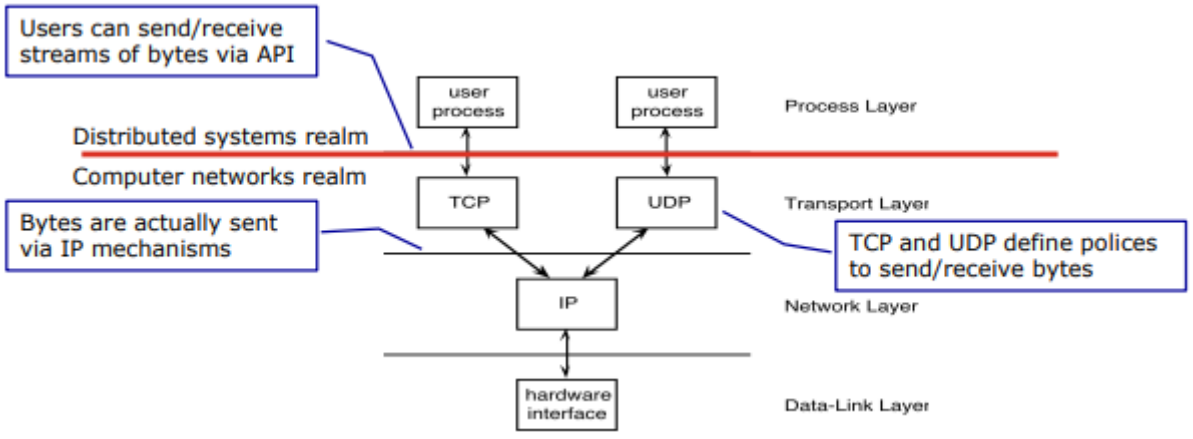
Stream-oriented communication

Comunicazione interprocesso

Modello ISO/OSI

> **Modello ISO/OSI**
>
> Modello di riferimento per le reti di computer. Definisce una struttura a sette livelli che descrive come i dati viaggiano attraverso una rete, dalla sorgente al destinatario.
>
> Descrive un modello per la trasmissione dei dati tra due end system mediante la rete.

Quando si parla di una trasmissione di dati tra processi, a livello dello strato del processo, gli user process si scambiano stream di byte mediante API (socket). Vengono gestiti mediante le politiche TCP o UDP a livello di trasporto e spediti effettivamente mediante il meccanismo IP (livello di rete) verso il livello di collegamento.



Se consideriamo l'ambito del Sistema Distribuito, nello strato del processo gli utenti possono inviare/ricevere un stream di byte (dati) mediante API (socket).

In una rete, due end system (hosts) ospitano i processi che eseguono le applicazioni.

Possono avvenire due modelli di scambio dei dati:

1. **Modello client/server**: client dell'host fa una richiesta e riceve un servizio da un server. *Esempio: email client/server, WWW client (browser)/server;*
2. **Modello peer-to-peer**: interazione simmetrica tra host (anche più di 2).
   Ogni dispositivo o nodo partecipante ha un ruolo equivalente e può agire sia come client che come server. Non ci sono nodi centrali che controllano o coordinano l'intera rete e ciascun nodo ha la capacità di inviare e ricevere dati direttamente con gli altri

nodi.
Esempio: teleconferenza.

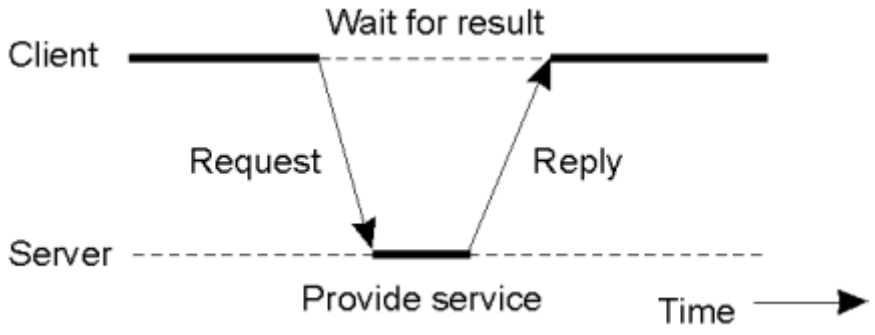
Modello Client-Server

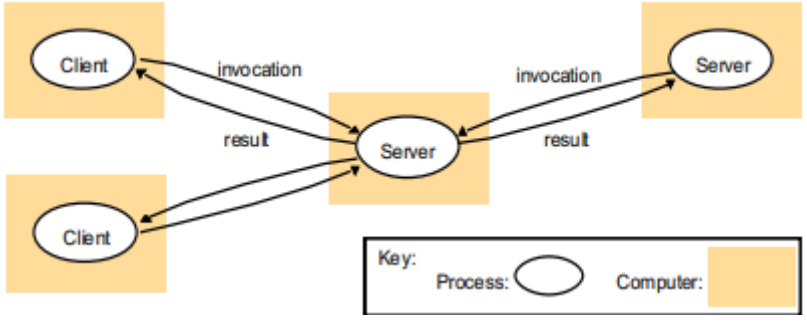
> **Modello client-server**
>
> Il modello client-server è un'architettura di rete in cui i dispositivi di rete o i programmi sono divisi in due categorie distinte: client e server.

Client: Il client è un'applicazione o un dispositivo che richiede e utilizza servizi o risorse da un server. I client inviano richieste ai server e ricevono le risposte per soddisfare le proprie esigenze.

Server: Il server è un'applicazione o un dispositivo che fornisce servizi o risorse ai client. Risponde alle richieste dei client, elaborandole e restituendo i risultati appropriati.

Nel modello client-server, i client e i server comunicano tra loro attraverso una rete utilizzando protocolli di comunicazione standard, come ad esempio HTTP, TCP/IP o UDP. I server sono in genere più potenti in termini di risorse computazionali e di archiviazione rispetto ai client, e sono progettati per gestire molteplici richieste simultanee da parte dei client.

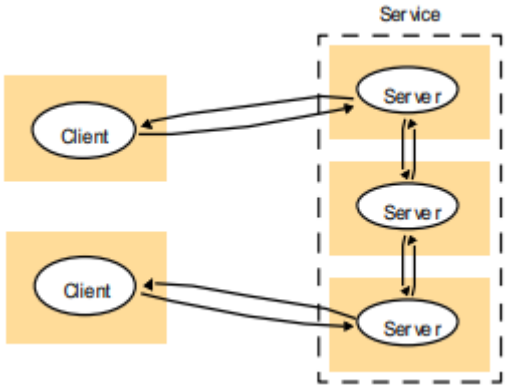




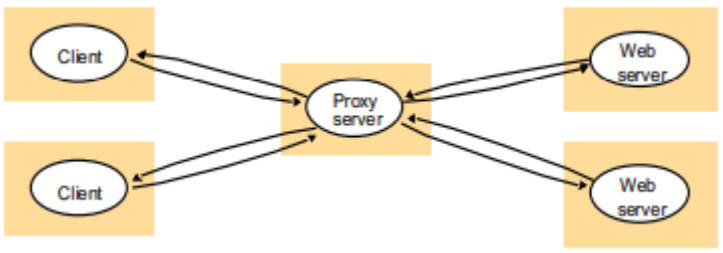
L'architettura di base prevede che un client acceda ad un server con una richiesta e che il server risponda con un risultato.

Configurazioni client/server

1. **Configurazioni client/server con accesso a server multipli**:
    - In questo tipo di configurazione, il client si connette direttamente a uno specifico server per accedere ai servizi o alle risorse offerti da quel server.
    - Il client deve essere configurato con le informazioni necessarie (come l'indirizzo IP o il nome del server) per stabilire una connessione diretta con il server desiderato.
    - Ogni server ospita un servizio specifico e i client interagiscono direttamente con il server appropriato per accedere a quel servizio.
    - Questo approccio è più comune in ambienti in cui i servizi sono distribuiti su server separati per scopi di scalabilità, ridondanza o specializzazione dei servizi.



2. **Accesso via proxy**:
    - Nel caso dell'accesso via proxy, il client non si connette direttamente al server finale che fornisce il servizio o la risorsa desiderata, ma invece si connette a un server proxy intermedio.
    - Il server proxy si comporta da intermediario tra il client e il server finale. Riceve le richieste dai client e le instrada al server finale, quindi inoltra le risposte dal server finale al client.
    - Il client non è necessariamente a conoscenza del server finale e può interagire solo con il server proxy.
    - Il server proxy può essere configurato per fornire diversi servizi, tra cui caching delle risorse, controllo degli accessi, filtraggio delle richieste e ottimizzazione delle prestazioni.



In sintesi, mentre le configurazioni client/server con accesso a server multipli richiedono che i client si connettano direttamente ai server desiderati, l'accesso via proxy implica che i client interagiscano solo con un server proxy, che si occupa poi di instradare le richieste ai server

finali. Entrambi gli approcci hanno le loro applicazioni specifiche e vantaggi, a seconda delle esigenze di scalabilità, sicurezza e gestione della rete.

Protocollo

> **Protocollo**
>
> Insieme di regole e convenzioni che governano la comunicazione e lo scambio di informazioni tra dispositivi, sistemi o entità in una rete o in un ambiente distribuito. Queste regole standardizzate definiscono il formato, l'ordine e il significato dei messaggi scambiati, così come i metodi per l'avvio, la gestione e la terminazione delle comunicazioni.

Esempio:

- *TCP/IP: vengono scambiati stream di byte di lunghezza infinita (meccanismo) che possono essere segmentati in messaggi (politica) definiti da un protocollo condiviso;*
- *HTTP - HyperText Transer Protocol;*
- *FTP - File Transfer Protocol;*
- *SMTP - Simple Mail Transfer Protocol.*

Per creare un applicazione, si deve definire:

- Il protocollo di comunicazione;
- La condivisione del protocollo tra gli attori dell’applicazione.

È importante quindi definire un protocollo di comunicazione e assicurarsi che tutti gli utilizzatori di un’applicazione condividano lo stesso protocollo.

Socket

I programmi vengono eseguiti dai processi, i quali comunicano tra di loro attraverso dei canali.

> **Canale**
>
> Un "canale" in una comunicazione interprocesso (IPC) è un meccanismo che permette a due o più processi di scambiare dati, messaggi o segnali tra loro su un sistema informatico. Questo canale può assumere diverse forme e utilizzare vari approcci, a seconda delle esigenze specifiche del sistema e dei processi coinvolti.
> I canali IPC sono essenziali per la costruzione di sistemi distribuiti e per la cooperazione tra processi su un sistema operativo.
>
> I dati possono essere in formato binario o testuali. Dall'esterno, ogni canale è identificato da un intero detto porta.

Esempio: schermo, tastiera, rete.

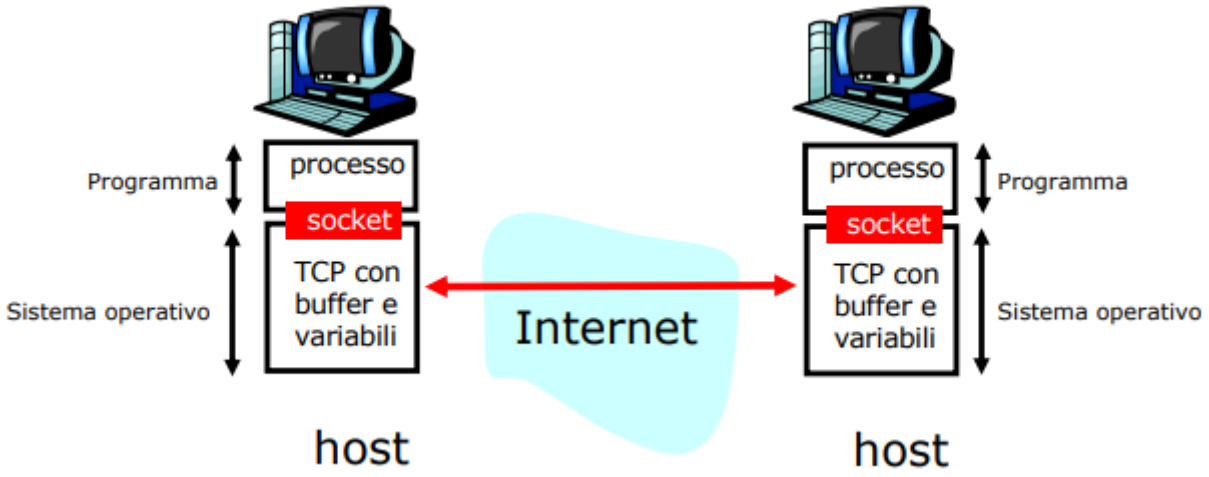
Socket

Particolari canali per la comunicazione tra processi che non condividono memoria (per esempio perchè risiedono su macchine diverse).

Un "socket" è un'interfaccia di programmazione (API) che rappresenta un'estremità di un canale di comunicazione bidirezionale tra processi. Essenzialmente, un socket fornisce un punto finale (API per accedere a TCP e UDP) attraverso il quale i processi possono inviare e ricevere dati, messaggi o segnali su una rete o all'interno di un sistema operativo.

In parole povere, due processi (applicazione nel modello client-server) comunicano inviando/leggendo dati in/da socket, sia sulla stessa macchina che su macchine diverse.

Esempio di socket di tipo stream (TCP)



Per potersi connettere o inviare ad un processo A, un processo B deve:

1. Conoscere la macchina (host) che esegue il processo A;
2. Conoscere la porta in cui A è connesso (well-known-port).

Riassunto Socket

Protocollo

- Di basso livello (flusso di byte/caratteri);
- L'applicazione si deve fare carico della codifica/decodifica dei dati;
- Non ci sono "messaggi" predefiniti: sono definiti a livello applicazione del progettista;

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)
- *Esempio: Superserver in Unix*

Aspetti importanti

Quando si parla di socket, si devono tenere conto dei seguenti problemi:

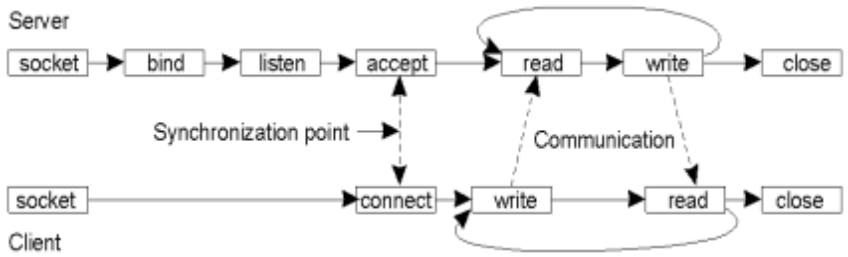
- **Gestione del ciclo di vita di client e server**
  Attivazione/terminazione del cliente e del server (es. Manuale o gestita da un middleware)
- **Identificazione e accesso al server**
  Informazioni che deve conoscere il cliente per accedere al server
  Come fa il client a conoscere l'indirizzo del server?
  - 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)
- **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 prestazione in relazione al carico (numero di clienti)

Socket per TCP/IP

La comunicazione TCP/IP avviene attraverso flussi di byte dopo una connessione esplicita tramite normali system call read/write.

- Sono sospensive (bloccano il processo finché il sistema operativo non ha effettuato la lettura/scrittura);
- Utilizzano un buffer (spazio di memoria dove trasferire i byte letti) di una certa dimensione (max caratteri che si possono leggere) per garantire flessibilità.

Immagine importante! In esame!!



Le socket trasportano flussi di bytes, quindi non ha fine. È la lettura/ scrittura che avviene per un numero arbitrario di byte. Quindi si devono prevedere cicli di lettura che termineranno in base alla dimensione dei "messaggi" come stabilito dal formato del protocollo applicativo in uso.

Pseudocodice della read

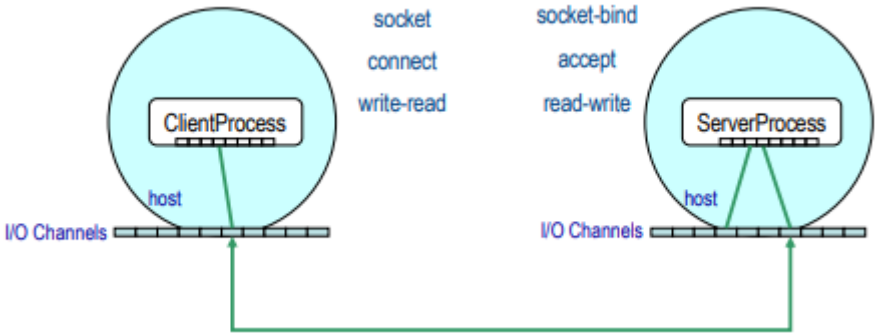
byteLetti read(*socket, buffer, dimBuffer*)

- *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

Primitiva	Funzione
Socket	Crea un nuovo punto di comunicazione
Bind	Assegna un indirizzo locale ad un socket
Listen	Annuncia la volontà di accettare nuove connessioni
Accept	Blocca i chiamanti fino a quando non arriva una richiesta di connessione
Connect	Tenta di stabilire una connessione
Write	Manda dei dati attraverso la connessione
Read	Riceve dei dati attraverso la connessione
Close	Lascia la connessione

Inizializzazione

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



Perché vengono utilizzate due socket diverse?

È un altro processo che elabora le richieste del clienti che si è connesso, mentre il processo principale continua ad accettare nuove richieste. Quindi il motivo legato al perché vengono usati due socket differenti è legato al fatto che un socket ha l'unico compito di accettare nuove richieste.

Socket in Java

java.net.Socket

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.

java.net.ServerSocket

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 if 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.

Progettazione di un applicazione con le socket

Architettura client

L’architettura è concettualmente è 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 problemi di sicurezza.

Architettura server

L’architettura generale prevedere che

- Venga creata una socket con una porta nota per accettare le richieste di connessione;

- Entri in un ciclo infinito in cui alterna
    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 principali

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.

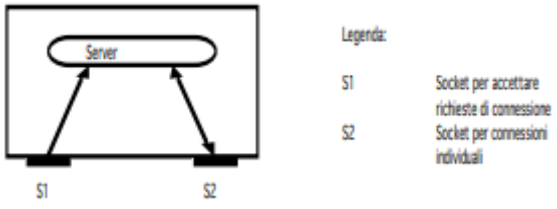
Architettura del server

I server possono essere:

- Iterativi: soddisfano una richiesta alla volta;
- Concorrenti processo singolo: simulano la presenza di un server dedicato;
- Concorrenti multi-processo: creano dei server dedicati;
- Concorrenti multi-thread: creano trend dedicati.

Server iterativo

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.

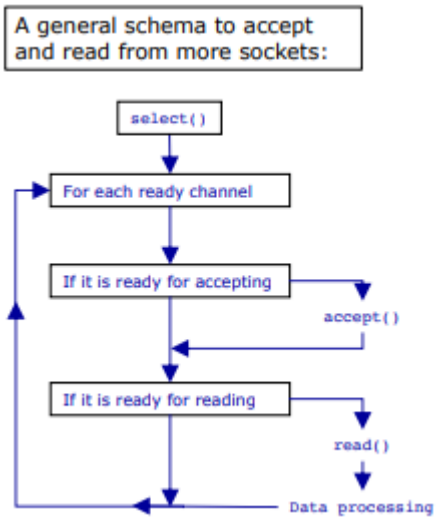


Vantaggi: semplice da progettare
Svantaggi:

- Viene servito un cliente alla volta, gli altri devono attendere;
- Un client può impedire l'evoluzione di altri client;
- Non scala.
  Soluzione: server concorrenti

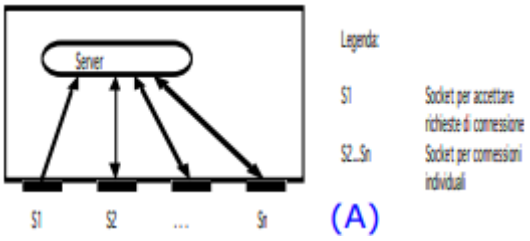
Server concorrente

Un server concorrente può gestire più connessioni client.



La sua realizzazione può essere:

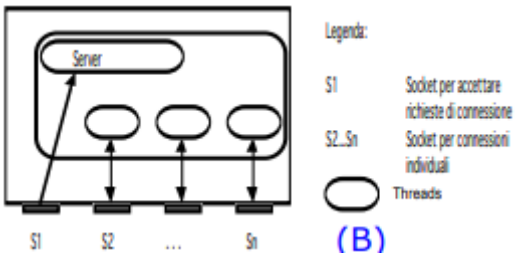
- Simulata con un solo processo:



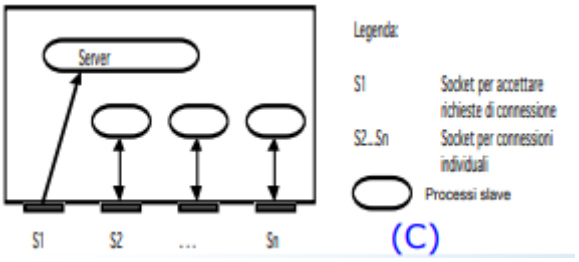
  - In C: funzione *select*
    Permette di gestire in modo non bloccante i diversi canali di I/O, sospendendo il processo finchè non è possibile fare una operazione di I/O.

    In Java: uso *Selector* che conosce i canali *ready to use*

  - In Java: uso dei *Thread*



- Reale creando nuovi processi slave
  In C: uso della funzione *fork*



Selector in Java

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()

Attenzione!! In esame!!

```
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.

Server multiprocesso

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.

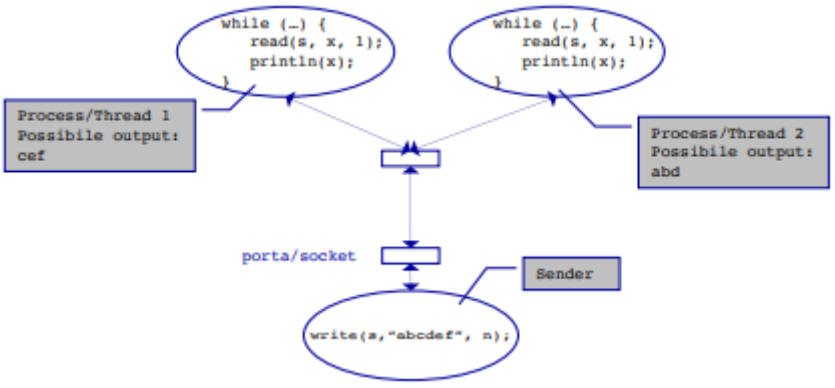
Il codice deve prevedere quindi che:

- 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.

Condivisione del canale

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



Creazione di un processo clone in Java

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

Questa classe viene utilizzata per creare processi del sistema operativo.

- Ogni istanza di ProcessBuilder gestiscee una collezione di attributi del processo;
- Il metodo start() crea una nuova istanza di Processo con tali attributi;
- Il metodo start() può essere invocato ripetutamente dalla stessa istanza per creare nuovi sottoprocessi con attributi identici o correlati.

```
ProcessBuilder pb = new
ProcessBuilder("C:\\Windows\\system32\\program.exe");
pb.inheritIO(); // ⟵ passes IO from forked process
try {
        Process p = pb.start(); // ⟵ forkAndExec on Unix
        p.waitFor(); // ⟵ waits for the forked process to complete
} catch (Exception e) {
        e.printStackTrace();
}
```

Server multi-thread

Un server concorrente che crea nuovi thread. Viene realizzato un server multi-thread per poter servire più clienti in modo concorrente.

La struttura del codice sarebbe:

1. Quando è possibile, accetta una connessione;
2. Per una connessione, creare un thread che si occupi della comunicazione con il client;
3. Ripeti il ciclo verificando la condizione.

Il server multi-thread risolve tutti i problemi riscontrati prima:

- I clienti vengono serviti in sequenza;
- Problemi di performance;
- Problemi di blocco del servizio.

Modelli a confronto

Monoprocesso (iterativo e concorrente)	**Multiprocesso**
• Gli utenti condividono lo stesso spazio di lavoro; • Adatto ad applicazioni cooperative che prevedono la modifica dello stato (lettura/ scrittura)	• Ogni utente ha uno spazio di lavoro autonomo; • Adatto ad applicazioni cooperative che non modificano lo stato del server (sola lettura); • Adatto ad applicazione autonome che modificano uno spazio di lavoro proprio (lettura/scrittura).