# Le Socket

### PROCESSI

Un processo NON è un programma in esecuzione. Il processo è un'astrazione che si occupa di eseguire i programmi (infatti non è che quando il programma finisce finisce anche il processo) gestito dal SO. Viene caricato in memoria e “attivato”. Contiene almeno un flusso di esecuzione

Uno stesso programma può dare origine a diversi processi:

* Diversi utenti eseguono lo stesso programma
* Uno stesso programma viene eseguito più volte, anche contemporaneamente, dallo stesso utente

Processo comunica attraverso canali a loro volta identificati da porte -> Processo identificato da IP - porta.

### SOCKET

La socket è un concetto astratto, serve a identificare (coppia host:porta) dei canali con certe caratteristiche. Funge da intermediario -> permette di identificare un processo all’interno di una rete e mandare stream di byte usando TCP/IP.

l’invio di stream bytes NON limita il tipo di messaggi da inviare.

PROCESSO che comunica con -> SOCKET che si interfaccia con -> TCP/IP

### ASPETTI CRITICI

* Gestione ciclo di vita
* Nelle socket l’identificazione delle risorse (**naming**) non è risolta, in quanto manca l’indirizzo dei componenti.
* Comunicazione client-server
* Distinguere client e server.

Non c’è proprio una sintassi o semantica in TCP/IP ma c’è nel protocollo.

trasparenza zero = l’applicazione (che siamo noi) deve occuparsi di tutto

**IDENTIFICAZIONE SERVER**

Ci sono tante alternative:

* Server costante, non si fa mai, è poco sicuro
* Chiedere all’utente l’indirizzo (web browser)
* DNS
* Adottare un protocollo (DHCP)

### COMUNICAZIONE VIA SOCKET

socket -> è system call, funzione sospensiva (ferma la computazione per switchare in kernel mode, il programma nel frattempo va messo in wait). La comunicazione avviene attraverso TCP/IP che decide quando e come inviare i dati.

In caso ci sia una fork() di un processo P1 (che apre una socket S1), creando P2, entrambi posso leggere S1 ma solo in modo concorrente.

La richiesta di creazione di una socket è fatta da un processo.

* system call e crea socket
* BIND = assegna un indirizzo
* LISTEN = rende la socket affidabile e raggiungibile
* ACCEPT = accetta nuove connessioni, si viene a creare una nuova socket dedicata al client (non per accettare). Nel frattempo il server si sospende fino all’arrivo di una nuova connessione.
* CONNECT = si connette, il formato del protocollo è stabilito da TCP/IP in quanto non siamo ancora a livello applicazione
* quando il processo diventerà un zombie verrà eliminato

Le socket sono sempre N+1. Se ho 3 processi client collegati allora ci saranno 4 socket aperte in quanto me ne serve almeno una per accettare nuove connessioni.

È possibile associare una porta nota ad un client.

**READ E WRITE**

Le chiamate READ/WRITE sono sospensive, bloccano il processo fino a quando il sistema operativo non ha completato l’operazione di lettura o scrittura.

Utilizzano un buffer. Per esempio la READ definisce un buffer per leggere N caratteri, ma potrebbe tornare avendone letti solo k<N.

Memoria non infinita (buffer, leggo 1000 byte per volta), se il pezzo dello stream è troppo grande non riesco a leggerlo. Come faccio a sapere quando finiscono i dati, è importante saperlo perché la socket è bloccante, allora blocco la cpu in kernel mode, devo sapere quando finisco. Poi devo fare un CICLO DI LETTURA per leggere tutti i dati che mi sono stati mandati.

eseguo WRITE = SOLO con un ciclo di istruzioni (non con una sola istruzione).

Come faccio a sapere quando finiscono i dati, è importante saperlo perché la socket è bloccante, allora blocco la cpu in kernel mode, devo sapere quando finisco. Uso i cicli di lettura.

semplificazione del buffer se si usano dei msg a lunghezza fissa.

**PROBLEMI**: Come faccio a sapere se il server mi invierà altri dati, devo continuare a stare in ascolto (non va bene).

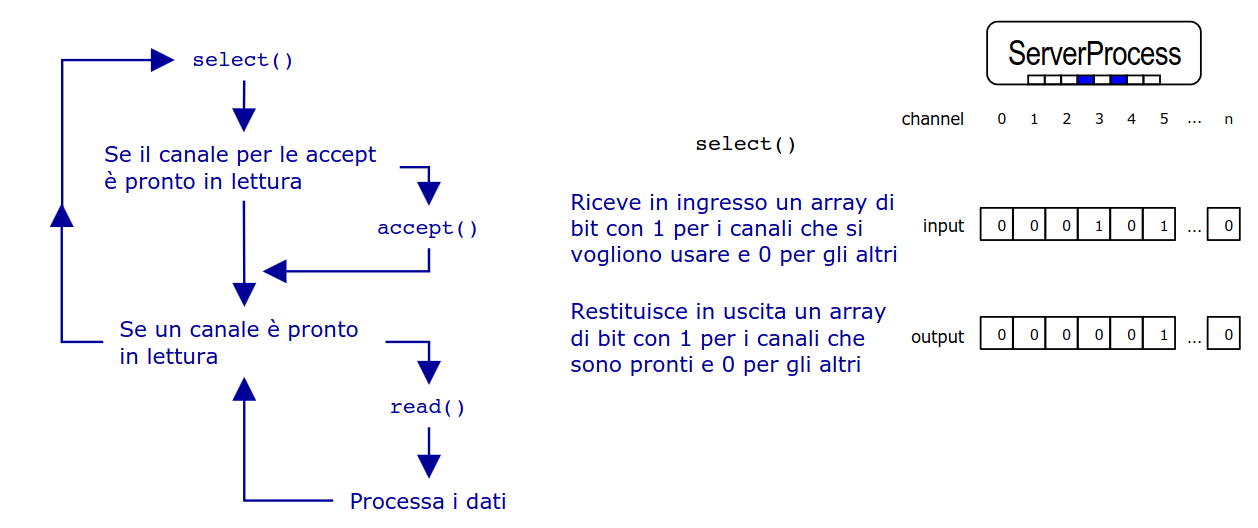
# Architetture Dei Server

I server possono essere iterativi -> soddisfano una richiesta alla volta, non usa la distribuzione, però è semplice da implementare.

### CONCORRENTI PROCESSO SINGOLO

Singolo processo. Avere più connessioni in contemporanea, però la WRITE/READ sono bloccanti quindi fanno perdere tempo. **Select** legge il canale SSE è pronto per essere letto, non blocca READ/WRITE, dice solo quando leggere il canale.

La select() permette di usare più canali di comunicazione. Serve quindi per effettuare operazioni non bloccanti.



In Java esistono i **channel**. Servono per le socket (non le sostituiscono). Possono essere bloccanti (non hanno senso) o non bloccanti (non mette il canale in sleep e l’operazione restituisce subito). I canali non bloccanti si possono usare con i selector di Java.

### CONCORRENTI MULTI-PROCESSO

Creano server dedicati. Faccio la fork() dei processi, creandone di nuovi (slave) che non condividono più la memoria, ereditano i canali di comunicazione ed eseguono 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

Quindi ogni figlio gestisce un client. Questo è un server iterativo, in quanto ho fatto tutto in una singola iterazione.

Se faccio una fork e mi dimentico di chiudere un canale potrei avere un problema di concorrenza (più processi READ/WRITE su una stessa socket).

Nel multi-thread e multi-processo ogni processo gestisce una socket, quindi è semplice da implementare. Nei singoli-processo è molto più complicato.

### CONFRONTO TRA MODELLI

Server mono-processo

* stessa memoria condivisa
* se devo leggere/scrivere in un contesto cooperativo

Server multi-processo

* applicazioni di sola lettura/contesto cooperativo ma che non modifica lo stato dei server
* oppure contesto autonomo in spazio di lavoro autonomo

### CONCORRENTI MULTI-THREAD

Un multi thread è molto efficiente quando non ci sono risorse condivise da gestire.

Creano thread dedicati. Usano i thread per creare dei processi slave. Però se condividono tutti i processi, condividono anche la memoria. Sono sistemi concorrenti ma non distribuiti.

Li vediamo meglio in [3. Concorrenza](https://docs.google.com/document/u/2/d/1Dv-DqA02cVfUGtLH7rYivmhqdnT_TVJfEaHJA7uzot4/edit)

# LE SOCKET IN JAVA

**java.net.Socket** usato per stabilire connessione con un server  
**java.net.ServerSocket** è una socket lato server, si usa per accettare le richieste di connessione dai clienti.

* **connect(SocketAddress, timeout)**: connette il socket al server, con uno specifico valore di timeout
* **InputStream getInputStream()**: restituisce lo stream di input legato al socket
* **OutputStream getOutputStream()**: restituisce l’output stream legato al socket per scrivere bytes.