Stream-oriented communication

2.1 Comunicazione interprocesso

2.1.1 Socket

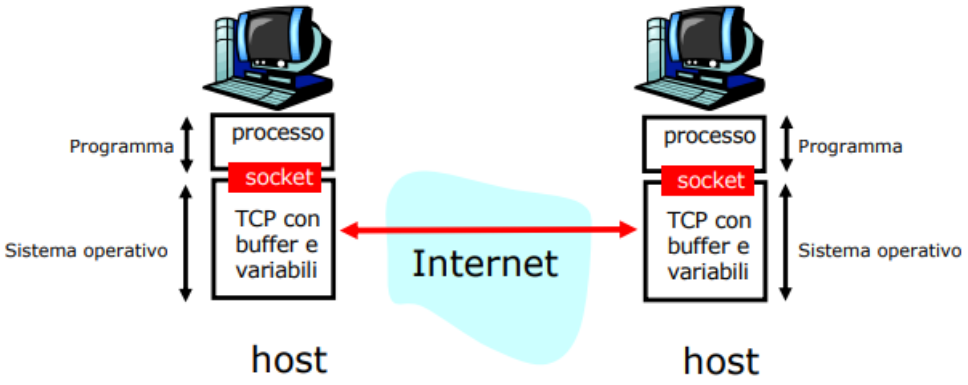
I programmi vengono eseguiti dai processi, i quali comunicano tra di loro attraverso dei canali.

> **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:

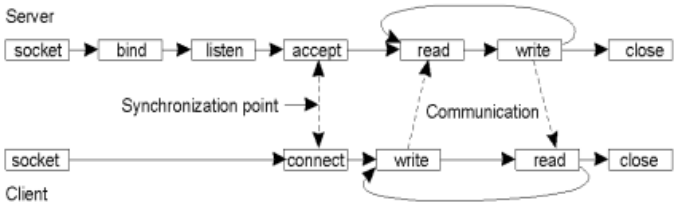
- **Gestione del ciclo di vita di client e server**
  Attivazione/terminazione del cliente e del server (es. Manuale o gestita da un middleware)
- **Identificazione e accesso al server**
  Informazioni che deve conoscere il cliente per accedere al server
  Come fa il client a conoscere l'indirizzo del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

2.1.1.1 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!!



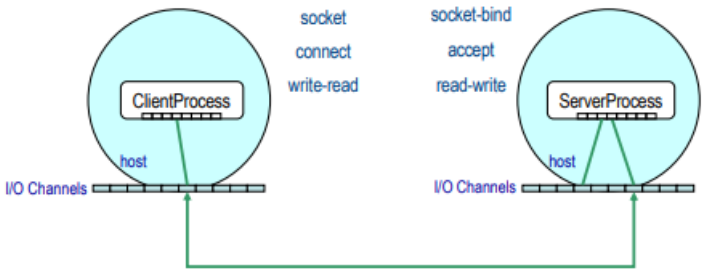
Le socket trasportano flussi di bytes, quindi non ha fine. È la lettura/ scrittura che avviene per un numero arbitrario di byte. Quindi si devono prevedere cicli di lettura che termineranno in base alla dimensione dei “messaggi” come stabilito dal formato del protocollo applicativo in uso.

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

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.

Progettazione di un applicazione con le socket

Architettura client	Architettura server
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.	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**.

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.

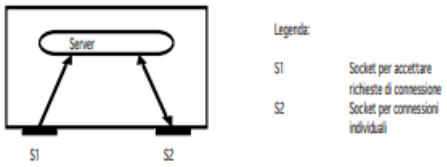
2.1.2 Architettura del server

I server possono essere:

- **Iterativi**: soddisfano una richiesta alla volta;
- **Concorrenti mono-processo**: simulano la presenza di un server dedicato;
- **Concorrenti multi-processo**: creano dei server dedicati;
- **Concorrenti multi-thread**: creano thread 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 well-known-port per essere successivamente soddisfatte.



Vantaggi: semplice da progettare.

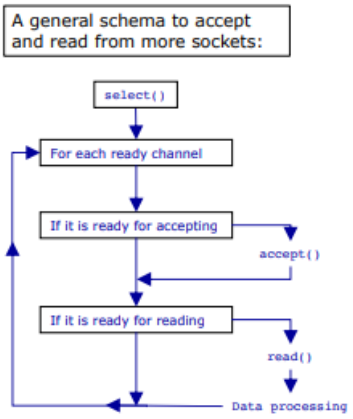
Svantaggi:

- Viene servito un cliente alla volta, gli altri devono attendere;
- Un client può impedire l'evoluzione di altri client;
- Non scalabile.

Soluzione: server concorrenti

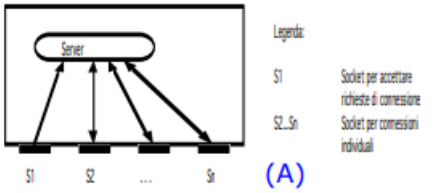
Server concorrente

Un server concorrente può gestire più connessioni client.



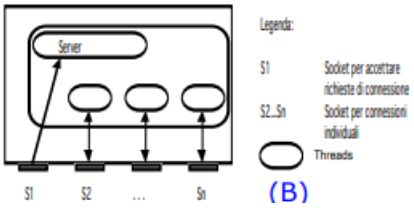
La sua realizzazione può essere:

1. **Simulata con un solo processo**:

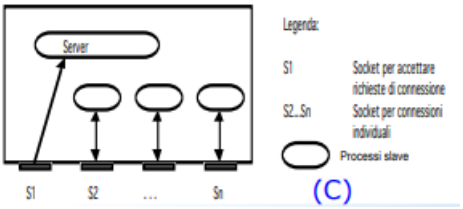


   - In C: funzione *select*
     Permette di gestire in modo non bloccante i diversi canali di I/O, sospendendo il processo finchè non è possibile fare una operazione di I/O.

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



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



I/O non bloccante - select()

Le operazioni di lettura e scrittura comportano l'uso di system call bloccanti, ovvero 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).

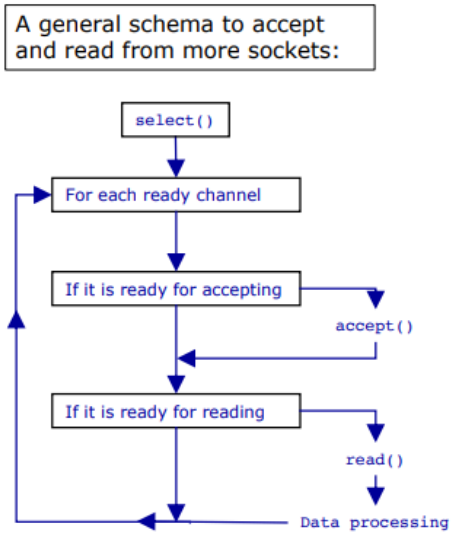
1. Dico al sistema quali canali voglio usare in modalità non bloccante
2. Chiamo la select() che controlla quali canali sono «pronti»
3. Sui canali pronti effetto l'operazione read() o write() desiderata
4. Ciclo tornando al punto 1

In Java sono stati introdotti i channel che possono operare in modo bloccante o non bloccante.

1. Un canale non-bloccante non mette il chiamante in sleep
2. L'operazione richiesta o viene completata immediatamente o restituisce un rusultato 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.

I canali socket in modo non bloccante possono essere usati con i selector.
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.

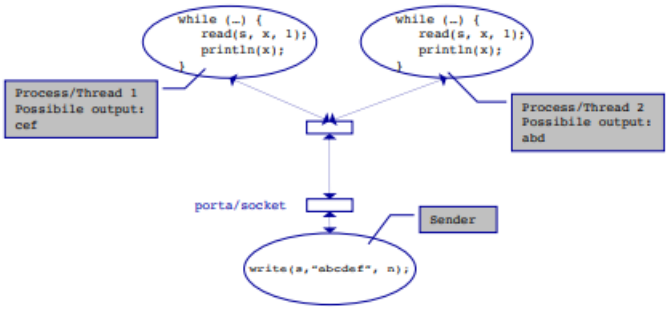
Il codice deve prevedere quindi che:

- 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).