Esercitazione di Sistemi Distribuiti e Pervasivi

Socket e programmazione di rete

Gabriele Civitarese
gabriele.civitarese@unimi.it

EveryWare Lab
Università degli Studi di Milano
-
Docente: Claudio Bettini

Slide realizzate a partire da versioni precedenti di Letizia Bertolaja, Sergio Mascetti, Dario Freni e Claudio Bettini

Copyright

Alcune slide di questo corso sono in parte personalizzate da quelle fornite dall'autore del testo di riferimento e distribuite dalla casa editrice (Distributed Systems: Principles and Paradigms, A. S. Tanenbaum, M. Van Steen, Prentice Hall, 2007). Tutte le altre slide sono invece di proprieta del docente. Tutte le slide sono soggette a diritto d'autore e quindi non possono essere ri-distribuite senza consenso. Lo stesso vale per eventuali registrazioni o videoregistrazioni delle lezioni.

Some slides for this course are partly adapted from the ones distributed by the publisher of the reference book for this course (Distributed Systems: Principles and Paradigms, A. S. Tanenbaum, M. Van Steen, Prentice Hall, 2007). All the other slides are from the teacher of this course. All the material is subject to copyright and cannot be redistributed without consent of the copyright holder. The same holds for audio and video-recordings of the classes of this course.

Argomenti del corso

- Socket e programmazione di rete
- Server multithread e gestione della concorrenza nelle applicazioni di rete
- Remote Procecure Call
- Server REST

Suddivisione lezioni

Lezioni "teoriche"

- 1ª lezione:
  - Socket TCP e UDP
  - Sviluppo client-server (server iterativi/concorrenti)
- 2ª lezione:
  - Sincronizzazione
  - Segnalazione
- 3ª lezione:
  - Comunicazione tra processi
  - JSON, XML e ProtocolBuffer
  - Remote Procedure Call e il framework *grpc*
- 4ª lezione:
  - Server REST
  - Puntatori a tool moderni per sviluppo sistemi distribuiti
  - Presentazione del progetto

Suddivisione lezioni (2)

Supporto allo sviluppo del progetto

- 1ª lezione:
  - Progettazione del sistema e del protocollo
  - Implementazione comunicazione
- 2ª lezione:
  - Analisi dei problemi di sincronizzazione
- 3ª lezione:
  - Realizzazione del progetto per la parte Server REST
- 4ª lezione
  - Conclusione progetto
  - Valutazione

Materiale

- I lucidi ed eventuale altro materiale oltre a comunicazioni urgenti saranno disponibili alla pagina `http://everywarelab.di.unimi.it/sdp`
- Libri di testo (consigliati):
  - A. Tanenbaum M. van Steen: "Distributed Systems", Pearson (per i concetti teorici)
  - E. R. Harold: "Java Network Programming", O'Reilly
  - D. Maggiorini: "Introduzione alla programmazione client-server", Pearson (in italiano)
- Vario materiale in rete. Ad esempio:
  - Java tutorials

Progetto ed esame

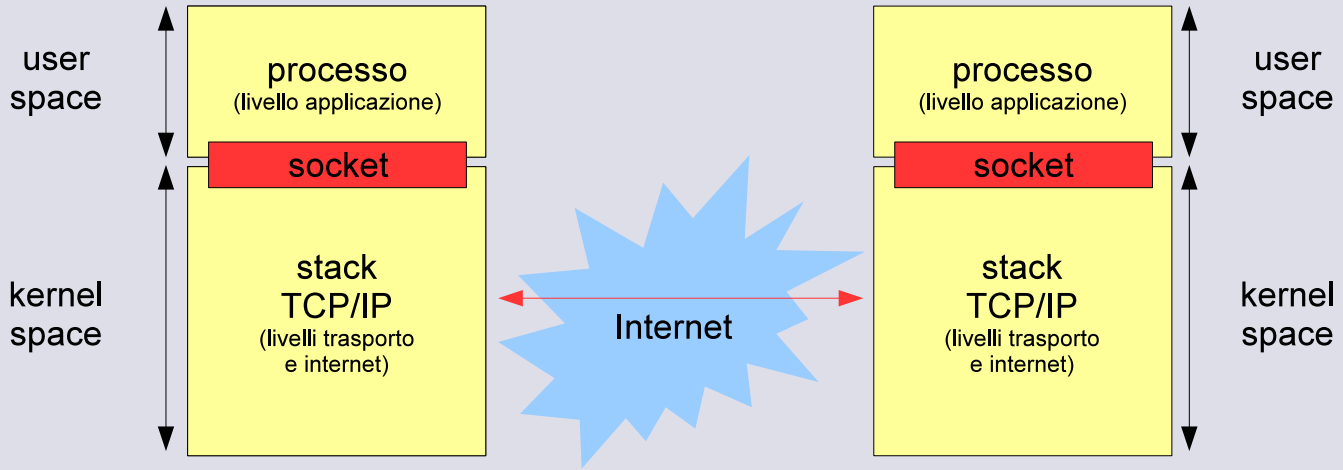
- Il testo del progetto verrà presentato durante l'ultima lezione "teorica" di laboratorio
  - Durante le successive esercitazioni verrà fornito supporto per la sua realizzazione
- Una volta ottenuto il voto di teoria, il progetto potrà essere presentato ad una delle sessioni previste (controllate il sito del corso!)
- La discussione del progetto prevede:
  - Esecuzione dello stesso
  - Spiegazione del codice
  - Domande di teoria (inerenti alle parti di laboratorio)

Motivazione

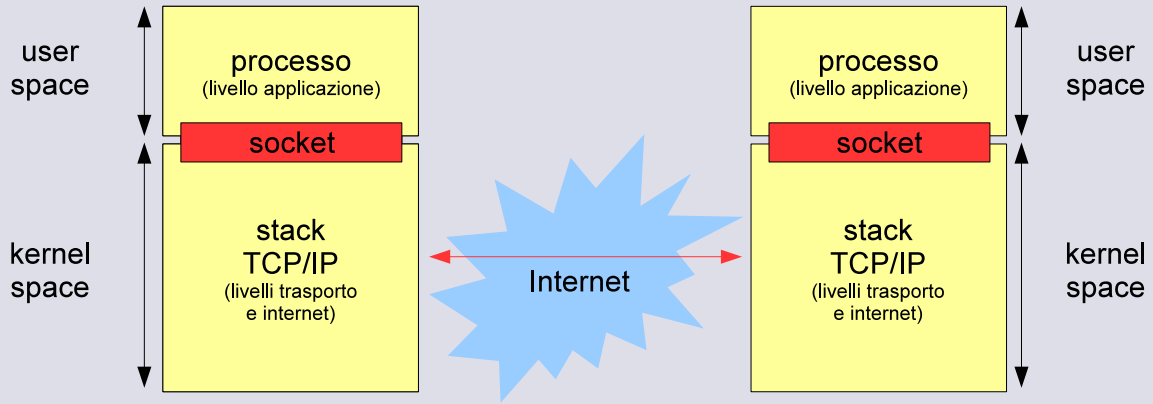
Perchè tutto a basso livello nel 2019?

- Queste lezioni focalizzeranno l'hands-on su: comunicazione tra processi, concorrenza e sincronizzazione a basso livello in Java
- Si parte dalle socket e si arriva alle primitive di sincronizzazione
- Ci sono sicuramente tool più recenti e ad alto livello ma...
  - ...lo scopo del corso di laboratorio è di comprendere da vicino i problemi di sincronizzazione
  - ...verranno comunque forniti puntatori a tecnologie più recenti

Una **socket** è un endpoint di una comunicazione a doppio senso tra due processi in rete. Ogni socket è legata ad un numero di porta, in modo tale che il livello di trasporto (TCP o UDP) possa identificare l'applicazione per i quali dati sono destinati.



Comunicazione tra processi



Le socket si suddividono in:

- **listening socket**, in attesa di connessioni entranti
- **established socket**, riferimento ad una connessione attiva (TCP)
- **datagram socket**, riferimento per invio/ricezione pacchetti senza connessione (UDP)

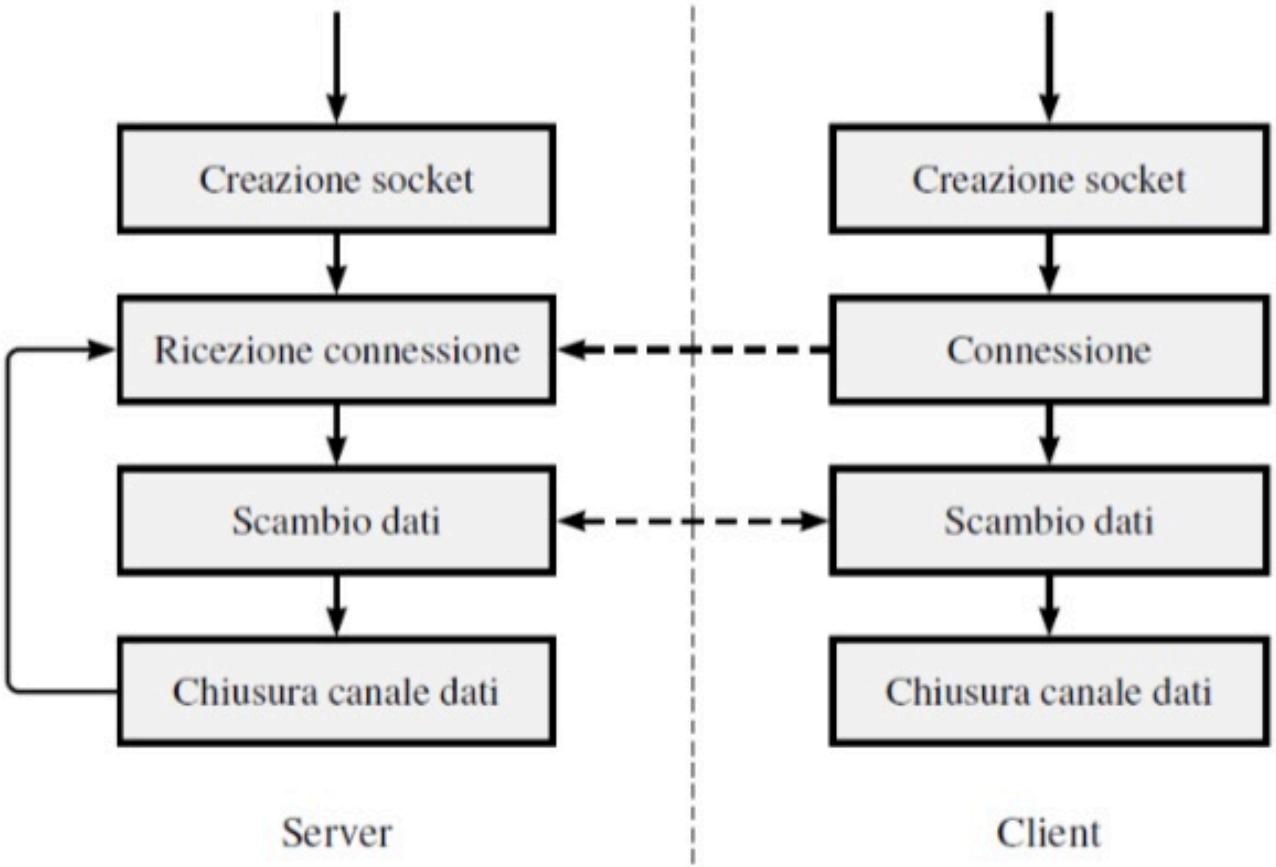
Comunicazione client-server TCP (connection oriented)

Client

- crea una socket specificando indirizzo del destinatario e porta del servizio
- stabilita la connessione, la **established socket** creata viene utilizzata come interfaccia di comunicazione

Server

- crea una **listening socket** specificando una porta di servizio
- all'arrivo di una connessione in entrata, viene creata una nuova **established socket**
- la comunicazione avviene mediante la **established socket** del server e quella del client


Creazione socket
Ricezione connessione
Scambio dati
Chiusura canale dati
Server
Creazione socket
Connessione
Scambio dati
Chiusura canale dati
Client

Il package java.net

- Classi per il networking
  - classi client e server TCP
  - classe UDP Datagram
  - classe Multicast Datagram
- Caratteristiche
  - Indipendenza dal sistema operativo
  - Object-oriented: conveniente, riutilizzabile
  - Gestione degli errori: via eccezioni

Le classi principali in java.net

- java.net.Socket
- java.net.ServerSocket
- java.net.DatagramSocket
- java.net.DatagramPacket
- java.net.MulticastSocket
- java.net.InetAddress
- java.net.URL
- java.net.URLConnection
- java.net.URLEncoder
- java.net.HttpURLConnection

La classe Socket

- Alcuni costruttori
  - `Socket(InetAddress, int) // connects to the address and port`
  - `Socket(String, int) // connects to the named host and port`
  - `Socket(InetAddress, int, InetAddress, int) // can specify local port`
- Alcuni metodi
  - `getInputStream() // returns an InputStream for the socket`
  - `getOutputStream() // returns an OutputStream for the socket`
  - `getInetAddress() // return the remote address`
  - `getPort() // return the remote port`
  - `getLocalAddress() // return the local address`
  - `getLocalPort() // return the local port`
  - `close() // close the connection`

La classe ServerSocket

- Alcuni costruttori
  - `ServerSocket(int) // create a server socket on a specified port`
  - `ServerSocket(int, int) // can specify backlog length`
  - `ServerSocket(int, int, InetAddress) // can specify local address`
- Alcuni metodi
  - `accept() // listen for a connection, blocked until connection is made`
  - `getInetAddress() // returns the local address`
  - `getLocalPort() // returns the local port`
  - `close() // close this socket`

Connessione a un passo

```
//tramite hostname
Socket s = new Socket("ewserver.di.unimi.it", 80);
//tramite indirizzo IP
Socket s = new Socket("159.149.145.200",80);
```

Uso di *connect*()

```
//creazione indirizzo per la socket tramite oggetto
InetSocketAddress isa = new InetSocketAddress("ewserver.di.unimi.it",80);
//creazione dell'oggetto socket non connesso ad alcun server
Socket s = new Socket();
//connessione esplicita
s.connect(isa);
```

Listening

```
//bind sulla porta 80
ServerSocket serverSocket = new ServerSocket(80);

//bloccante. crea established socket con un singolo client
Socket s = serverSocket.accept();
```

- Il server si blocca su *accept*() fino a che un client non effettua una connessione
- A connessione ricevuta, viene creata una established socket e il flusso di esecuzione prosegue
- Connessioni multiple dei client vengono messi in una coda ed accettate una ad una

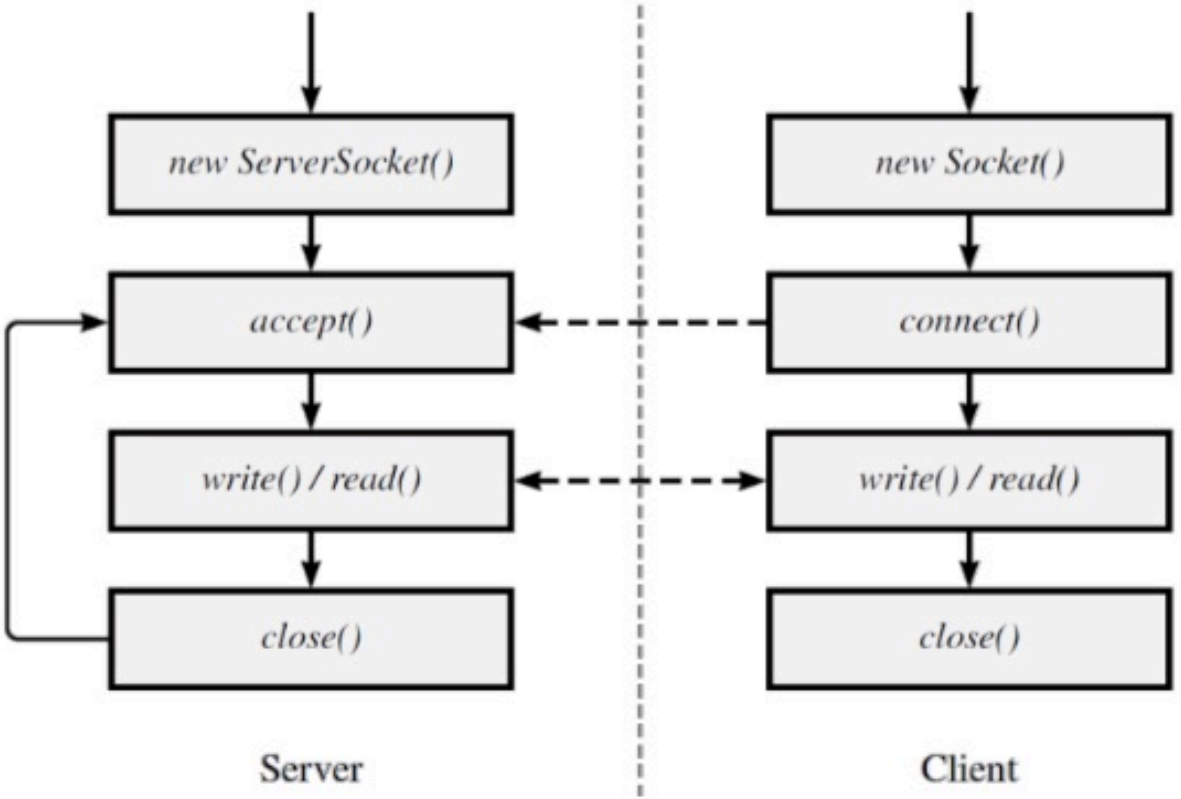
Come comunicano le socket?

- La classe Socket prevede due metodi distinti:
  - *getInputStream*(): stream di tipo *InputStream* per riceve i dati dalla socket (**bloccante** !!!)
  - *getOutputStream*(): stream di tipo *OutputStram* per scrivere sulla socket
- Sono due canali monodirezionali
- Possono essere wrappati facilmente con classi per gestire stream di input/output:
  - BufferedReader
  - BufferedWriter
  - DataOutputStream
  - PrintWriter
  - ...

Chiusura

- È sempre buona norma effettuare la chiusura esplicita delle socket
- Per questo scopo si usa il metodo *close*()
  - Sia per Socket che per ServerSocket
- Homework: scoprire cosa succede a scrivere/leggere da socket chiuse.
  - Fondamentale gestire bene le eccezioni!

Per riassumere...



Semplice server iterativo - Esercizio

Esercizio guidato: UpperCaseServer

- il client legge una riga da tastiera (stream `inFromUser`) e la invia al server via socket (stream `outToServer`)
- il server legge la riga dalla socket
- il server converte la riga in maiuscolo e la invia in risposta al client
- il client legge dal server la nuova riga (stream `inFromServer`) e la stampa a video

TCPClient.java

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

class TCPClient {
        public static void main(String argv[]) throws Exception {
                String sentence;
                String modifiedSentence;

                /* Inizializza l'input stream (da tastiera) */
                BufferedReader inFromUser =
                        new BufferedReader(new InputStreamReader(System.in));

                /* Inizializza una socket client, connessa al server */
                Socket clientSocket = new Socket("localhost", 6789);

                /* Inizializza lo stream di output verso la socket */
                DataOutputStream outToServer =
                        new DataOutputStream(clientSocket.getOutputStream());
```

TCPClient.java(2)

```
                /* Inizializza lo stream di input dalla socket */
                BufferedReader inFromServer =
                        new BufferedReader(new
                                        InputStreamReader(clientSocket.getInputStream()));

                /* Legge una linea da tastiera */
                sentence = inFromUser.readLine();

                /* Invia la linea al server */
                outToServer.writeBytes(sentence + '\n');

                /* Legge la risposta inviata dal server (linea terminata da \n) */
                modifiedSentence = inFromServer.readLine();

                System.out.println("FROM SERVER: " + modifiedSentence);

                clientSocket.close();
        }
}
```

TCPServer.java

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

class TCPServer {

        public static void main(String argv[]) throws Exception
        {
                String clientSentence;
                String capitalizedSentence;

                /* Crea una "listening socket" sulla porta specificata */
                ServerSocket welcomeSocket = new ServerSocket(6789);

                while(true) {
                        /*
                         * Viene chiamata accept (bloccante).
                         * All'arrivo di una nuova connessione crea una nuova
                         * "established socket"
                         */
                        Socket connectionSocket = welcomeSocket.accept();
```

TCPServer.java(2)

```
                    /* Inizializza lo stream di input dalla socket */
                    BufferedReader inFromClient =
                            new BufferedReader(new
                                        InputStreamReader(connectionSocket.getInputStream()));

                    /* Inizializza lo stream di output verso la socket */
                    DataOutputStream  outToClient =
                            new DataOutputStream(connectionSocket.getOutputStream());

                    /* Legge una linea (terminata da \n) dal client */
                    clientSentence = inFromClient.readLine();

                    capitalizedSentence = clientSentence.toUpperCase() + '\n';

                    /* Invia la risposta al client */
                    outToClient.writeBytes(capitalizedSentence);

                }
        }
}
```

Comunicazione client-server UDP (connection-less)

Client

- crea una **datagram socket**
- prepara un pacchetto datagram contenente indirizzo server e porta di destinazione
- invia il pacchetto attraverso la **datagram socket**
- attende il pacchetto di risposta dal server sulla **datagram socket**

Server

- crea una **datagram socket** specificando una porta di ascolto
- attende un pacchetto in entrata dal client sulla **datagram socket**
- estrae indirizzo e porta del client mittente dal pacchetto in entrata
- prepara un pacchetto datagram di risposta specificando indirizzo client e porta di destinazione
- invia il pacchetto attraverso la **datagram socket**
- si rimette in attesa di altri pacchetti

La classe DatagramSocket

- Alcuni costruttori
  - `DatagramSocket() // binds to any available port`
  - `DatagramSocket(int) // binds to the specified port`
  - `DatagramSocket(int, InetAddress) // can specify a local address`
- Alcuni metodi
  - `getLocalAddress() // returns the local address`
  - `getLocalPort() // returns the local port`

Client UDP (UpperCaseServer)

UDPClient.java

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

class UDPClient {
        public static void main(String args[]) throws Exception {
                /* Inizializza l'input stream (da tastiera) */
                BufferedReader inFromUser =
                        new BufferedReader(new InputStreamReader(System.in));

                /* Crea una datagram socket */
                DatagramSocket clientSocket = new DatagramSocket();

                /* Ottiene l'indirizzo IP dell'hostname specificato
                 * (contattando eventualmente il DNS) */
                InetAddress IPAddress = InetAddress.getByName("localhost");

                byte[] sendData = new byte[1024];
                byte[] receiveData = new byte[1024];

                String sentence = inFromUser.readLine();
                sendData = sentence.getBytes();
```

UDPClient.java (2)

```
                /* Prepara il pacchetto da spedire specificando
                 * contenuto, indirizzo e porta del server */
                DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
                                                               IPAddress, 9876);

                /* Invia il pacchetto attraverso la socket */
                clientSocket.send(sendPacket);

                /* Prepara la struttura dati usata per contenere il pacchetto in ricezione */
                DatagramPacket receivePacket =
                        new DatagramPacket(receiveData, receiveData.length);

                /* Riceve il pacchetto dal server */
                clientSocket.receive(receivePacket);

                String modifiedSentence = new String(receivePacket.getData());

                System.out.println("FROM SERVER:" + modifiedSentence);
                clientSocket.close();
        }
}
```

UDPServer.java

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

class UDPServer {
        public static void main(String args[]) throws Exception {
                /* Inizializza la datagram socket specificando la porta di ascolto */

                DatagramSocket serverSocket = new DatagramSocket(9876);

                byte[] receiveData = new byte[1024];
                byte[] sendData  = new byte[1024];

                while(true)
                {

                        /* Prepara la struttura dati usata per contenere il pacchetto in ricezione */
                        DatagramPacket receivePacket =
                                new DatagramPacket(receiveData, receiveData.length);

                        /* Riceve un pacchetto da un client */
                        serverSocket.receive(receivePacket);
```

UDPServer.java

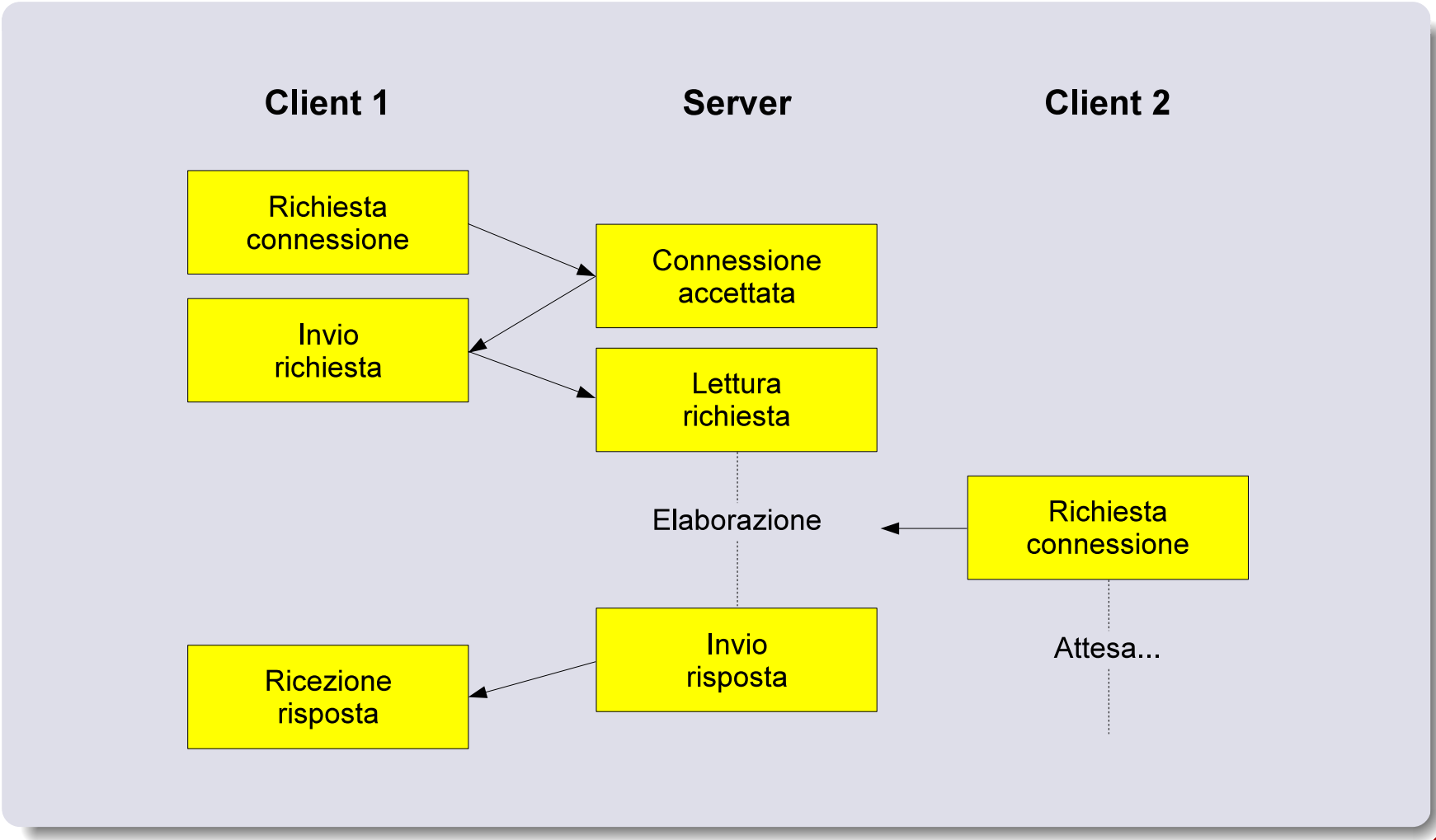
```
                    String sentence = new String(receivePacket.getData());

                    /* Ottiene dal pacchetto informazioni sul mittente */
                    InetAddress IPAddress = receivePacket.getAddress();
                    int port = receivePacket.getPort();

                    String capitalizedSentence = sentence.toUpperCase();

                    sendData = capitalizedSentence.getBytes();

                    /* Prepara il pacchetto da spedire specificando
                     * contenuto, indirizzo e porta del destinatario */
                    DatagramPacket sendPacket =
                            new DatagramPacket(sendData, sendData.length, IPAddress,
                                    port);

                    /* Invia il pacchetto attraverso la socket */
                    serverSocket.send(sendPacket);
                }
            }
}
```

Esercizio I

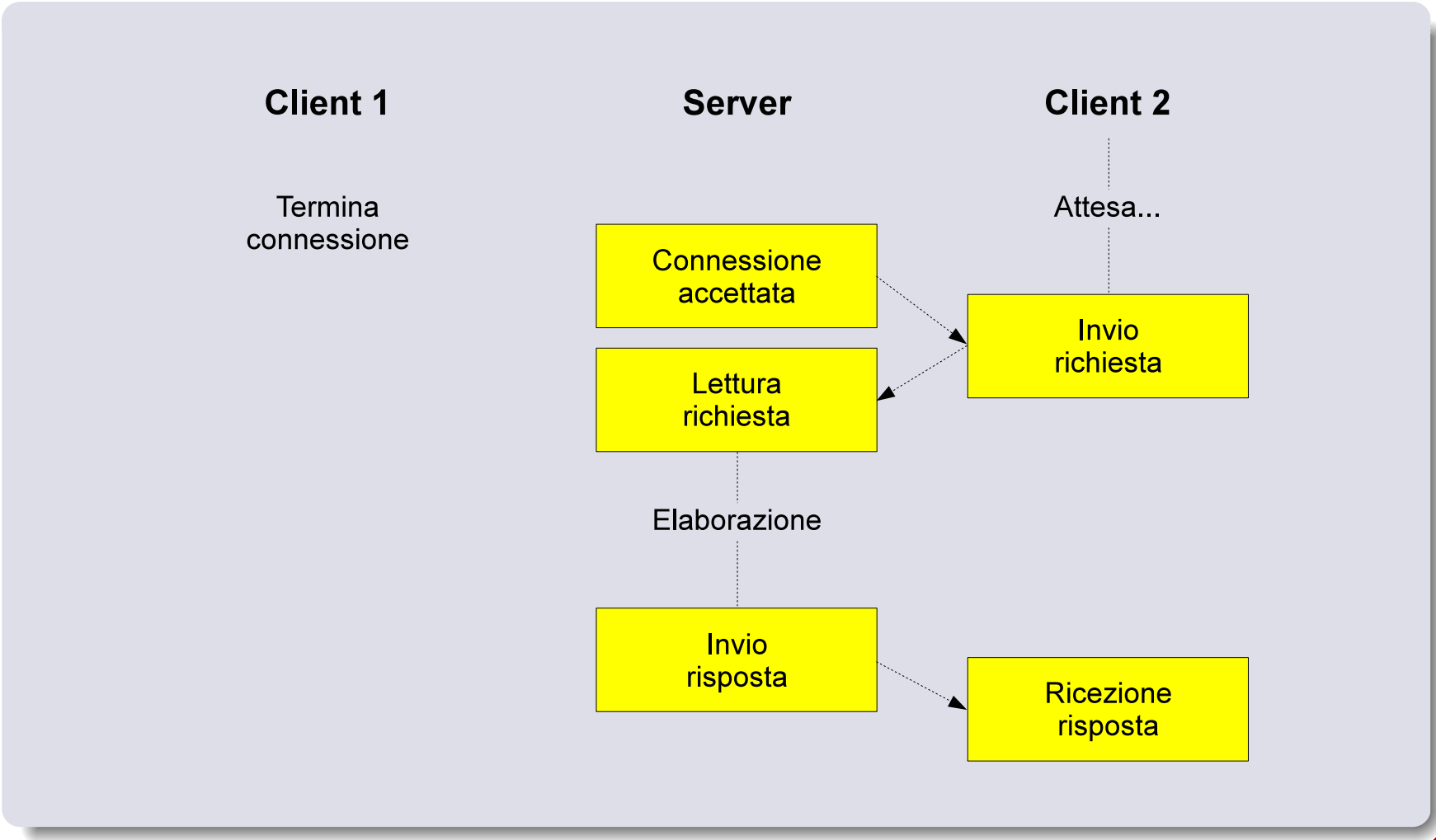
Realizzare un servizio TCP per effettuare le somme

- Il server (iterativo):
  - deve leggere la porta per mettersi in ascolto da riga di comando
  - deve stampare a monitor l'indirizzo e la porta dei client che si connettono
  - riceve due interi dal client, effettua la somma e risponde col risultato
- Il client:
  - deve leggere l'indirizzo e la porta del server da riga di comando
  - deve leggere due numeri da standard input e inviarli al server
  - riceve e stampa la risposta del server
- Gestire correttamente tutte le possibili eccezioni
- Realizzare la versione UDP

Server iterativi: cosa succede (1)



Server iterativi: cosa succede (2)



Una piccola modifica al server...

```
DataOutputStream  outToClient =
        new DataOutputStream(connectionSocket.getOutputStream());

clientSentence = inFromClient.readLine();

/* Attende 10 secondi */
Thread.sleep(10000);

capitalizedSentence = clientSentence.toUpperCase() + '\n';

outToClient.writeBytes(capitalizedSentence);
```

- Qual é il problema lampante?
- Come possiamo risolverlo?

Ripasso: Cosa sono i thread?

- Sequenze di esecuzione distinte all'interno dello stesso processo
- Condividono lo stesso spazio di memoria
- Schedulati come se fossero processi
- Utili per applicazioni concorrenti:
  - Eventi asincroni
  - Overlapping di I/O e computazione
  - ...
- ...spero che vi ricordiate le lezioni di Sistemi Operativi!

Primo Modo: Ereditando Thread

- Creazione del thread:
  - Si estende la classe `Thread` definendo un costruttore che prende come argomenti i riferimenti alle strutture dati alle quali il thread dovrà accedere.
  - Si ridefinisce il metodo `run()` perché il thread possa eseguire il suo specifico compito.
- Invocazione del thread:
  - Si crea un'istanza del thread.
  - Si chiama il metodo `start()`.

Thread in Java (2)

Secondo Modo: Implementando Runnable

- Creazione del thread:
  - Si implementa l'interfaccia Runnable definendo un costruttore che prende come argomenti i riferimenti alle strutture dati alle quali il thread dovrà accedere.
  - Si implementa il metodo `run()` perché il thread possa eseguire il suo specifico compito.
- Invocazione del thread:
  - Si crea un'istanza di un oggetto Thread, passando al suo costruttore un'istanza della classe che implementa Runnable.
  - Si chiama il metodo `start()`.

Metodi a confronto

Ereditando Thread

```
public class MyThread extends Thread{
...
public void run(){
...
```

Implementando Runnable

```
public class RunnableThread implements Runnable{
...
public void run(){
...
```

Chiamata ereditando Thread

```
MyThread thread = new MyThread();
thread.start();
```

Chiamata implementando Runnable

```
Thread thread = new Thread(new RunnableThread());
thread.start();
```

Thread in Java (3)

- Dopo aver creato ed avviato l'esecuzione di un thread ci si trova con due esecuzioni in due punti diversi nel codice del programma.
  - `run()` nel thread
  - Il punto subito dopo la chiamata di `start()`.
- Il thread (processo) padre può avere un riferimento all'oggetto del thread figlio.
- È possibile utilizzare quel riferimento per manipolare i thread in vari modi.

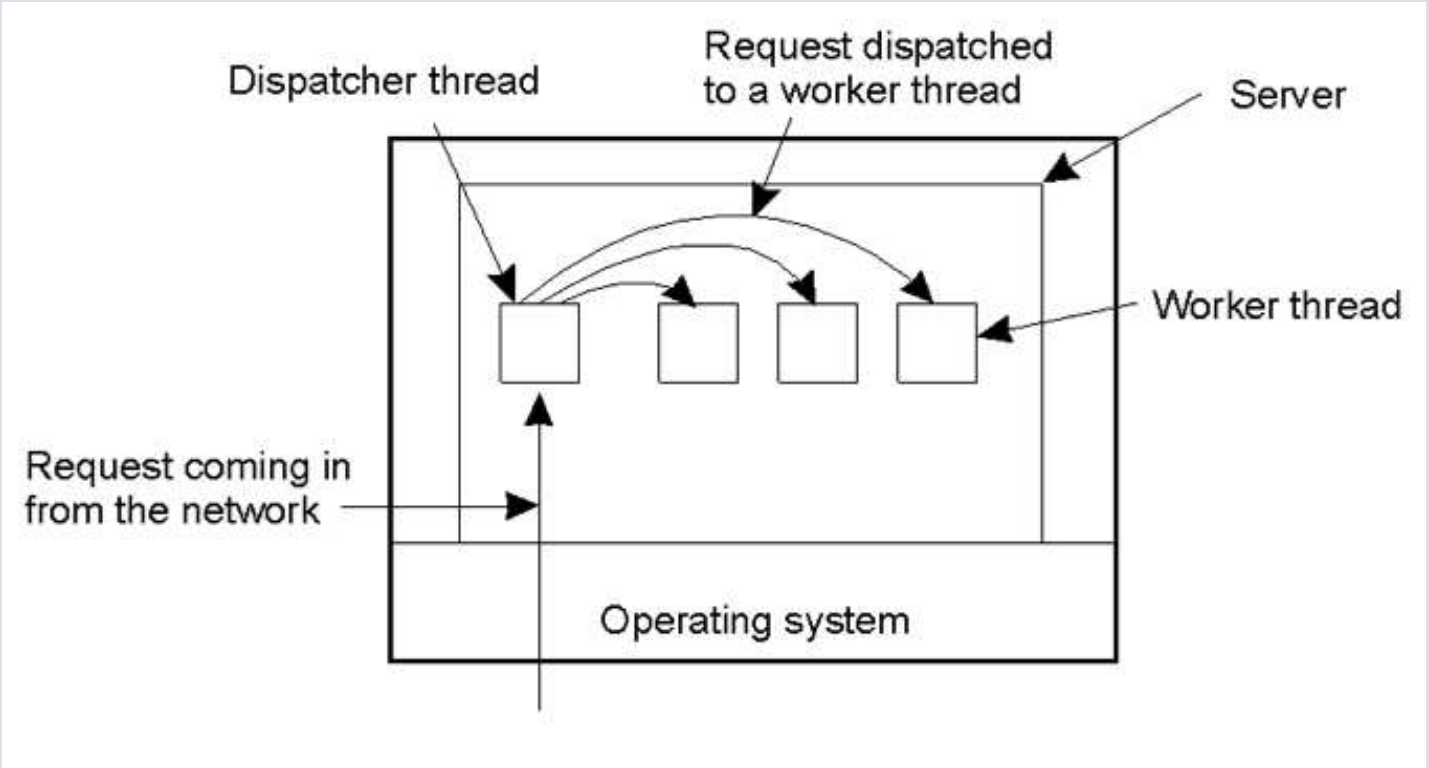
Nota bene

Una volta che il metodo `run()` termina, il thread termina anch'esso e non é possibile farlo ripartire.

Server concorrenti

- Richieste da parte di diversi client possono arrivare in modo concorrente alla porta alla quale il server è in ascolto.
- Nei server iterativi, le richieste vengono accodate e il server deve accettarle e servire la relativa richiesta in modo sequenziale.
- Soluzione: Il server può gestire più connessioni simultaneamente attraverso l'uso dei thread, lanciando un thread per ciascuna connessione con un client.

Modello dispatcher/worker



MultiServer.java

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

class TCPMultiServer {

        public static void main(String argv[]) throws Exception
        {
                ServerSocket welcomeSocket = new ServerSocket(6789);

                while(true) {
                        Socket connectionSocket = welcomeSocket.accept();

                        /* Creazione di un thread e passaggio della established socket */
                        TCPServerThread theThread =
                                new TCPServerThread(connectionSocket);

                        /* Avvio del thread */
                        theThread.start();
                }
        }
}
```

ServerThread.java

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

public class TCPServerThread extends Thread {
        private     Socket connectionSocket = null;
        private     BufferedReader inFromClient;
        private     DataOutputStream  outToClient;

        /* L'argomento del costruttore e' una established socket */
        public TCPServerThread(Socket s) {
                connectionSocket = s;

                try{
                        inFromClient =
                                new BufferedReader(
                                        new InputStreamReader(connectionSocket.getInputStream()));

                        outToClient =
                                new DataOutputStream(connectionSocket.getOutputStream());
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
```

ServerThread.java (2)

```
        public void run() {
                String clientSentence;
                String capitalizedSentence;

                try {

                        clientSentence = inFromClient.readLine();

                        capitalizedSentence = clientSentence.toUpperCase() + '\n';

                        outToClient.writeBytes(capitalizedSentence);

                        connectionSocket.close();

                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
}
```

Esercizio II

Realizzare un servizio TCP per effettuare calcoli secondo la notazione polacca inversa.

- Il client:
  - trasmette in sequenza: due numeri interi e l'operazione da effettuare (chiesti da linea di comando)
  - ogni volta che invia un parametro, attende la conferma dal server
    - In caso di esito negativo il parametro deve essere chiesto nuovamente, altrimenti il programma prosegue
  - Riceve il risultato finale dal server e lo stampa
- Il server (multithread):
  - deve assicurarsi che il protocollo venga rispettato
  - se il client invia un parametro "sbagliato" (e.g. stringa al posto di numero intero), chiede al client di ri-trasmetterlo
  - alla fine computa il risultato e lo trasmette al client