

# Sistemi Distribuiti

Laurea in Informatica

Flavio De Paoli

[flavio.depaoli@unimib.it](mailto:flavio.depaoli@unimib.it)

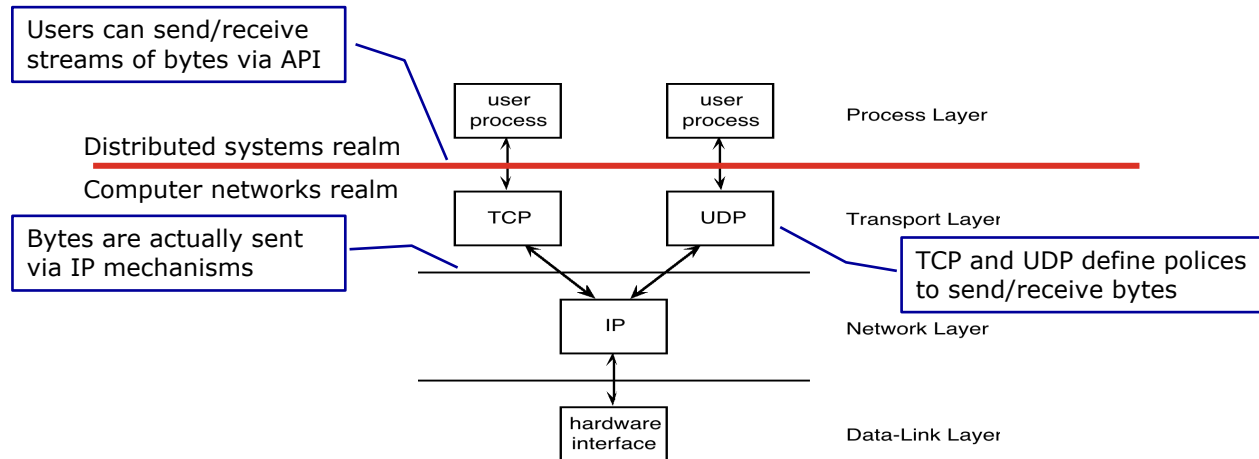
●●● **INSIDE&S Lab** ●●●  
<http://inside.disco.unimib.it/>

# Stream-oriented communication

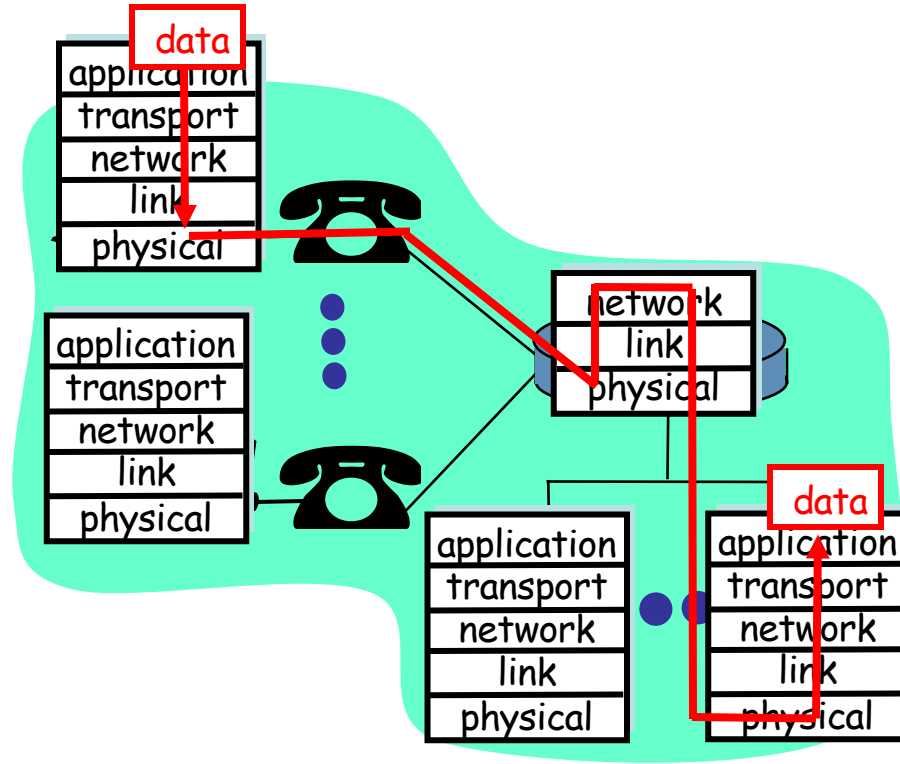
## Le socket

●●● INSIDE&S Lab ●●●  
<http://inside.disco.unimib.it/>

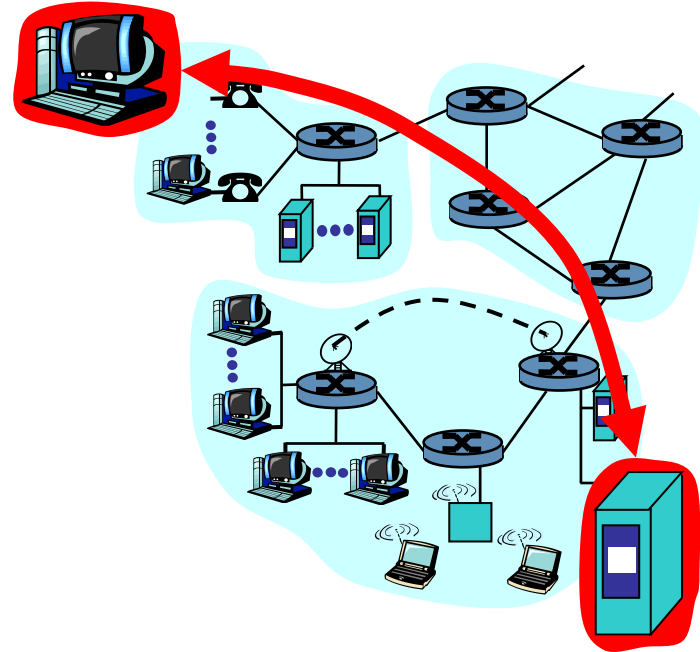
- Breve ripasso del modello ISO/OSI per TCP/IP
- Identificazione dei processi
  - Indirizzi IP e Porte
- L'interfaccia API per le socket
- Le socket in Java
- I modelli architetturali
  - Iterativo
  - Concorrente mono processo
  - Concorrente multi processo



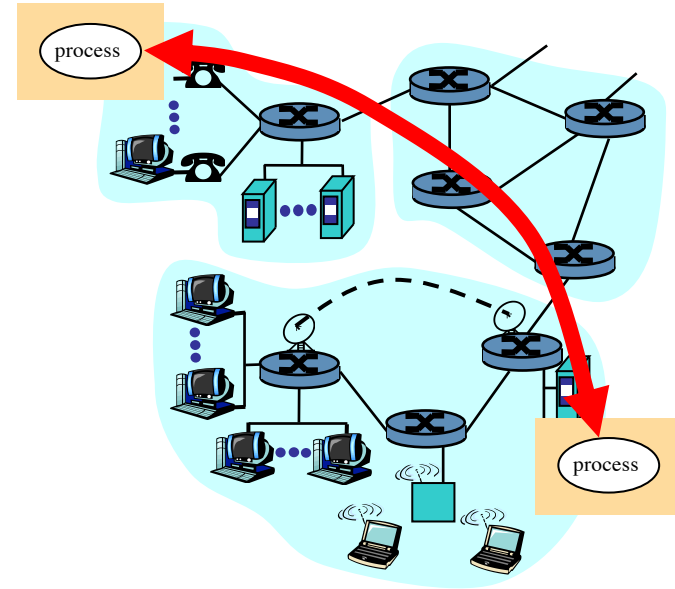
# Layering: physical communication



- end system (hosts)
  - Ospitano i processi che eseguono le applicazioni
  - es., WWW, email
  - "edge of network"
- modello client/server
  - client host requests, receives service from server
  - e.g., WWW client (browser)/ server; email client/server
- modello peer-to-peer
  - interazione simmetrica tra host
  - es.: teleconferenza, Gnutella



- **end system (hosts)**
  - Ospitano i processi che eseguono le applicazioni
  - es., WWW, email
  - "edge of network"
- **modello client/server**
  - client host requests, receives service from server
  - e.g., WWW client (browser)/ server; email client/server
- **modello peer-to-peer**
  - interazione simmetrica tra host
  - es.: teleconferenza, Gnutella



- I programmi vengono eseguiti dai processi
  - Programma = sequenza di istruzioni eseguibili dalla “macchina”
- I processi sono entità gestite dal Sistema Operativo
  - Processo = area di memoria RAM per effettuare le operazioni e memorizzare i dati + registro che ricorda la prossima istruzione da eseguire + canali di comunicazione
- Ogni processo *comunica attraverso canali*
  - Un canale gestisce *flussi di dati in ingresso e in uscita* (dati in formato binario o testuale)
  - Per esempio lo schermo, la tastiera e la rete sono “*canali*”
  - Dall'esterno ogni canale è identificato da un numero intero detto “*porta*”
- Le socket sono particolari canali per la comunicazione tra processi che non condividono memoria (per esempio perché risiedono su macchine diverse)
- Per potersi connettere o inviare dati ad un processo A, un processo B deve conoscere la macchina (*host*) che esegue A e la porta cui A è connesso (*well-known port*)



## Servizio TCP

- *Orientato alla connessione*: il client invia al server una richiesta di connessione
- *Trasporto affidabile (reliable transfer)* tra processi mittente e ricevente
- *Controllo di flusso (flow control)*: il mittente rallenta per non sommergere il ricevente
- *Controllo della congestione (congestion control)*: il mittente rallenta quando la rete è sovraccarica
- *Non offre* garanzie di banda e ritardo minimi

## Servizio UDP

- Trasporto non affidabile tra processi mittente e ricevente
- Non offre connessione, affidabilità, controllo di flusso, controllo di congestione, garanzie di ritardo e banda

D: perché esiste UDP?

Può essere conveniente per le applicazioni che tollerano perdite parziali (es. video e audio) a vantaggio delle prestazioni

## Servizio UDP:

- Scompone il flusso di byte in segmenti
- Li invia, uno per volta, ai servizi network

## Servizio TCP:

- Scompone e invia come UDP
- Ogni segmento viene numerato per garantire
  - Riordinamento dei segmenti arrivati
  - Controllo delle duplicazioni (scarto i segmenti con ugual numero d'ordine)
  - Controllo delle perdite (rinvio i segmenti mancanti)
- Per progettare e realizzare sistemi distribuiti
  - NON è necessario conoscere il funzionamento (information hiding) dei processi
  - Ciò che importa è lo scambio dati (stream di byte) tra i processi

## □ TCP

- Utilizza variabili e buffer per realizzare il trasferimento bidirezionale di flussi di bytes ("pipe") tra processi
- Prevede ruoli client/server durante la connessione
- NON prevede ruoli client/server per la comunicazione
- Utilizza i servizi dello strato IP per l'invio dei flussi di bytes

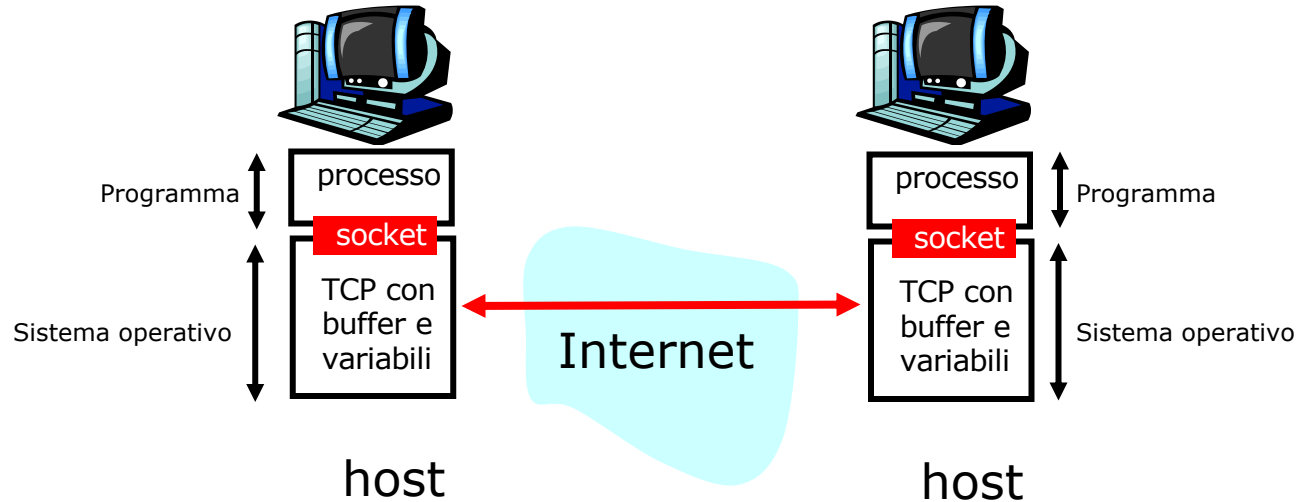
## □ API: **Application Programming Interface**

- Definisce l'interfaccia tra applicazione e strato di trasporto

## □ **Socket**: API per accedere a TCP e UDP

- Due processi (applicazione nel modello client server) comunicano inviando/leggendo dati in/da socket

# Socket: funzionamento di base



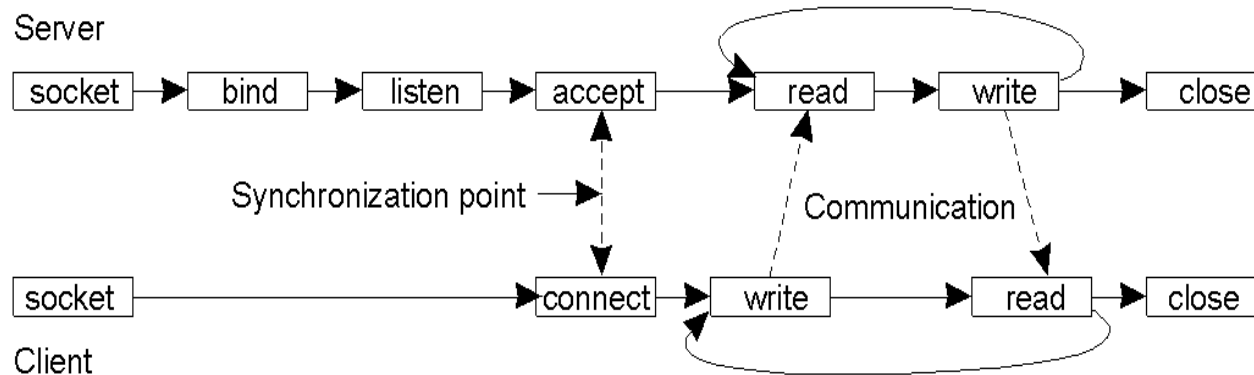
- Gestione del ciclo di vita di client e server
  - Attivazione/terminazione del cliente e del server (es. Manuale o gestita da un middleware)
- Identificazione e accesso al server
  - Informazioni che deve conoscere il cliente per accedere al server
- Comunicazione tra cliente e server
  - Le primitive disponibili e le modalità per la comunicazione (es. TCP/IP: Stream di dati inviati con send/receive)
- Ripartizione dei compiti tra client e server
  - Dipende dal tipo di applicazione (es. controllo: una banca gestisce tutto lato server)
  - Influenza la prestazioni in relazione al carico (numero di clienti)

- Come fa il client a conoscere l'indirizzo del server?
- Alternative:
  - inserire nel codice del client l'indirizzo del server espresso come costante (es. il client di un servizio bancario)
  - chiedere all'utente l'indirizzo (es. web browser)
  - utilizzare un name server o un repository da cui il client può acquisire le informazioni necessarie (es. Domain Name Service – DNS – per tradurre nomi simbolici)
  - adottare un protocollo diverso per l'individuazione del server (es. broadcast per DHCP)

How are the fundamental issues addressed in TCP/IP?

- ❑ Identify the counterpart (**naming**)
  - Low level identification: the name of hosts and protocols
- ❑ Accessing the counterpart (**access point**)
  - Use of the IP address (host:port) to access a process
- ❑ Communicating 1 (**protocol**)
  - Stream of bytes
- ❑ Communicating 2 (**syntax and semantics**)
  - Application protocols with predefined semantics (http, smtp)
- ❑ What level of transparency?
  - Very low: the programmer/user need to
    - know network addresses
    - parse bytes to get the content (message)

- La comunicazione TCP/IP avviene attraverso *flussi di byte* (byte stream), dopo una *connessione esplicita*, tramite normali system call read/write
- Read e write
  - Sono sospensive (bloccano il processo finché il sistema operativo non ha effettuato la lettura/scrittura)
  - Utilizzano un buffer per garantire flessibilità (es: la read definisce un buffer per leggere N caratteri, ma potrebbe ritornare avendone letti solo  $k < N$ )



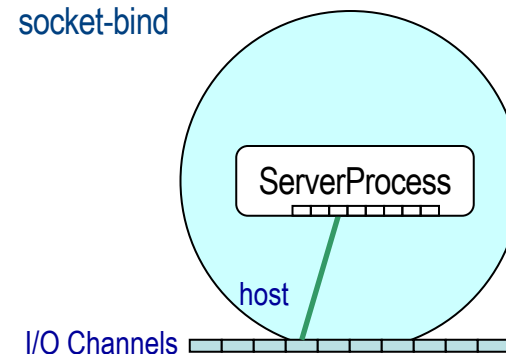
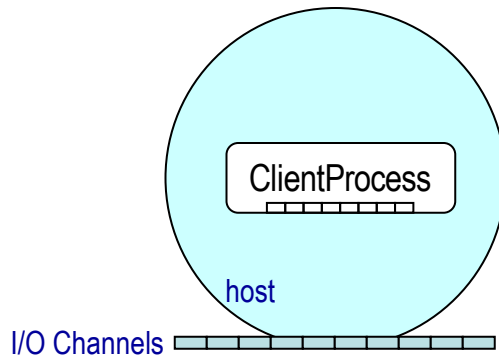


# API socket system calls (Berkeley)

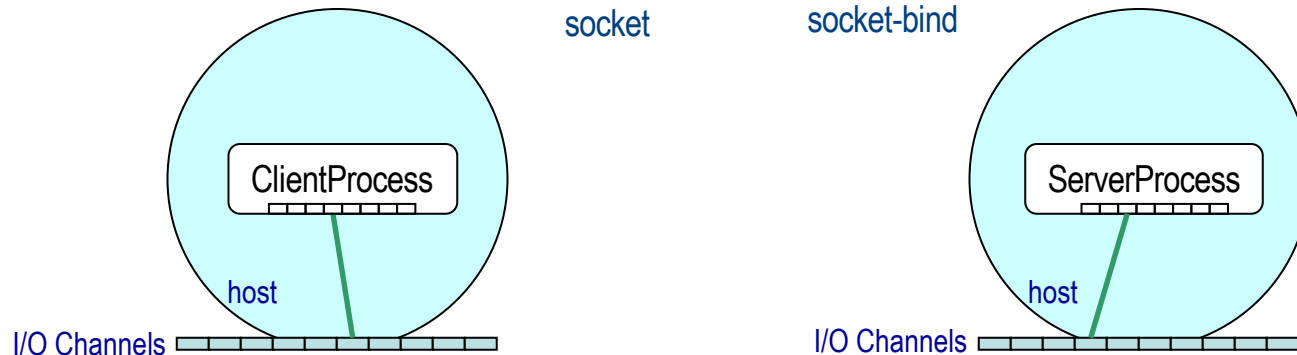
- Many calls are provided to access TCP and UDP services
- The most relevant ones are in the table below

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a <b>local address</b> to a socket, set the queue length
Listen	Announce willingness to accept connections
Accept	<b>Block caller</b> until a connection request arrives
Connect	Actively attempt to establish a connection
Write	Send <b>some data</b> over the connection
Read	Receive <b>some data</b> over the connection
Close	Release the connection

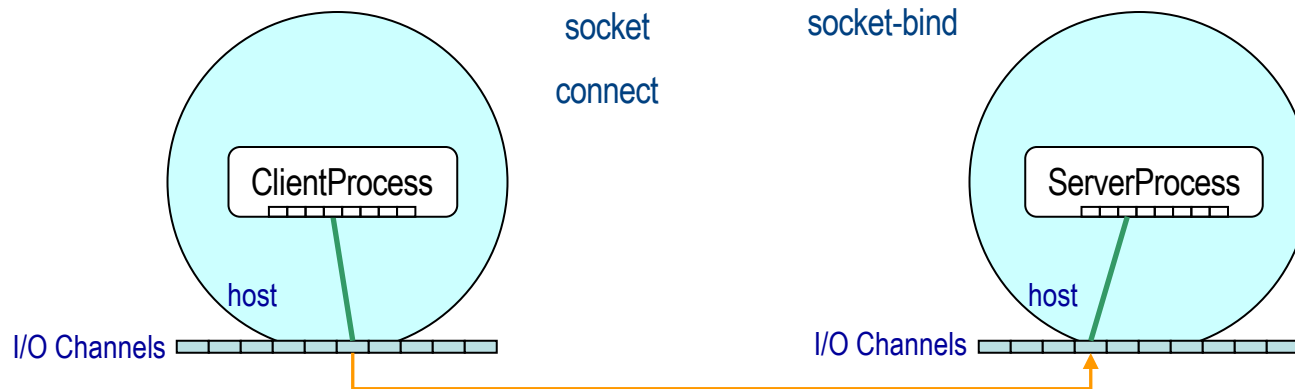
- Il server crea una socket collegata alla well-known port (che identifica il servizio fornito) dedicata a ricevere richieste di connessione
- Con la `accept()`, il server crea una nuova socket, cioè un nuovo canale, dedicato alla comunicazione con il client



- Il server crea una socket collegata alla well-known port (che identifica il servizio fornito) dedicata a ricevere richieste di connessione
- Con la `accept()`, il server crea una nuova socket, cioè un nuovo canale, dedicato alla comunicazione con il client

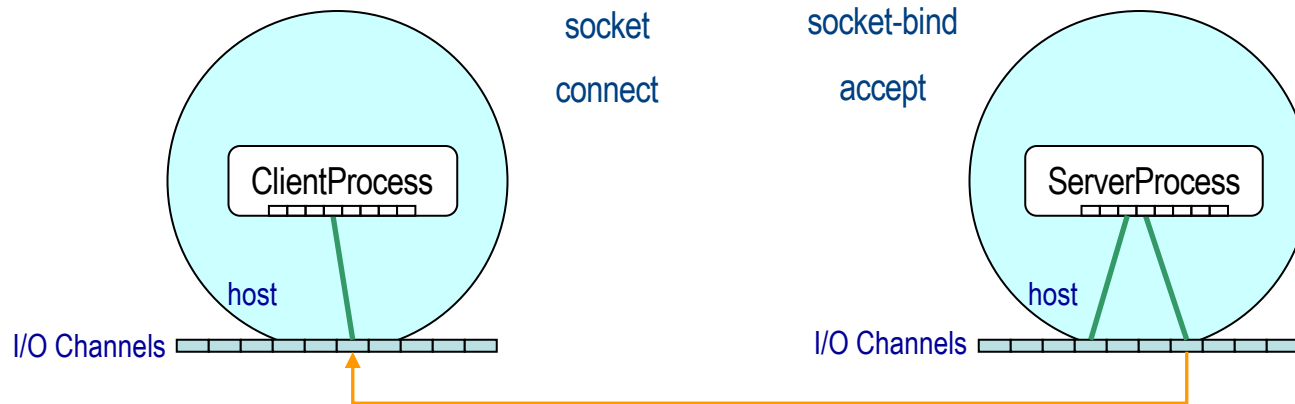


- Il server crea una socket collegata alla well-known port (che identifica il servizio fornito) dedicata a ricevere richieste di connessione
- Con la `accept()`, il server crea una nuova socket, cioè un nuovo canale, dedicato alla comunicazione con il client



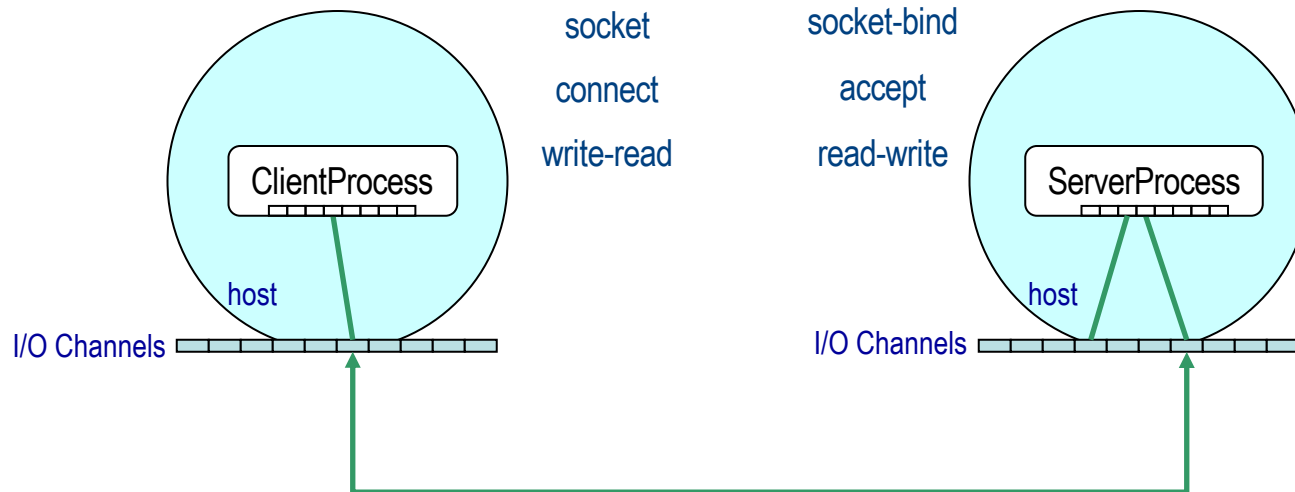
Domanda: Chi stabilisce il formato della richiesta?  
L'applicazione o lo strato di trasporto (TCP)?

- Il server crea una socket collegata alla well-known port (che identifica il servizio fornito) dedicata a ricevere richieste di connessione
- Con la `accept()`, il server crea una nuova socket, cioè un nuovo canale, dedicato alla comunicazione con il client



Domanda: Perché non si usa la stessa socket?

- Il server crea una socket collegata alla well-known port (che identifica il servizio fornito) dedicata a ricevere richieste di connessione
- Con la `accept()`, il server crea una nuova socket, cioè un nuovo canale, dedicato alla comunicazione con il client



- Le socket trasportano “stream” (= flussi) di bytes, quindi ... non c’è il concetto di “messaggio” (il flusso è continuo, senza fine), ... la lettura/scrittura avviene per un numero arbitrario di byte
- Il prototipo (in pseudocodice) della read è quindi

```
byteLetti read(socket, buffer, dimBuffer);
```

*byteLetti* = byte effettivamente letti

*socket* = il canale da cui leggere

*buffer* = lo spazio di memoria dove trasferire i byte letti

*dimBuffer* = dimensione del buffer = numero max di caratteri che si possono leggere

Quindi si devono prevedere ***cicli di lettura*** che termineranno in base alla dimensione dei “messaggi” come stabilito dal formato del protocollo applicativo in uso

# Le socket in Java

●●● INSIDE&S Lab ●●●  
<http://inside.disco.unimib.it/>



- ❑ Java definisce alcune classi che costituiscono un'interfaccia ad oggetti alle *system call* illustrate in precedenza
- ❑ Le principali
  - `java.net.Socket`
  - `java.net.ServerSocket`
- ❑ Queste classi accorpano funzionalità e mascherano alcuni dettagli con il vantaggio di semplificarne l'uso
- ❑ Come per ogni *framework* è necessario conoscerne il modello e il funzionamento per poterlo utilizzare in modo efficace
- ❑ Le prossime slide discutono i principali metodi delle due classi

Reference:

<https://docs.oracle.com/en/java/javase/13/docs/api/jdk.net/jdk/net/package-summary.html>

- Constructors

**public Socket()**

Creates an unconnected socket, with the system-default type of `SocketImpl`.

**public Socket(String host, int port)**

**throws UnknownHostException, IOException**

Creates a stream socket and connects it to the specified port number on the named host. If the specified host is `null`, the loopback address is assumed.

The `UnknownHostException` is thrown if the IP address of the host could not be determined.

**public Socket(InetAddress address, int port)**

**throws IOException**

Creates a stream socket and connects it to the specified port number at the specified IP address.

- Methods to manage connections

**public void bind(SocketAddress bindpoint) throws IOException**

Binds the socket to a local address. If the address is `null`, then the system will pick up an ephemeral port and a valid local address to bind the socket.

**public void connect(SocketAddress endpoint) throws IOException**

Connects this socket to the server.

**public void connect(SocketAddress endpoint, int timeout)**

**throws IOException**

Connects this socket to the server with a specified timeout value (in milliseconds).

**public void close()**

Closes this socket.

□ Methods to establish I/O channels to exchange bytes

**public InputStream getInputStream() throws IOException**

Returns an input stream for this socket.

If this socket has an associated channel then the resulting input stream delegates all of its operations to the channel. If the channel is in non-blocking mode then the input stream's read operations will throw an `IllegalBlockingModeException`.

When a broken connection is detected by the network software the following applies to the returned input stream:

- The network software may discard bytes that are buffered by the socket. Bytes that aren't discarded by the network software can be read using `read`.
- If there are no bytes buffered on the socket, or all buffered bytes have been consumed by `read`, then all subsequent calls to `read` will throw an `IOException`.
- If there are no bytes buffered on the socket, and the socket has not been closed using `close`, then `available` will return 0.

**public OutputStream getOutputStream() throws IOException**

Returns an output stream for writing bytes to this socket.

If this socket has an associated channel, then the resulting output stream delegates all of its operations to the channel.

If the channel is in non-blocking mode, then the output stream's write operations will throw an `IllegalBlockingModeException`.

## □ Constructors

**public ServerSocket() throws IOException**

Creates an unbound server socket.

**public ServerSocket(int port) throws IOException**

Creates a server socket, bound to the specified port.

A port of 0 creates a socket on any free port.

The maximum queue length for incoming connection indications (a request to connect) is set to 50.

If a connection indication arrives when the queue is full, the connection is refused.

**public ServerSocket(int port, int backlog) throws IOException**

Creates a server socket and binds it to the specified local port number, with the specified backlog.

A port of 0 creates a socket on any free port.

The maximum queue length for incoming connection indications (a request to connect) is set to the backlog parameter.

If a connection indication arrives when the queue is full, the connection is refused.

- Methods to manage connections

- public void bind(SocketAddress endpoint) throws IOException**

- Binds the ServerSocket to a specific address (IP address and port number).

- If the address is null, then the system will pick up an ephemeral port and a valid local address to bind the socket.

- public void bind(SocketAddress endpoint, int backlog) throws IOException**

- Binds the ServerSocket to a specific address (IP address and port number).

- If the address is null, then the system will pick up an ephemeral port and a valid local address to bind the socket.

- The backlog argument must be a positive value greater than 0. If the value passed is equal or less than 0, then the default value will be assumed.

- public Socket accept() throws IOException**

- Listens for a connection to be made to this socket and accepts it. Returns the new Socket.

- The method blocks until a connection is made.

- Utility methods

- public InetAddress getInetAddress()**

- Returns the local address of this server socket or null if the socket is unbound.

- public int getLocalPort()**

- Returns the port on which this socket is listening or -1 if the socket is not bound yet.

- public SocketAddress getLocalSocketAddress()**

- Returns the address of the endpoint this socket is bound to, or null if it is not bound yet.

- Let's study an example in which:
  - A server accept a connection with a client, and then sends a stream of bytes (e.g., a string of characters)
  - A client request a connection, and then reads the stream of bytes sent by the server
  
- The goal is to give evidence that:
  - The behavior of the involved processes is independent from each other
  - They exchange streams of byte
  - The notion of "message" or "application protocol" is not part of the socket definition

```
1. import java.io.PrintWriter;
2. import java.net.ServerSocket;
3. import java.net.Socket;

4. public class SenderServerSocket {

5.     final static String message =
6.         "This is a not so short text to test the reading capabilities of clients.";

7.     public static void main(String[] args) {

8.         try {
9.             Socket clientSocket;
10.            ServerSocket listenSocket;

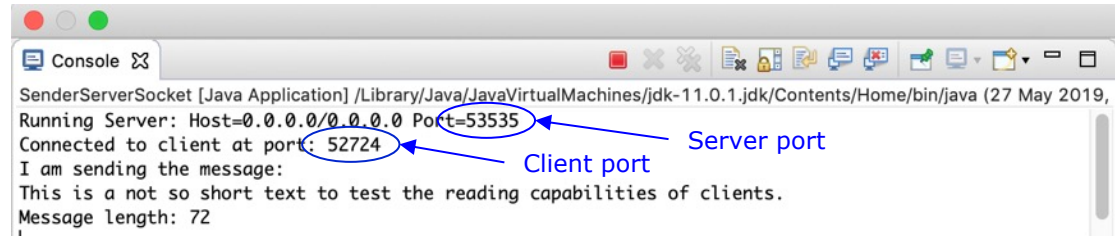
11.            listenSocket = new ServerSocket(53535);
12.            System.out.println("Running Server:" +
13.                " Host= " + listenSocket.getInetAddress() +
14.                " Port= " + listenSocket.getLocalPort());
```

```
15. // loop to open a connection, send the message, close the connection
16. while (true) {
17.     clientSocket = listenSocket.accept();
18.     System.out.println("Connected to client at port: " + clientSocket.getPort());
19.     PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

20.     System.out.println("I am sending the message: \n" + message);
21.     System.out.println("message length: " + message.length());

22.     out.write(message);
23.     out.flush();

24.     clientSocket.close();
25. }
26. } catch (Exception e) {
27.     e.printStackTrace();
28. }
29. }
30. }
```





```
1. import java.io.DataInputStream;
2. import java.io.IOException;
3. import java.net.InetAddress;
4. import java.net.Socket;

5. public class ReceiverClientSocket {

6.     public static void main(String[] args) {
7.         Socket socket; // my socket
8.         InetAddress serverAddress; // the server address
9.         int serverPort; // the server port

10.        try { // connect to the server
11.            serverAddress = InetAddress.getByName(args[0]);
12.            serverPort = Integer.parseInt(args[1]);
13.            socket = new Socket(serverAddress, serverPort);

14.            System.out.println("Connected to: " + socket.getInetAddress());
```

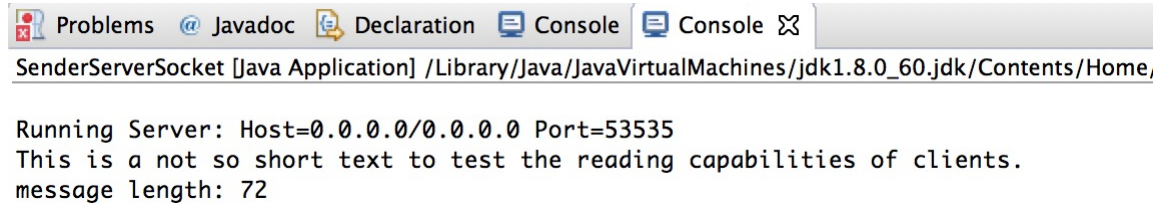
```
15.    DataInputStream in; // the source of stream of bytes
16.    byte[] byteReceived = new byte[1000]; // the temporary buffer
17.    String messageString = ""; // the text to be displayed

18.    // the stream to read from
19.    in = new DataInputStream(socket.getInputStream());
20.    System.out.println("Ready to read from the socket");

21.    // The following code shows in detail how to read from a TCP socket
22.    int bytesRead = 0; // the number of bytes read
23.    bytesRead = in.read(byteReceived);
24.    messageString += new String(byteReceived, 0, bytesRead);

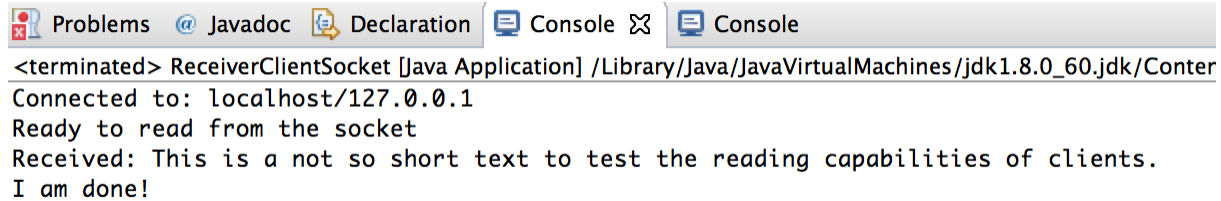
25.    System.out.println("Received: " + messageString);
26.    System.out.println("I am done!");

27.    socket.close();
28.    } catch (IOException e) {
29.        e.printStackTrace();
30.    }
31. }
32. }
```



The screenshot shows an IDE interface with tabs for Problems, Javadoc, Declaration, and two Console windows. The active console window displays the following text:

```
SenderServerSocket [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home,  
  
Running Server: Host=0.0.0.0/0.0.0.0 Port=53535  
This is a not so short text to test the reading capabilities of clients.  
message length: 72
```



The screenshot shows the same IDE interface with the second Console window active. It displays the following text:

```
<terminated> ReceiverClientSocket [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Conter  
Connected to: localhost/127.0.0.1  
Ready to read from the socket  
Received: This is a not so short text to test the reading capabilities of clients.  
I am done!
```

- ❑ It works!
- ❑ But the client coding is not correct! Why?
- ❑ To discover the answer, let's introduce the Lazy Server:
  - A server that sends a few bytes at a time with a delay between sendings

```
15. // loop to open a connection, send the message, close the connection
16. while (true) {
17.     clientSocket = listenSocket.accept();

18.     PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
19.     System.out.println("message length: " + message.length());

20.     final int chunk = 9; // number of bytes sent each time
21.     boolean end = false;

22.     int i;
23.     for (i = 0; i < message.length() - chunk; i += chunk) {
24.         System.out.println(message.substring(i, i + chunk));
25.         Thread.sleep(1000); // lazy sender: wait 1" before sending
26.         out.write(message.substring(i, i + chunk));
27.         out.flush();
28.     }
29.     System.out.println(message.substring(i, message.length()));
30.     out.write(message.substring(i, message.length()));
31.     out.flush();

32.     clientSocket.close();
33. }
34. ...
```

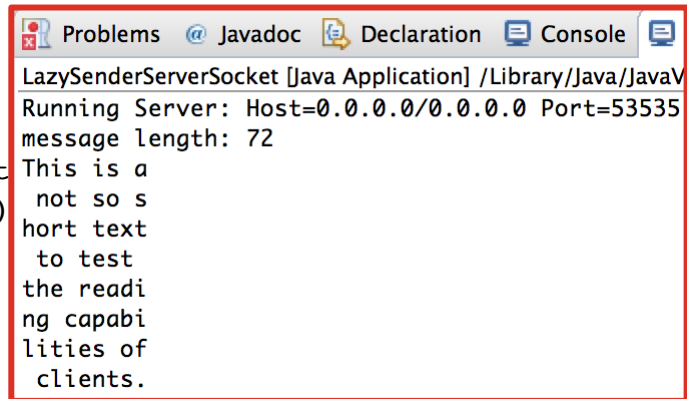
```
15. // loop to open a connection, send the message, close the
16. while (true) {
17.     clientSocket = listenSocket.accept();

18.     PrintWriter out = new PrintWriter(clientSocket.getOutputStream());
19.     System.out.println("message length: " + message.length());

20.     final int chunk = 9; // number of bytes sent each time
21.     boolean end = false;

22.     int i;
23.     for (i = 0; i < message.length() - chunk; i += chunk) {
24.         System.out.println(message.substring(i, i + chunk));
25.         Thread.sleep(1000); // lazy sender: wait 1" before sending
26.         out.write(message.substring(i, i + chunk));
27.         out.flush();
28.     }
29.     System.out.println(message.substring(i, message.length()));
30.     out.write(message.substring(i, message.length()));
31.     out.flush();

32.     clientSocket.close();
33. }
34. ...
```



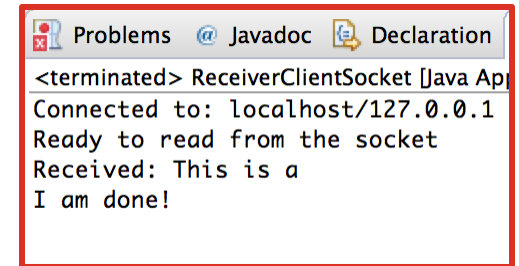
```
15.    DataInputStream in; // the source of stream of bytes
16.    byte[] byteReceived = new byte[1000]; // the temporary buffer
17.    String messageString = ""; // the text to be displayed

18.    // the stream to read from
19.    in = new DataInputStream(socket.getInputStream());
20.    System.out.println("Ready to read from the socket");

21.    // The following code shows in detail how to read from a TCP socket
22.    int bytesRead = 0; // the number of bytes read
23.    bytesRead = in.read(byteReceived);
24.    messageString += new String(byteReceived, 0, bytesRead);

25.    System.out.println("Received: " + messageString);
26.    System.out.println("I am done!");

27.    socket.close();
28. } catch (IOException e) {
29.     e.printStackTrace();
30. }
31. }
32. }
```



Q: What's wrong with the code?

```

15.    DataInputStream in; // the source of stream of bytes
16.    byte[] byteReceived = new byte[1000]; // the temporary buffer
17.    String messageString = ""; // the text to be displayed

18.    // the stream to read from
19.    in = new DataInputStream(socket.getInputStream());
20.    System.out.println("Connected to: localhost/127.0.0.1");
21.    // The socket is ready to read from the socket
22.    int bytesRead = 0;
23.    while (bytesRead < messageString.length()) {
24.        byte[] receivedBytes = new byte[1000];
25.        if (in.read(receivedBytes) > 0) {
26.            String receivedText = new String(receivedBytes);
27.            messag
28.            System.out.println("Received: " + receivedText);
29.        }
30.        System.out.println("I am done!");

31.        socket.close();
32.    } catch (IOException e) {
33.        e.printStackTrace();
34.    }
35. }
36. }

```

Problems @ Javadoc Declaration Console Console

<terminated> SmartReceiverClientSocket [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_60.jdk/

Connected to: localhost/127.0.0.1

Ready to read from the socket

Received: This is a

Received: This is a not so s

Received: This is a not so short text

Received: This is a not so short text to test

Received: This is a not so short text to test the readi

Received: This is a not so short text to test the reading capabi

Received: This is a not so short text to test the reading capabilities of

Received: This is a not so short text to test the reading capabilities of clients.

I am done!

```
15.    DataInputStream in; // the source of stream of bytes
16.    byte[] byteReceived = new byte[1000]; // the temporary buffer
17.    String messageString = ""; // the text to be displayed
18.
19.    // the stream to read from
20.    in = new DataInputStream(socket.getInputStream());
21.    System.out.println("Ready to read from the socket");
22.
23.    // The following code shows in detail how to read from a TCP socket
24.    int bytesRead = 0; // the number of bytes read
25.    while (true) {
26.        bytesRead = in.read(byteReceived);
27.        if (bytesRead == -1)
28.            break; // no more bytes
29.        messageString += new String(byteReceived, 0, bytesRead);
30.        System.out.println("Received: " + messageString);
31.    }
32.    System.out.println("I am done!");
33.
34.    socket.close();
35. } catch (IOException e) {
36.     e.printStackTrace();
37. }
```

← Data from sockets  
must always be read  
with a reading loop



**Client:** L'architettura è concettualmente più semplice di quella di un server

- È spesso un'applicazione convenzionale che usa una socket anziché da un altro canale I/O
- Ha effetti solo sull'utente client: non ci sono particolari problemi di sicurezza

**Server:** L'architettura generale prevede che

- venga creata una socket con una porta nota per accettare le richieste di connessione
- entri in un ciclo infinito in cui alternare
  1. attesa/accettazione di una richiesta di connessione da un client
  2. ciclo lettura-esecuzione, invio risposta fino al termine della conversazione (stabilito spesso dal client)
  3. chiusura connessione

Problematiche connesse:

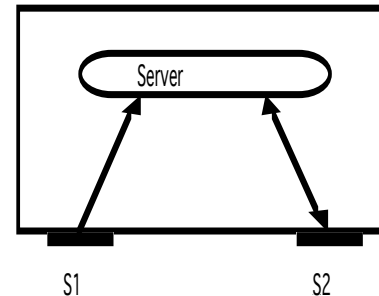
- L'affidabilità del server è strettamente dipendente dall'affidabilità della comunicazione tra lui e i suoi client
- La modalità connection-oriented determina
  - l'impossibilità di rilevare interruzioni sulle connessioni (il client controlla il server)
  - la necessità di una connessione (una socket) per ogni conversazione
  - problemi di sicurezza per la condivisione dei dati e il controllo affidato al client

# Architetture dei server

●●● INSIDE&S Lab ●●●  
<http://inside.disco.unimib.it/>

- I server possono essere:
  - iterativi  
soddisfano una richiesta alla volta
  - concorrenti processo singolo  
simulano la presenza di un server dedicato
  - concorrenti multi-processo  
creano server dedicati
  - concorrenti multi-thread  
creano thread dedicati

- Al momento di una richiesta di connessione il server crea una socket temporanea per stabilire una connessione diretta con il client
- Le eventuali ulteriori richieste per il server verranno accodate alla porta nota per essere successivamente soddisfatte
- Vantaggi
  - Semplice da progettare
- Svantaggi
  - Viene servito un cliente alla volta, gli altri devono attendere
  - Un client può impedire l'evoluzione di altri client
  - Non scala
- Soluzione: server concorrenti



Legenda:

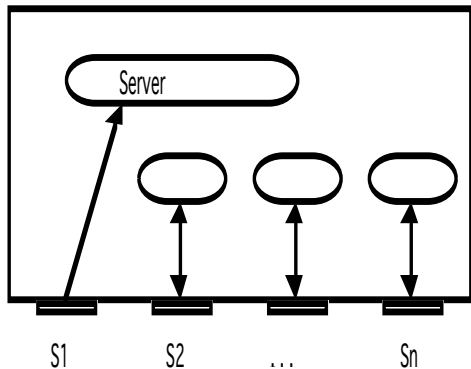
- S1 Socket per accettare richieste di connessione
- S2 Socket per connessioni individuali

Let's consider a server that reads from a socket and writes on the console

```
1. public static void main(String[] args) {
2.     ServerSocket listenSocket;
3.     Socket clientSocket;
4.     byte[] byteReceived = new byte[1000];
5.     String messageString = "";

6.     try {
7.         listenSocket = new ServerSocket(5353); // new socket to listen for requests at port 53535
8.         while (true) { // listen, serve, close, next-client loop
9.             clientSocket = listenSocket.accept(); // new socket to talk to the client
10.            DataInputStream in = new DataInputStream(clientSocket.getInputStream()); // the reading channel
11.            int bytesRead = 0;
12.            while (true) { // loop to read all the bytes sent by the current client
13.                bytesRead = in.read(byteReceived);
14.                if (bytesRead == -1)
15.                    break; // no more bytes
16.                messageString += new String(byteReceived, 0, bytesRead); // all bytes received so far
17.                System.out.println("Received: " + messageString);
18.            }
19.            clientSocket.close();
20.            messageString = "";
21.            System.out.println("*** Available to the next client.");
22.        }
23.    } catch (Exception e) {
24.        e.printStackTrace();
25.    }
26. }
```

- Un server concorrente può gestire più connessioni client
- La sua realizzazione può essere
  - simulata con un solo processo
    - (A) in C: funzione *select*
    - in Java: uso *Selector*
    - che conoscere i canali *ready to use*
  - (B) in Java: uso dei *Thread*
  - reale creando nuovi processi slave
    - (C) in C: uso della funzione *fork*

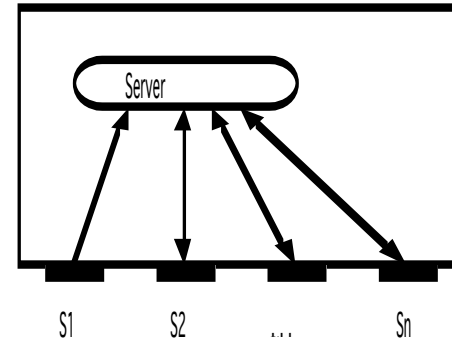


Legenda:

S1 Socket per accettare richieste di connessione  
S2...Sn Socket per connessioni individuali

○ Processi slave

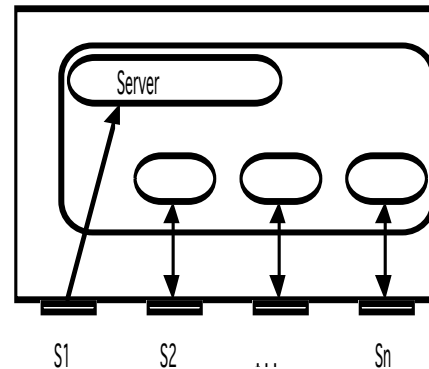
(C)



Legenda:

S1 Socket per accettare richieste di connessione  
S2...Sn Socket per connessioni individuali

(A)



Legenda:

S1 Socket per accettare richieste di connessione  
S2...Sn Socket per connessioni individuali

○ Threads

(B)

- Un server concorrente può gestire più connessioni client
- La sua realizzazione può essere

- simulata con un solo processo

(A) in C: 1

in Java: u

che conos

(B) in Java: uso del *Thread*

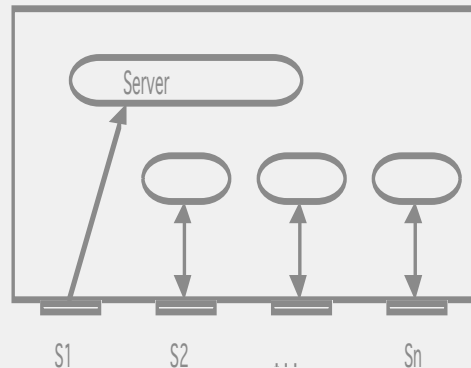
- reale creando nuovi processi slave

(C) in C: uso della funzione *fork*

(A) viene discusso nel seguito

(B) verrà discusso nella parte di concorrenza

(C) dovrebbe essere noto da sistemi operativi (faremo un breve ripasso)

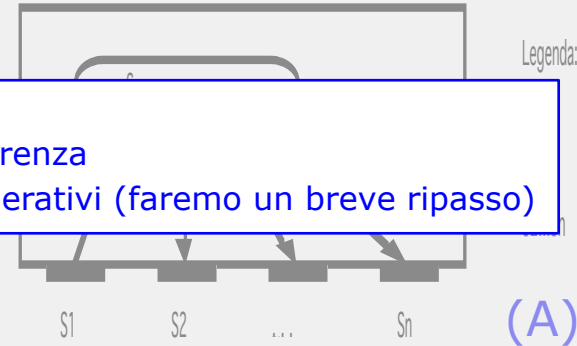


Legenda:

S1 Socket per accettare richieste di connessione  
S2...Sn Socket per connessioni individuali

Processi slave

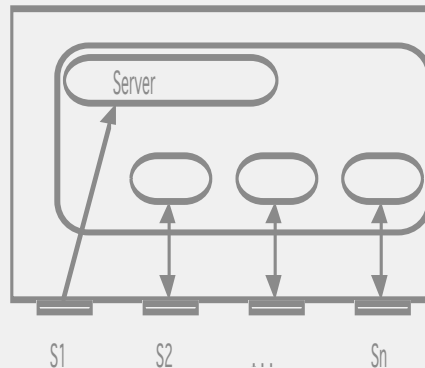
(C)



Legenda:

Socket per accettare richieste di connessione  
Socket per connessioni individuali

(A)



Legenda:

S1 Socket per accettare richieste di connessione  
S2...Sn Socket per connessioni individuali

Threads

(B)

```
1. ... // declarations
2. Socket[] clientSocket = new Socket[2];
3. DataInputStream[] in = new DataInputStream[2];

4. try {
5.     listenSocket = new ServerSocket(53535);
6.     // accept two connections
7.     clientSocket[0] = listenSocket.accept();
8.     clientSocket[1] = listenSocket.accept();

9.     // create two reading channels
10.    in[0] = new DataInputStream(clientSocket[0].getInputStream());
11.    in[1] = new DataInputStream(clientSocket[1].getInputStream());

12.    // what happens when start reading from one of the two channels?
13.    int bytesRead = 0;
14.    while (true) {
15.        bytesRead = in[0].read(byteReceived);
16.        if (bytesRead == -1)
17.            break; // no more bytes
18.        messageString += new String(byteReceived, 0, bytesRead);
19.        System.out.println("Received: " + messageString);
20.    }
21.    ... // the rest of the code (e.g., a similar loop to read from in[1] )
```

Let's consider a server that accepts two clients to read from sockets and write on the console

This server is not concurrent.  
Why?  
How can we change the code?

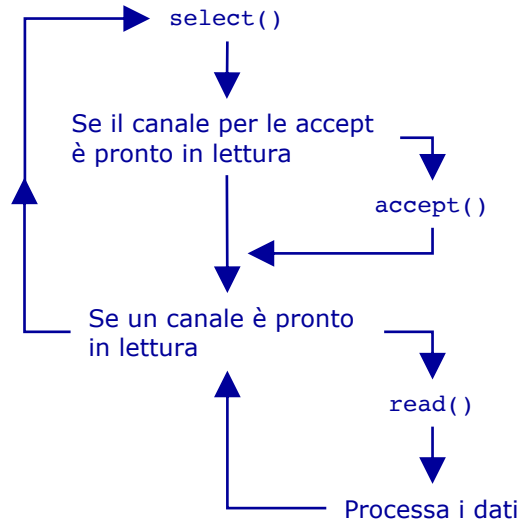


- Le operazioni di lettura e scrittura comportano l'uso di system call bloccanti

Bloccante = si attende la conclusione dell'operazione richiesta prima di restituire il controllo al chiamante

- Per leggere in modo non bloccante serve sapere prima di fare una operazione di lettura o scrittura se il canale è *pronto* (cioè se faccio una operazione di lettura/scrittura il controllo mi viene restituito immediatamente)
- La system call `select()` ha questo compito
- Il codice del server diventa:
  1. Dico al sistema quali canali voglio usare in modalità non bloccante
  2. Chiamo la `select()` che controlla quali canali sono «pronti»
  3. Sui canali pronti effetto l'operazione `read()` o `write()` desiderata
  4. Ciclo tornando al punto 1

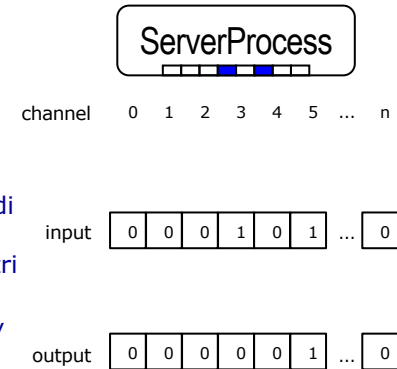
- La `select()` permette gestire in modo non bloccante i diversi canali di I/O
  - Sospende il processo finché non è possibile fare una operazione di I/O
- Un server concorrente realizza un ciclo:



`select()`

Riceve in ingresso un array di bit con 1 per i canali che si vogliono usare e 0 per gli altri

Restituisce in uscita un array di bit con 1 per i canali che sono pronti e 0 per gli altri



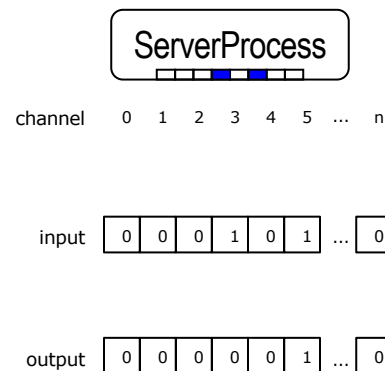
```
#include <sys/types.h>
#include <sys/time.h>
```

```
int select(int maxfdp, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
```

- Usa una maschera di bit `fd_set` di lunghezza `maxfdp (= max_canali + 1)`
  - `max_canali` è definito in `<stdio.h>`
    - In System V è la costante `FOPEN_MAX`
    - In 4.3BSD è dato da `getdtablesize()`
- Una invocazione di `select` segnala che
  - uno dei file descriptor di `readfds` è pronto per la lettura, o che
  - uno dei file descriptor di `writefd` è pronto per la scrittura, o che
  - uno dei file descriptor di `exceptfds` è in una eccezione pendente.
- Usa la maschera di bit di descrittori di file `fd_set` definita in `<sys/types.h>`
- Le macro per manipolare la maschera:

```
FD_ZERO(fd_set *fdset); /* clear all bits in fdset */
FD_SET(int fd, fd_set *fdset); /* turn the bit for fd on in fdset */
FD_CLR(int fd, fd_set *fdset); /* turn the bit for fd off in fdset */
FD_ISSET(int fd, fd_set *fdset); /* test the bit for fd in fdset */
```

- È possibile specificare un `timeout`
  - Se ha valore 0, allora ritorna subito dopo aver controllato i descrittori;
  - Se ha valore  $> 0$ , allora attende che uno dei descrittori sia pronto per l'I/O, ma non oltre il tempo fissato da `timeout`;
  - Se `timeout` ha valore `NULL`, allora attende indefinitamente che uno dei descrittori sia pronto per l'I/O. (si può interrompere l'attesa con una `signal`)



**NB: non fa parte del programma di esame. È uno spunto per chi volesse approfondire.**

- Dalla versione 1.4 è stato introdotto i *channel* che possono operare in modo bloccante o non bloccante
  - Un canale non-bloccante non mette il chiamante in sleep
  - L'operazione richiesta o viene completata immediatamente o restituisce un risultato che nulla è stato fatto
- Solo canali di tipo socket possono essere usati nei due modi
- I canali socket permettono di interagire con i canali di rete
  - Sono implementati dalle classi *ServerSocketChannel*, *SocketChannel*, e *DatagramChannel*
- I canali socket in modo non bloccante possono essere usati con i *selector*
  - Possono essere gestite in modo più efficiente delle socket definite in `java.net`
  - Permettono di gestire più canali in multiplex

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/nio/channels/package-summary.html>

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/nio/channels/Selector.html>

<https://developer.ibm.com/tutorials/j-nio/>

- Un `Selector` permette di gestire dei `SelectableChannel` con una `select()` in Java
- Un selettore può essere creato invocando il metodo statico `open()` della classe `Selector`

```
Selector selector = Selector.open();
```

- I canali da monitorare con la `select` devono essere
  1. messi in modalità non bloccante, e poi
  2. registrati con il metodo `register()`

```
channel.configureBlocking(false);
```

```
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
```

- Ogni canale può essere registrato per monitorare quattro tipo di eventi (modi di utilizzo) definiti da costanti nella classe `SelectionKey`

**Connect** – when a client attempts to connect to the server: `SelectionKey.OP_CONNECT`

**Accept** – when the server accepts a connection from a client: `SelectionKey.OP_ACCEPT`

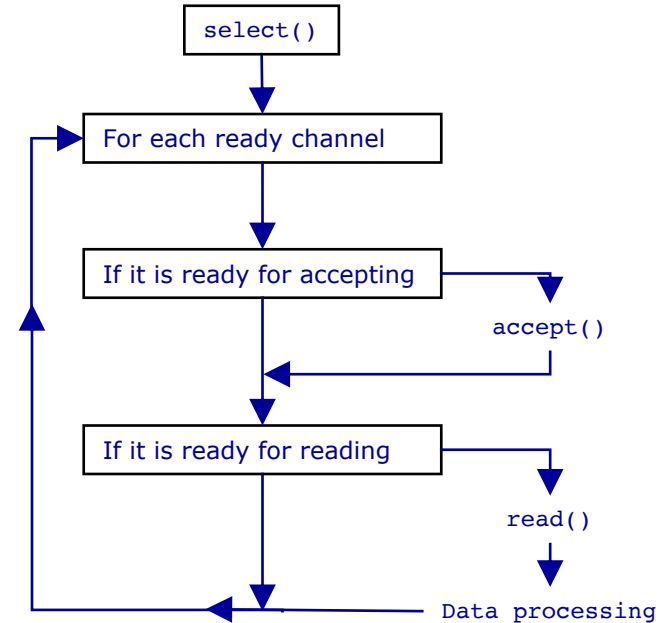
**Read** – when the server is ready to read from the channel: `SelectionKey.OP_READ`

**Write** – when the server is ready to write to the channel: `SelectionKey.OP_WRITE`

- Gli oggetti della classe `SelectionKey` identificano i canali su cui fare le operazioni desiderate

```
1. create ServerSocketChannel;  
2. create Selector;  
3. set the ServerSocketChannel in non-blocking mode  
4. associate the ServerSocketChannel with the Selector;  
5. while(true) {  
6.     waiting events from the Selector;  
7.     event arrived;  
8.     create keys;  
9.     for each key created by Selector {  
10.        check the type of request;  
11.        isAcceptable:  
12.            get the client SocketChannel;  
13.            associate that SocketChannel with the Selector;  
14.            record it for read/write operations  
15.            continue;  
16.        isReadable:  
17.            get the client SocketChannel;  
18.            read from the socket;  
19.            process the read data  
20.            continue;  
21.        isWriteable:  
22.            get the client SocketChannel;  
23.            write on the socket;  
24.            continue;  
25.    }  
26. }
```

A general schema to accept and read from more sockets:



```
1. import java.io.IOException;
2. import java.io.PrintWriter;
3. import java.net.InetAddress;
4. import java.net.Socket;
5. import java.util.Scanner; // this class is meant to both read and parse text data into primitive types

6. public class SenderClient {
7.     private Socket socket;
8.     private Scanner scanner;

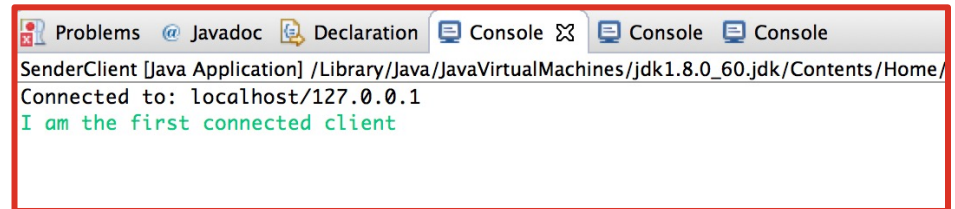
9.     private SenderClient(InetAddress serverAddress, int serverPort) throws Exception {
10.         ...
11.     }

12.     private void start() throws IOException, InterruptedException {
13.         ...
14.     }

15.     public static void main(String[] args) throws Exception {
16.         SenderClient client = new SenderClient(InetAddress.getByName(args[0]), Integer.parseInt(args[1]));
17.         System.out.println("Connected to: " + client.socket.getInetAddress());
18.         client.start();
19.     }
20. }
```

```
9.  private SenderClient(InetAddress serverAddress, int serverPort) throws Exception {
10.      this.socket = new Socket(serverAddress, serverPort);
11.      this.scanner = new Scanner(System.in);
12.  }

13.  private void start() throws IOException, InterruptedException {
14.      String input;
15.      PrintWriter out = new PrintWriter(this.socket.getOutputStream(), true);
16.      while (true) {
17.          input = scanner.nextLine();
18.          if (input.contentEquals("exit"))
19.              break;
20.          out.print(input);
21.          out.flush();
22.      }
23.      System.out.println("Client terminate.");
24.      socket.close();
25.  }
```





```
1. public class ConcurrentServer {
2.     public static void main(String[] args) throws IOException {
3.         Selector selector = Selector.open();
4.         // selectable and good for a stream-oriented listening socket
5.         ServerSocketChannel serverSocket = ServerSocketChannel.open();
6.         serverSocket.bind(new InetSocketAddress("localhost", 5353));
7.         serverSocket.configureBlocking(false); // set the channel in non blocking mode
8.         serverSocket.register(selector, SelectionKey.OP_ACCEPT); // register the channel with the selector
9.
10.        while (true) {
11.            selector.select();
12.            // each key refers to registered channels ready for an operation
13.            Set<SelectionKey> selectedKeys = selector.selectedKeys();
14.            // create an iterator for the ready channels
15.            Iterator<SelectionKey> iter = selectedKeys.iterator();
16.            while (iter.hasNext()) {
17.                SelectionKey key = iter.next();
18.                if (key.isAcceptable()) { // connect a new client
19.                    acceptClientRequest(selector, serverSocket); // accept and set a connection with a client
20.                }
21.                if (key.isReadable()) { // serve a connected client
22.                    readClientBytes(key); // read and process the incoming data
23.                }
24.                iter.remove();
25.            }
26.        }
27.    }
28. }
```

```
26. private static void acceptClientRequest(Selector selector, ServerSocketChannel serverSocket)
27.                                     throws IOException {
28.     SocketChannel client = serverSocket.accept();
29.     client.configureBlocking(false); // set the channel in non blocking mode
30.     client.register(selector, SelectionKey.OP_READ); // register the channel for the read operation
31. }

32. private static void readClientBytes(SelectionKey key) throws IOException {
33.     SocketChannel client = (SocketChannel) key.channel();
34.     int BUFFER_SIZE = 256;
35.     ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE); // buffer to read and write
36.     try { // read bytes coming from the client
37.         if (client.read(buffer) == -1) {
38.             client.close();
39.             return;
40.         }
41.     } catch (Exception e) { // client is no longer active
42.         e.printStackTrace();
43.         return;
44.     }
45.     // Show bytes on the console using the port number to discriminate senders
46.     buffer.flip(); // set the limit to the current position and then set the position to zero
47.     Charset charset = Charset.forName("UTF-8");
48.     CharsetDecoder decoder = charset.newDecoder();
49.     CharBuffer charBuffer = decoder.decode(buffer);
50.     System.out.println(client.socket().getPort() + ": " + charBuffer.toString());
51. }
```

```
Problems @ Javadoc Declaration Console Console Console
SenderClient [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/
Connected to: localhost/127.0.0.1
I am the first connected client
```

```
Problems @ Javadoc Declaration Console Console Console
SenderClient [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/
Connected to: localhost/127.0.0.1
I am the second connected client
```

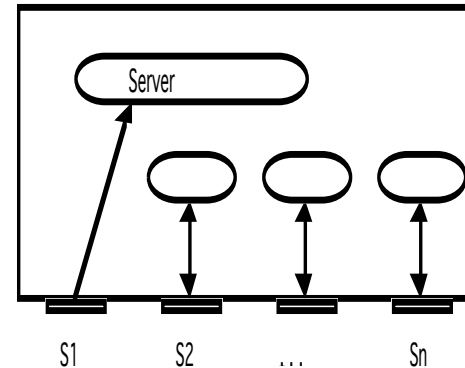
```
Problems @ Javadoc Declaration Console Console Console
ConcurrentServer [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/
63845: I am the first connected client
63846: I am the second connected client
```

```
Problems @ Javadoc Declaration Console Console Console
<terminated> SenderClient [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/
Connected to: localhost/127.0.0.1
I am the first connected client
exit
Client terminate.
```

```
Problems @ Javadoc Declaration Console Console Console
SenderClient [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/
Connected to: localhost/127.0.0.1
I am the latest client
```

```
Problems @ Javadoc Declaration Console Console Console
ConcurrentServer [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/
63845: I am the first connected client
63846: I am the second connected client
63872: I am the latest client
```

- Un server concorrente che crea nuovi processi slave in C: uso della funzione *fork()*
- La *fork()* crea un processo *clone* del padre che
  - eredita i canali di comunicazione
  - esegue lo stesso codice
- Il codice deve prevedere quindi che
  - Il padre chiuda la socket per la conversazione con il client
  - Il figlio chiuda la socket per l'accettazione di nuove connessioni
- La struttura del server è la stessa della versione iterativa in quanto ogni server gestisce un solo client



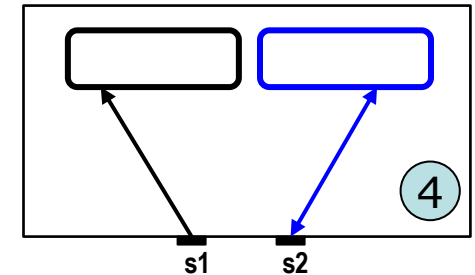
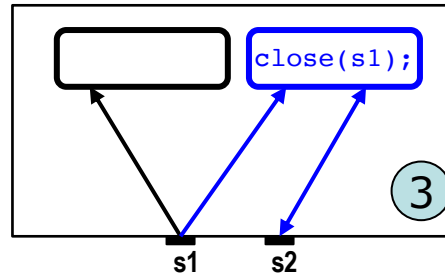
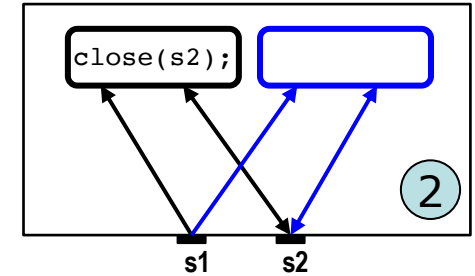
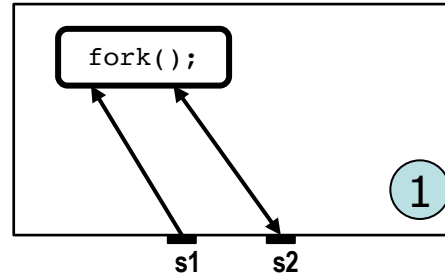
Legenda:

- S1 Socket per accettare richieste di connessione
- S2...Sn Socket per connessioni individuali
- Processi slave

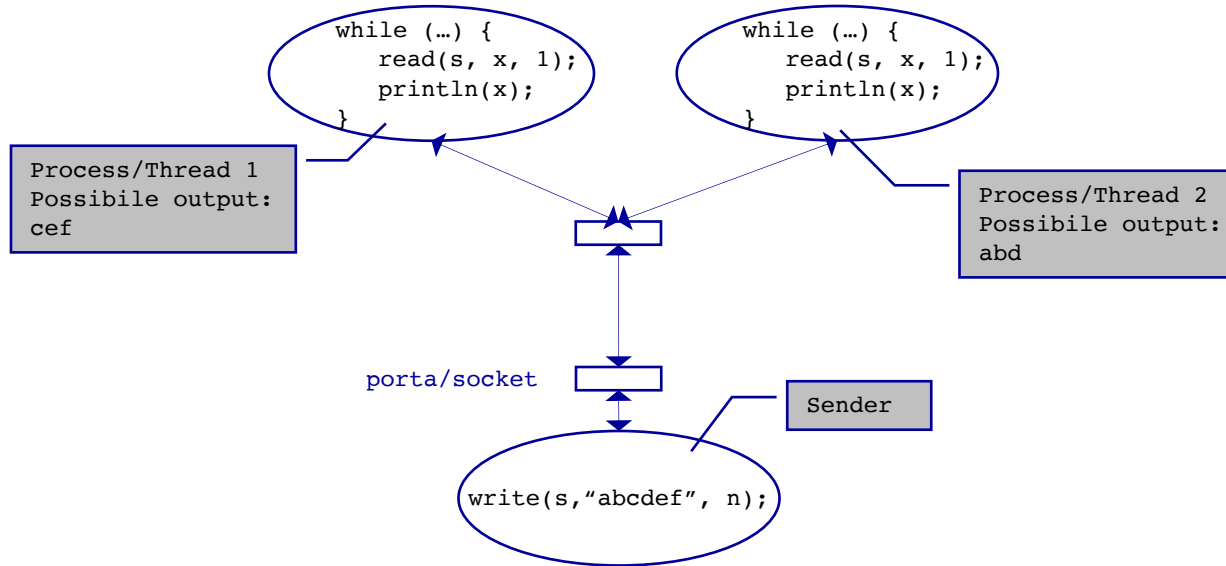
```

1. int s1, s2;
2. ...
3. pid_t pid;
4. pid = fork();
5. if (pid == -1) {
6.     /* fork error - cannot create child */
7.     perror("Cannot create child");
8.     exit(1);
9. } else if (pid == 0) {
10.    /* code for child */
11.    close(s1);
12.    /* probably a few statements then an exec() */
13. } else {
14.    /* code for parent */
15.    printf("Pid of latest child is %d\n", pid);
16.    close(s2);
17.    /* more code */
18.    exit(0);
19. }

```



**NB: non fa parte del programma di esame. È uno spunto per chi volesse approfondire.**



- La lettura/scrittura su una socket da parte di più processi determina un problema di concorrenza: accesso ad una risorsa condivisa (mutua esclusione)

- Java doesn't support a unix-like fork, but processes can be cloned by

```
public final class ProcessBuilder  
    extends Object
```

- This class is used to create operating system processes
  - Each ProcessBuilder instance manages a collection of process attributes
  - The start() method creates a new Process instance with those attributes
  - The start() method can be invoked repeatedly from the same instance to create new subprocesses with identical or related attributes
- An example

```
1. ProcessBuilder pb = new ProcessBuilder("C:\\Windows\\system32\\program.exe");  
2. pb.inheritIO(); // <-- passes IO from forked process  
3. try {  
4.     Process p = pb.start(); // <-- forkAndExec on Unix  
5.     p.waitFor(); // <-- waits for the forked process to complete  
6. } catch (Exception e) {  
7.     e.printStackTrace();  
8. }
```

Reference: <https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/ProcessBuilder.html>

Example: <https://stackoverflow.com/questions/30355085/how-do-i-fork-an-external-process-in-java>

- Quando conviene progettare un server mono processo (iterativo o concorrente) oppure multi processo?
- In altri termini: quali caratteristiche hanno?
- Mono processo (iterativo e concorrente)
  - gli utenti condividono lo stesso spazio di lavoro
  - adatto ad applicazioni cooperative che prevedono la modifica dello stato (lettura e scrittura)
- Multi processo
  - ogni utente ha uno spazio di lavoro autonomo
  - adatto ad applicazioni cooperative che non modificano lo stato del server (sola lettura)
  - adatto ad applicazioni autonome che modificano uno spazio di lavoro proprio (lettura e scrittura)



- Il protocollo
  - di basso livello (flusso di byte/caratteri)
  - l'applicazione si deve far carico della codifica/decodifica dei dati
  - NB: non ci sono "messaggi" predefiniti: sono definiti a livello applicazione dal progettista
- I servizi
  - elementari: bassa trasparenza (solo meccanismi base)
- Connessione
  - non c'è un servizio di naming
  - indirizzo fisico (host:port) per accedere
- Non c'è supporto alla gestione del ciclo di vita
  - creazione e attivazione esplicita dei componenti (client e server) (es. superserver in Unix)

# Un esempio client/server con socket TCP/IP in Java

●●● INSIDE&S Lab ●●●  
<http://inside.disco.unimib.it/>

- Vogliamo realizzare un semplice programma per giocare a “knock knock”, il popolare gioco di parole Inglese che si basa su domande e risposte con parole ed espressioni omofone di significato differente
- Le conversazioni seguono un protocollo pre-definito con domande e risposte costanti (messaggi predefiniti)
- Esempio di conversazione:  
Server: "Knock knock!"  
Client: "Who's there?"  
Server: "Atch."  
Client: "Atch who?"  
Server: "Bless you!"

Fonte: The Java Tutorial (JDK 8): <https://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html>  
Trail: Custom Networking  
Lesson: All About Sockets

- La soluzione consiste in tre classi
  - Una classe *KnockKnockProtocol*, che stabilisce il protocollo (definisce domande e risposte) e fornisce la funzione che permette al server di formulare le risposte
  - Una classe *KnockKnockServer*, che accetta la richiesta di connessione del client e lo interroga secondo il protocollo stabilito
  - Una classe *KnockKnockClient*, che stabilisce la connessione e riceve/invia i messaggi dal/al server
- In questo caso il protocollo è conosciuto solo dal server che usa le risposte per guidare il client a porre le domande giuste

Nota: qui il ruolo attivo viene assunto dal server che inizia la conversazione (in pratica un'inversione dei ruoli)

```
1.  public class KnockKnockProtocol {
2.      private static final int WAITING = 0; // la conversazione può iniziare
3.      private static final int SENTKNOCKKNOCK = 1; // inviato il messaggio iniziale
4.      private static final int SENTCLUE = 2; // ricevuto la domanda corretta e inviata la risposta
5.      private static final int ANOTHER = 3; // proseguire?

6.      private static final int NUMJOKES = 5; // Massimo numero di conversazioni con un client

7.      private int state = WAITING; // stato del gioco
8.      private int currentJoke = 0;

9.      private String[] clues = { "Turnip", "Little Old Lady", "Atch", "Who", "Who" };
10.     private String[] answers = { "Turnip the heat, it's cold in here!",
11.                                   "I didn't know you could yodel!",
12.                                   "Bless you!",
13.                                   "Is there an owl in here?",
14.                                   "Is there an echo in here?" };

15.     public String processInput(String theInput) {
16.         ...
17.     }
18. }
```

## Esempio di conversazione:

Server: "Knock knock!"  
Client: "Who's there?"  
Server: "Atch."  
Client: "Atch who?"  
Server: "Bless you!"

# La conversazione

```

1. public String processInput(String theInput) {
2.     String theOutput = null;

3.     if (state == WAITING) { // la conversazione può iniziare
4.         theOutput = "Knock! Knock!";    state = SENTKNOCKKNOCK; // inviato il messaggio iniziale
5.     } else if (state == SENTKNOCKKNOCK) { // risposta al messaggio iniziale
6.         if (theInput.equalsIgnoreCase("Who's there?")) { // ricevuto la domanda corretta
7.             theOutput = clues[currentJoke];    state = SENTCLUE; // inviata la risposta corrispondente
8.         } else { // ricevuto domanda diversa
9.             theOutput = "You're supposed to say \"Who's there?\"! Try again. Knock! Knock!"; // inviato suggerimento
10.        }
11.    } else if (state == SENTCLUE) { // inviata risposta
12.        if (theInput.equalsIgnoreCase(clues[currentJoke] + " who?")) { // ricevuta seconda domanda corretta
13.            theOutput = answers[currentJoke] + " Want another? (y/n)";    state = ANOTHER; // replica e turno finito
14.        } else { // ricevuta seconda domanda diversa, si ricomincia
15.            theOutput = "You're supposed to say \"" + clues[currentJoke] + " who?\"! Try again. Knock! Knock!";
16.            state = SENTKNOCKKNOCK;
17.        }
18.    } else if (state == ANOTHER) { // altro turno?
19.        if (theInput.equalsIgnoreCase("y")) {
20.            theOutput = "Knock! Knock!";
21.            if (currentJoke == (NUMJOKES - 1))
22.                currentJoke = 0;
23.            else
24.                currentJoke++;    state = SENTKNOCKKNOCK;
25.        } else {
26.            theOutput = "Bye.";    state = WAITING;
27.        }
28.    }
29.    return theOutput;
30. }

```

```
1.  public class KnockKnockServer {  
2.      public static void main(String[] args) throws IOException {  
3.          ServerSocket serverSocket = null;  
4.          try {  
5.              serverSocket = new ServerSocket(4444);  
6.          } catch (IOException e) {  
7.              e.printStackTrace();  
8.              System.err.println("Could not listen on port: 4444.");  
9.              System.exit(1);  
10.         }  
  
11.         Socket clientSocket = null;  
12.         try {  
13.             clientSocket = serverSocket.accept();  
14.         } catch (IOException e) {  
15.             System.err.println("Accept failed.");  
16.             System.exit(1);  
17.         }
```

```
18.     PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
19.     BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
20.     String inputLine, outputLine;
21.     KnockKnockProtocol kkp = new KnockKnockProtocol();

22.     outputLine = kkp.processInput(null);
23.     out.println(outputLine);

24.     while ((inputLine = in.readLine()) != null) {
25.         outputLine = kkp.processInput(inputLine);
26.         out.println(outputLine);
27.         if (outputLine.equals("Bye."))
28.             break;
29.     }

30.     out.close();
31.     in.close();
32.     clientSocket.close();
33.     serverSocket.close();
34. }
35. }
```



# KnockKnockClient

```
1. public class KnockKnockClient {
2.     public static void main(String[] args) throws IOException {
3.         Socket kkSocket = null;
4.         PrintWriter out = null;
5.         BufferedReader in = null;
6.         try {
7.             kkSocket = new Socket("localhost", 4444);
8.             out = new PrintWriter(kkSocket.getOutputStream(), true);
9.             in = new BufferedReader(new InputStreamReader(kkSocket.getInputStream()));
10.        } catch (UnknownHostException e) {
11.            System.err.println("Don't know about host: localhost:4444.");
12.            System.exit(1);
13.        } catch (IOException e) {
14.            System.err.println("Couldn't get I/O for the connection to: localhost:4444.");
15.            System.exit(1);
16.        }
17.        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
18.        String fromServer;
19.        String fromUser;
20.        while ((fromServer = in.readLine()) != null) {
21.            System.out.println("Server: " + fromServer);
22.            if (fromServer.equals("Bye.")) break;
23.            fromUser = stdIn.readLine();
24.            if (fromUser != null) {
25.                System.out.println("Client: " + fromUser);
26.                out.println(fromUser);
27.            }
28.        }
29.        out.close(); in.close(); stdIn.close(); kkSocket.close();
30.    }
31. }
```

- Limite delle soluzioni studiate: i client sono serviti in sequenza
  - Problemi di performance
  - Problemi di blocco del servizio
- È possibile realizzare server multi-thread per poter servire più clienti in modo concorrente
- Struttura logica della soluzione:

```
while (true) {  
    accept a connection ;  
    create a thread to deal with the client ;  
}
```

- Possiamo definire due classi
  - KKMultiServer che gestisce i thread
  - KKMultiServerThread che realizza un server dedicato ad un client

- I Thread sono flussi di controllo indipendenti.
- Verranno studiati più avanti con l'argomento concorrenza.

```
1.  public class KKMultiServer {  
  
2.      public static class KKMultiServerThread extends Thread {  
        ...  
29.    }  
  
30.    public static void main(String[] args) {  
31.        ServerSocket serverSocket = null;  
32.        boolean listening = true;  
  
33.        try {  
34.            serverSocket = new ServerSocket(4444);  
  
35.            while (listening)  
36.                new KKMultiServerThread(serverSocket.accept()).start();  
37.            serverSocket.close();  
38.        } catch (IOException e) {  
39.            System.err.println("Could not listen on port: 4444.");  
40.            System.exit(-1);  
41.        }  
42.    }  
43. }
```

```
2. public static class KKMultiServerThread extends Thread {
3.     private Socket socket = null;
4.     public KKMultiServerThread(Socket socket) {
5.         super("KKMultiServerThread");
6.         this.socket = socket;
7.     }
8.
9.     public void run() {
10.        try {
11.            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
12.            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
13.            String inputLine, outputLine;
14.            KnockKnockProtocol kkp = new KnockKnockProtocol();
15.            outputLine = kkp.processInput(null);
16.            out.println(outputLine);
17.
18.            while ((inputLine = in.readLine()) != null) {
19.                outputLine = kkp.processInput(inputLine);
20.                out.println(outputLine);
21.                if (outputLine.equals("Bye"))
22.                    break;
23.            }
24.
25.            out.close();
26.            in.close();
27.            socket.close();
28.        } catch (IOException e) {
29.            e.printStackTrace();
30.        }
31.    }
32. }
```