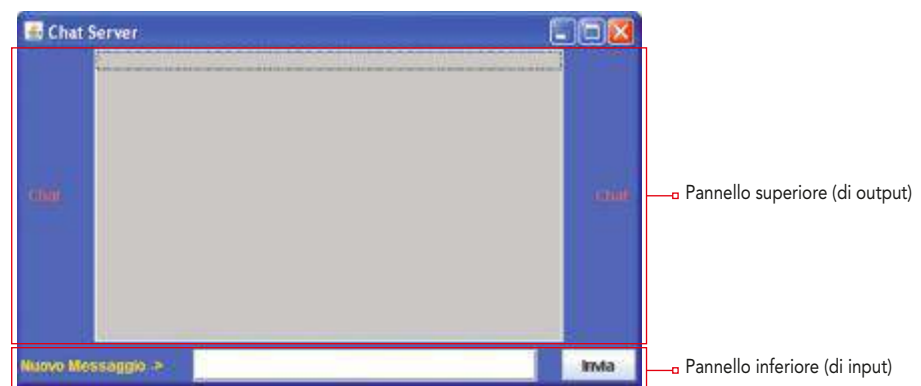
Riportiamo il codice che realizza l'interfaccia grafica solamente per il `server`, poiché è pressoché identica a quella del `client`:

```

1 public class ChatServer extends JFrame
2 {
3     public ChatServer()
4     {
5         super("Chat Server");
6         this.setSize(new Dimension(500,300)); // setto la grandezza della finestra
7         this.setLocationRelativeTo(null); // la metto al centro dello schermo
8         this.setEnabled(true); // setto la proprietà enable
9         this.setBackground(Color.blue); // setto il colore di sfondo
10        //creo il pannello per l'inserimento e la visualizzazione dei messaggi
11        PannelloChatServer pan = new PannelloChatServer();
12        this.getContentPane().add(pan);
13        this.setVisible(true);
14    }
15 }

```

Nella classe viene definito un oggetto della classe



L'interfaccia è costituita da due parti:

- un pannello superiore (di output) dove vengono visualizzati tutti i messaggi;
- un pannello inferiore (di input) dove l'utente scrive il proprio messaggio per inviarlo al **server** (e quindi agli altri utenti collegati).

La parte superiore del pannello è realizzata con questo segmento di codice: ►

```

29 public PannelloChatServer()
30 {
31     super();
32     this.setBackground(new Color(50, 100, 255));
33     // pannello superiore: lista messaggi
34     JPanel panlista = new JPanel(new BorderLayout(20,5));
35     panlista.setBackground(new Color(50, 100, 255));
36     lista = new List();
37     lista.setBackground(Color.lightGray);
38     lista.setSize(100,50);
39     lista.setVisible(true);
40     // scritte interali
41     JLabel chat1 = new JLabel(" Chat ",JLabel.CENTER);
42     chat1.setForeground(new Color(200,100,100));
43     JLabel chat2 = new JLabel(" Chat ",JLabel.CENTER);
44     chat2.setForeground(new Color(200,100,100));
45     // aggiungiamo gli oggetti sul pannello
46     panlista.add(chat1,BorderLayout.WEST);
47     panlista.add(lista,BorderLayout.CENTER);
48     panlista.add(chat2,BorderLayout.EAST);

```

La parte inferiore del pannello è realizzata con questo segmento di codice: ►

```

49 //pannello inserimento nuovo messaggio
50 JPanel nuovoMex = new JPanel(new BorderLayout(20,5));
51 nuovoMex.setBackground(new Color(50, 100, 255));
52
53 JLabel labNuovo = new JLabel("Nuovo Messaggio -> ",JLabel.CENTER);
54 labNuovo.setForeground(new Color(255,255,0));
55
56 JTextField textNuovo = new JTextField("");
57
58 JButton buttonInvia = new JButton("Invia");
59 buttonInvia.addActionListener(this);
60 // aggiungiamo gli oggetti sul pannello
61 nuovoMex.add(labNuovo,BorderLayout.WEST);
62 nuovoMex.add(textNuovo,BorderLayout.CENTER);
63 nuovoMex.add(buttonInvia,BorderLayout.EAST);
64
65 this.setLayout(new BorderLayout(0,5));
66 add(panlista,BorderLayout.CENTER);
67 add(nuovoMex,BorderLayout.SOUTH);
68
69 connetti();
70 } //fine costruttore classe PannelloChat

```

Concludono la classe due metodi, rispettivamente per avviare il **server** e predisporre la lista delle possibili connessioni (10 utenti massimo) e la gestione dell'evento sul pulsante di invio che inoltra il messaggio. Infatti anche l'utente che avvia il **server** può, di fatto, comunicare con gli altri utenti. ►

```

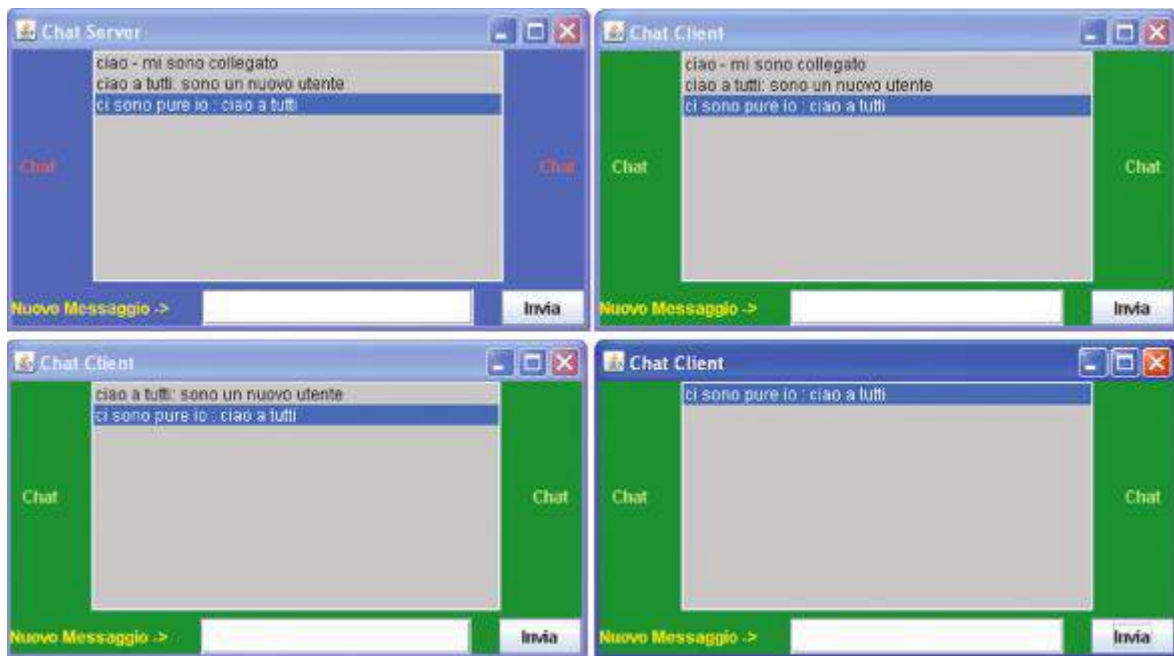
71 public void connetti()
72 {
73     //instancio il Thread per le connessioni : numero massimo giocatori = 10
74     gestioneServizio = new ThreadGestioneServizioChat(10,lista);
75 }
76
77 public void actionPerformed(ActionEvent e)
78 {
79     String bottone = e.getActionCommand();
80     if(bottone.equals("Invia"))
81     {
82         gestioneServizio.spedisciMessaggio(textNuovo.getText());
83         textNuovo.setText("");
84     }
85 }

```

Un esempio di esecuzione è il seguente:



In questa prima situazione abbiamo avviato il **server** e un **client** e inviato un primo messaggio. Aggiungiamo ora altri due **client** e inviamo un messaggio da ciascuno di essi: possiamo osservare come la comunicazione viene inviata a tutte le finestre.



L'unico problema che possiamo evidenziare è che non sono presenti i nomi dei mittenti, e quindi "non si sa di chi sono i messaggi"!



Prova adesso!

Modifica l'esempio sopra descritto aggiungendo una form di login in modo che l'utente inserisca il proprio nome e questo venga visualizzato nella sezione di output vicino a ogni messaggio, in modo da riconoscerne la "paternità".

Quindi fai le opportune modifiche per poter utilizzare questa applicazione in rete, creando un "sniffer" che ricerca sulla rete l'indirizzo **IP** del **server** che offre questo servizio sulla porta 6789.