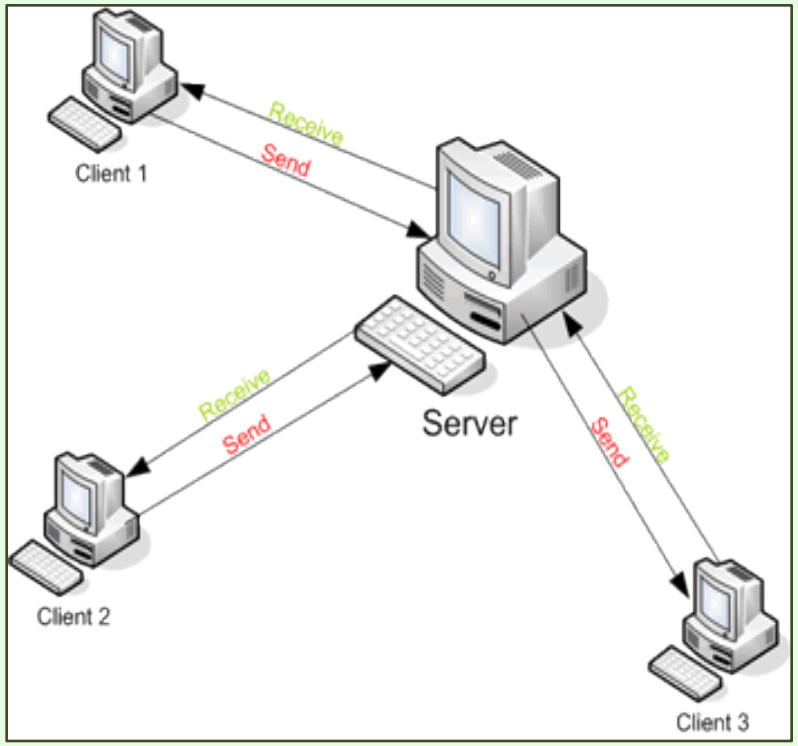
TCP concorrente
in ambiente Windows
(1)

Server concorrente

L'interazione **client-server**, di norma, non si limita ad una richiesta singola ma coinvolge un dialogo complesso, fatto di **tante interazioni**, cioè **molti client** che chiedono un certo servizio ad un server.

Dal punto di vista implementativo è molto più facile realizzare un **server concorrente** attraverso **tanti processi** (tanti thread) **indipendenti**, *ciascuno dedicato al servizio di un solo client*, che cercare di realizzare un solo processo capace di servire contemporaneamente tanti client.

Si può pensare ad un negozio nel quale un cliente, quando viene servito, ha una persona dedicata a farlo. Se tanti clienti sono serviti contemporaneamente è perché ci sono altrettanti commessi che lavorano in parallelo nel negozio, e ogni cliente e servito da un commesso dedicato.



Server sequenziali e concorrenti

Si possono immaginare due **modalità operative** che possono essere utilizzate dal server nel rapportarsi con i suoi client:

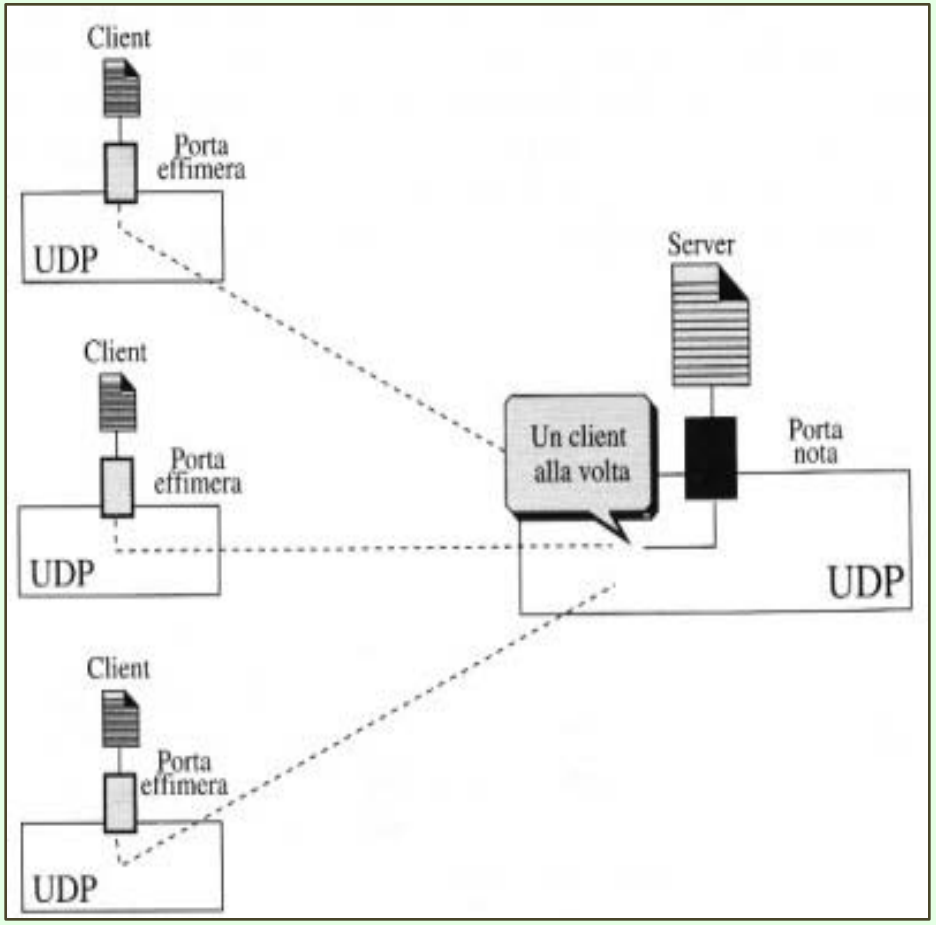
- **sequenziale** (o **iterativo**)
- **concorrente** (o **ricorsivo**).

Modalità sequenziale

Quando il **server** entra in colloquio con un client **termina di servirlo** e **chiude la connessione** con lui **prima di accettare esplicitamente una eventuale altra connessione** richiesta da un altro client.

Questo è il caso tipico in cui:

- le **richieste** e le **risposte** *impegnano* **un solo pacchetto IP**, come nel **trasporto UDP**
- la generazione delle **risposte** dura un **tempo trascurabile**.



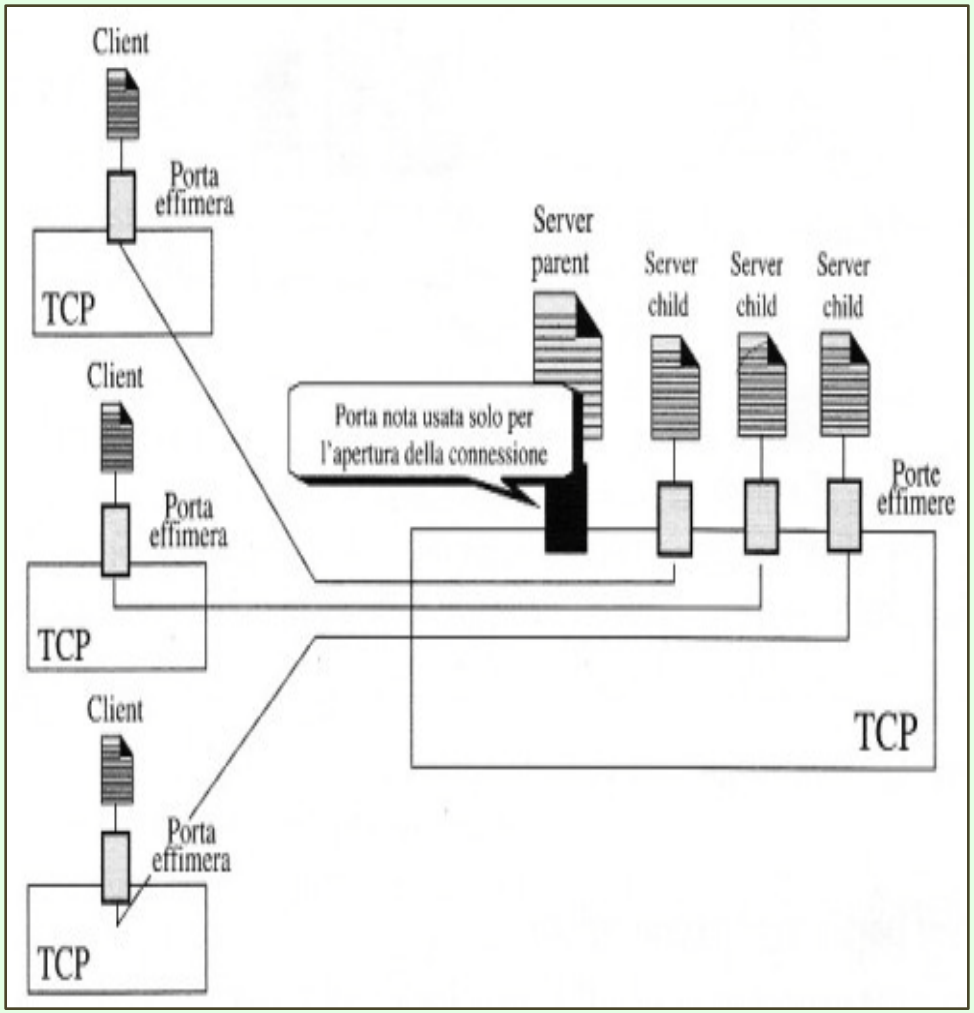
Server sequenziali e concorrenti

Modalità concorrente

Quando il **server** entra in colloquio con un client, **si duplica**:

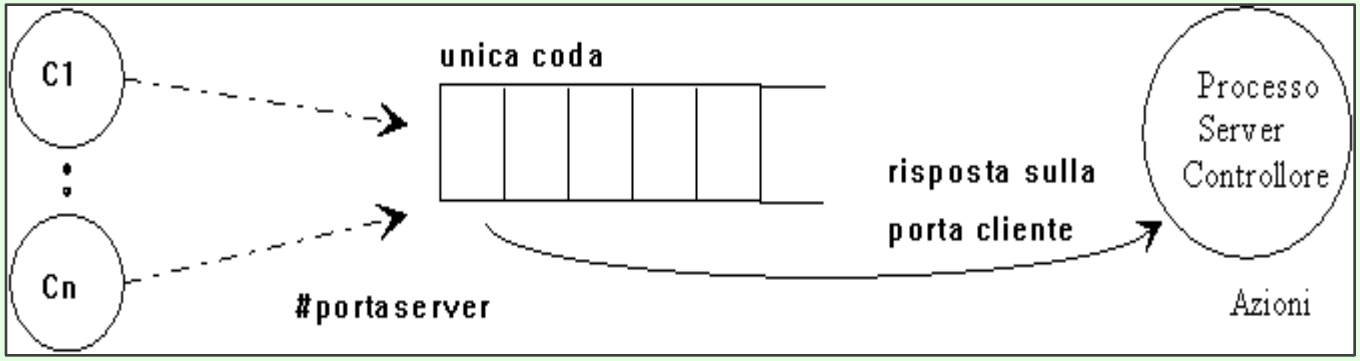
- **Una delle due copie continua a servire il client** fino al termine della sessione, quindi chiude la relativa connessione e termina.
- **L'altra copia si rimette in attesa di nuove richieste di connessione** da parte di altri client.

In genere **tutti i server TCP** sono **concorrenti**.



Server sequenziali

Server sequenziale senza connessione (uso di UDP): servizi senza stato (che introduce ritardi) e non soggetti a guasti.

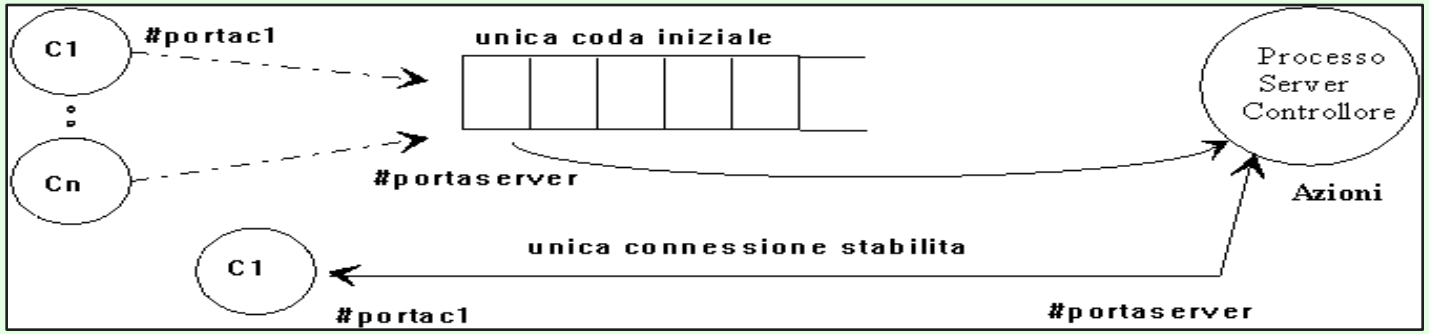


Il *server sequenziale non orientato alla connessione*, in realtà ***non processa i client davvero sequenzialmente***.
Ogni richiesta (messaggio ricevuto) e trattata indipendentemente dalle altre perché il server è stateless, senza stato, cioe senza memoria.
Il trattamento di richieste successive di client diversi è alternato.
Per realizzare una interazione veramente sequenziale il server dovrebbe concentrare la sua attenzione su un client per volta, e trattare tutte le richieste provenienti da un client prima di prendere in considerazione il client successivo, ma un server, che utilizza il protocollo UDP a livello trasporto, ha il socket associato alla porta e non ad una particolare connessione della porta (e quindi ad un particolare client) e su quella porta (well known del servizio) chiunque e in grado di inviare messaggi e quindi di inframmezzare le proprie richieste a quelle del client servito in quel momento.

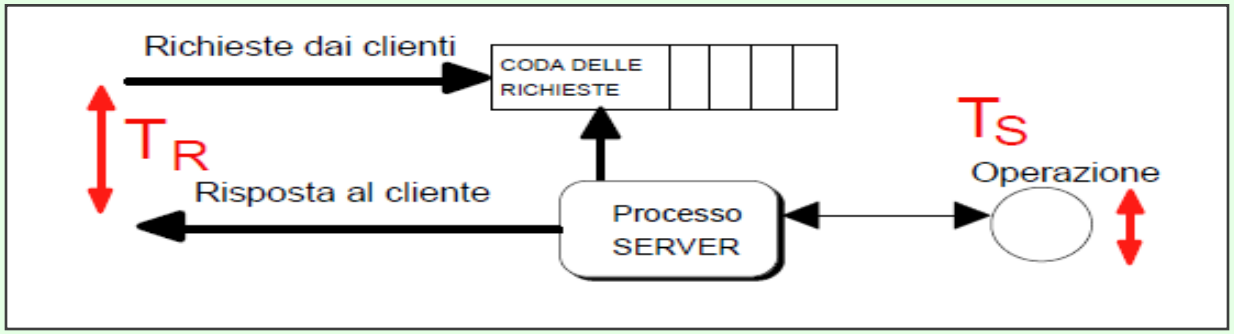
Per realizzare un server che esamini le richieste dei client in modo sequenziale occorre che **il server possa filtrare i messaggi che riceve sulla sua porta** e questo può essere fatto a livello applicativo.

Server sequenziali

Server sequenziale con connessione: servizi con requisiti di affidabilità (uso di TCP)



In questo caso per servire il singolo client **si può utilizzare una porta effimera** e la **porta well-known** sarà utilizzata **solo per accettare nuovi clienti**.

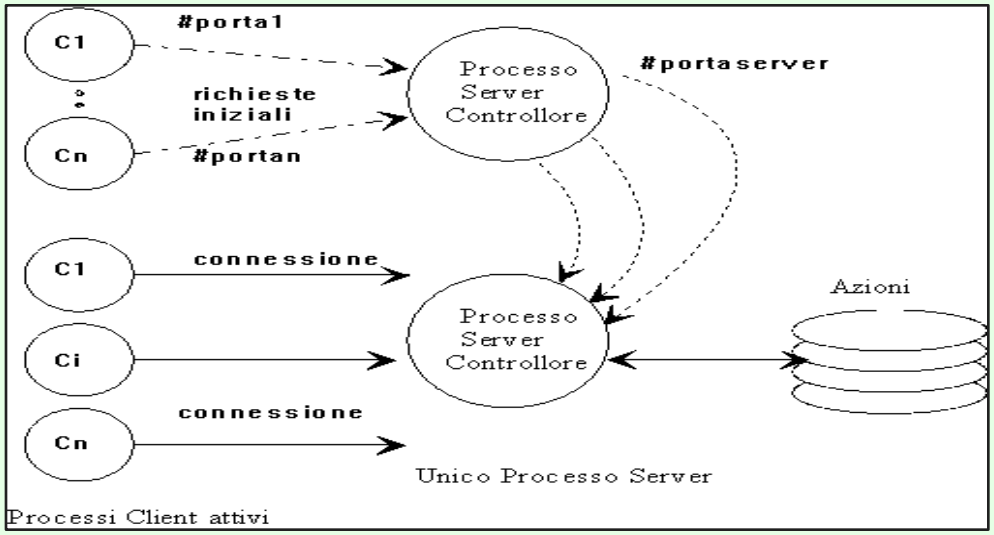


Dove: **TR** è il **ritardo totale** tra la spedizione della richiesta e l'arrivo della risposta dal server
TS è il **tempo di servizio** di una richiesta isolata

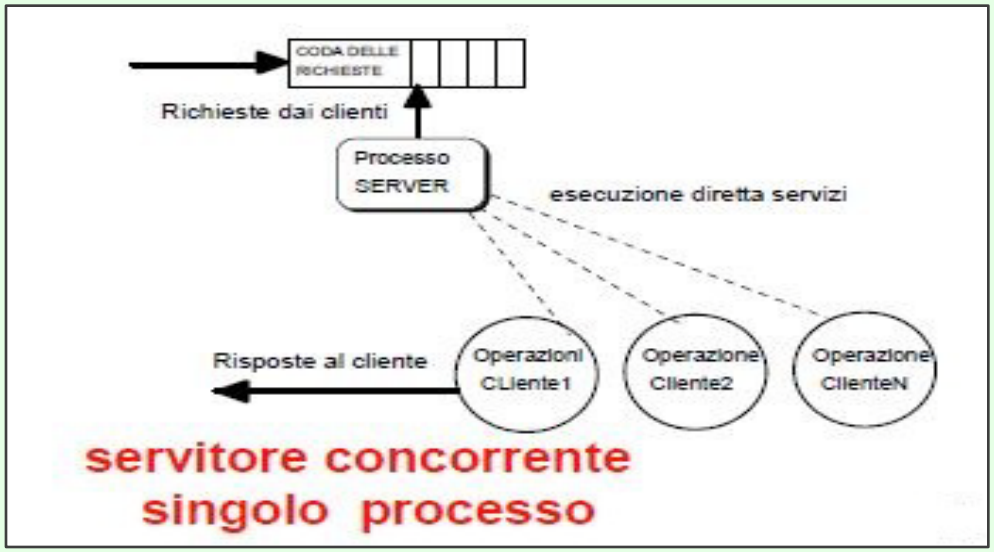
Soluzioni per **limitare l'overhead**: **limitare la lunghezza della coda** (sveltendo il servizio)
rifiutare le richieste a coda piena (rifiutando il servizio)

Server concorrenti

Server concorrente monoprocesso: un **unico processo server** si divide tra il servizio della coda delle richieste e le operazioni vere e proprie.



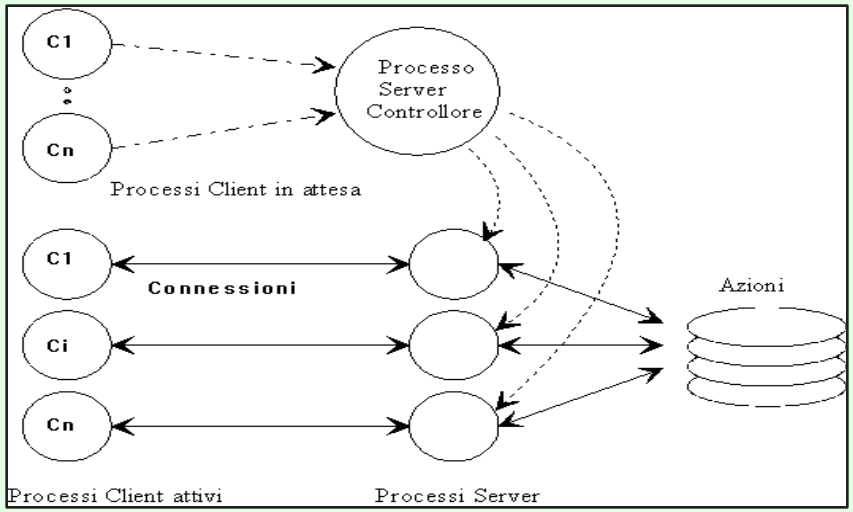
può essere orientato o no alla connessione

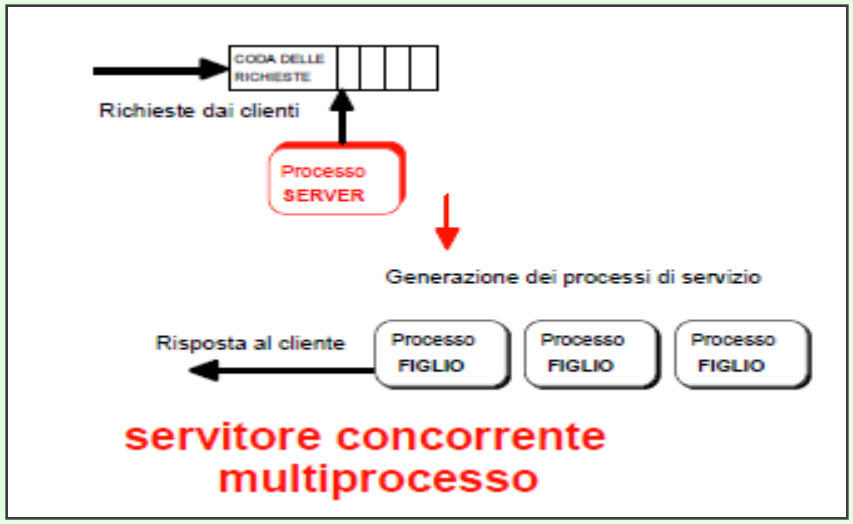


Un **unico processo server** che deve considerare tutte le richieste e servirle al meglio mantenendo le connessioni.

Server concorrenti

Server concorrente multiprocesso: un processo server si occupa della coda delle richieste e **genera processi figli** (thread), uno per ogni servizio. In questo caso si hanno quindi **processi multipli** consentendo l'ottimizzazione dell'uso del processore *eliminando i tempi di idle.*
Si deve garantire che il costo di generazione del processo non ecceda il guadagno ottenuto.





Server concorrente parallelo

Può essere orientato o no alla connessione

Il processo server si limita ad accettare una connessione, a creare il figlio e a passargliela. Il suo comportamento e identico per tutti i diversi servizi applicativi.
Si può pensare di avere un unico padre che attende le richieste dei client per tutti i servizi applicativi e che attiva poi un figlio opportuno a seconda del servizio richiesto.
Nel caso di **socket TCP**, a parte il socket legato alla connessione con il client, non c'è altro scambio di informazione tra padre e figlio.
Nel caso di **socket UDP** invece il server padre riceve un datagram e deve farlo avere al processo figlio. Il figlio sembra dover essere parte del padre (per poter ricevere il datagram) per condividere e sincronizzare tra padre e figlio l'accesso al socket (well known) che contiene ancora il primo datagram del client.

TCP concorrente in Linux

Server con s.o. Linux (rif.: esempio pagg. 76 e 80)

```
while (1)
    {
     communication_socket_id = accept(request_socket_id, (struct sockaddr*)&client_add, &client_add_size);
     if (communication_socket_id >= 0)
         {
          // connessione effettuata: creazione nuovo processo  per la gestione della comunicazione con il
                                     processo client
          pid = fork( );   // duplicazione del processo
          if (pid == 0)
              {  // processo figlio e  con le funzioni getpid() e getppid() si ottengono rispettivamente
                 //                    il processid del figlio e quello del padre
                 client_service(communication_socket_id);   // gestione della comunicazione con il client
                 close (communication_socket_id);
                 return 0;
               } // se pid !=0  allora è il processo padre e pid è il processid del figlio e getpid() restituisce  quello del padre

          close (communication_socket_id);
         }
     }
```

Una volta stabilita la connessione (accept eseguita con successo) il **processo** esegue una **fork( )** che gli consente di **creare in memoria una copia di se stesso.**

Il **processo figlio** viene dedicato a *servire la connessione* appena instaurata, il **processo padre** *attende nuove richieste di connessione* sulla stessa porta.

Questo avviene in genere nei Server TCP, come ad esempio per gli web server.

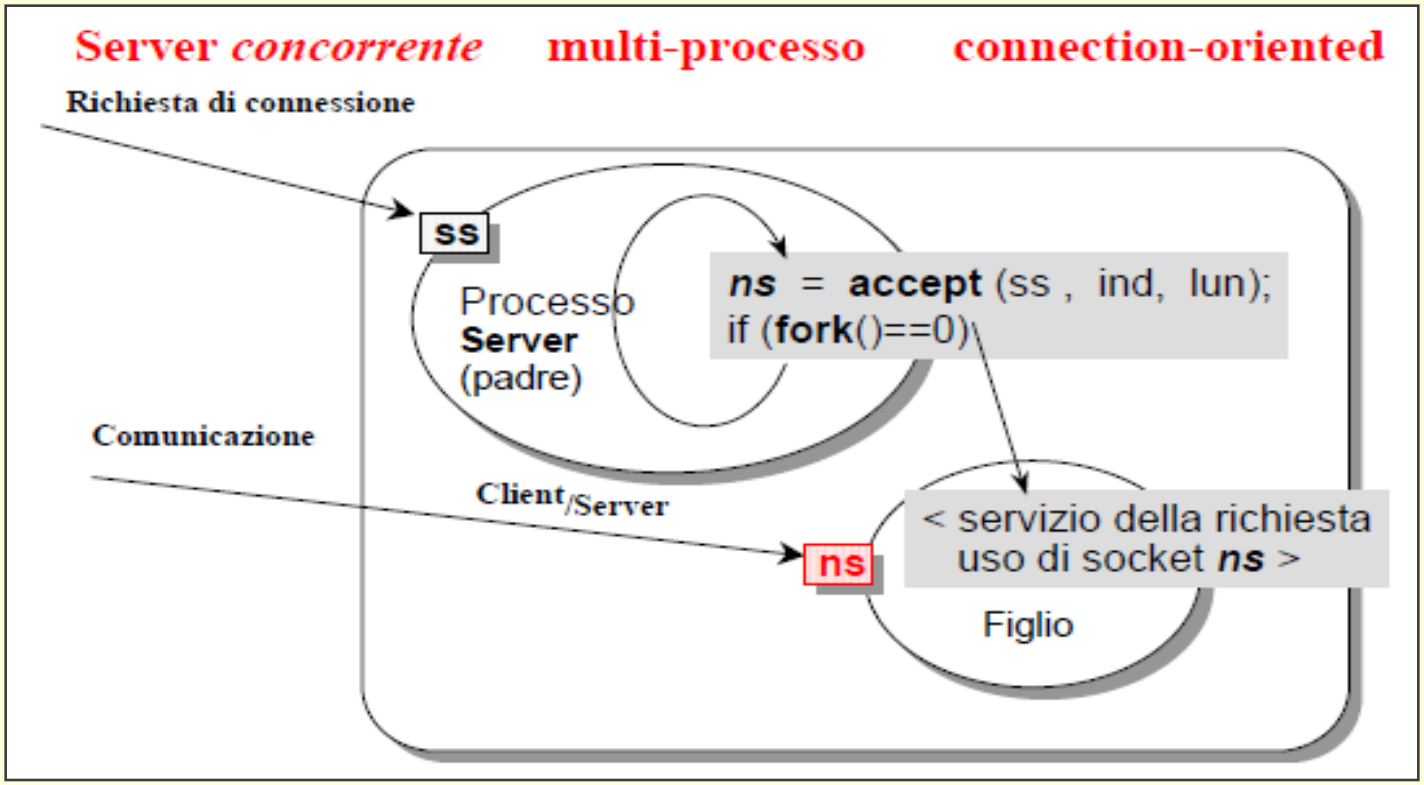
TCP concorrente in Linux

Server con s.o. Linux (rif.: esempio pagg. 76 e 80)

Il ***nuovo socket*** connesso permette di scindere le funzionalità del server, tra:

- **accettazione dei servizi**
- **connessioni in atto.**

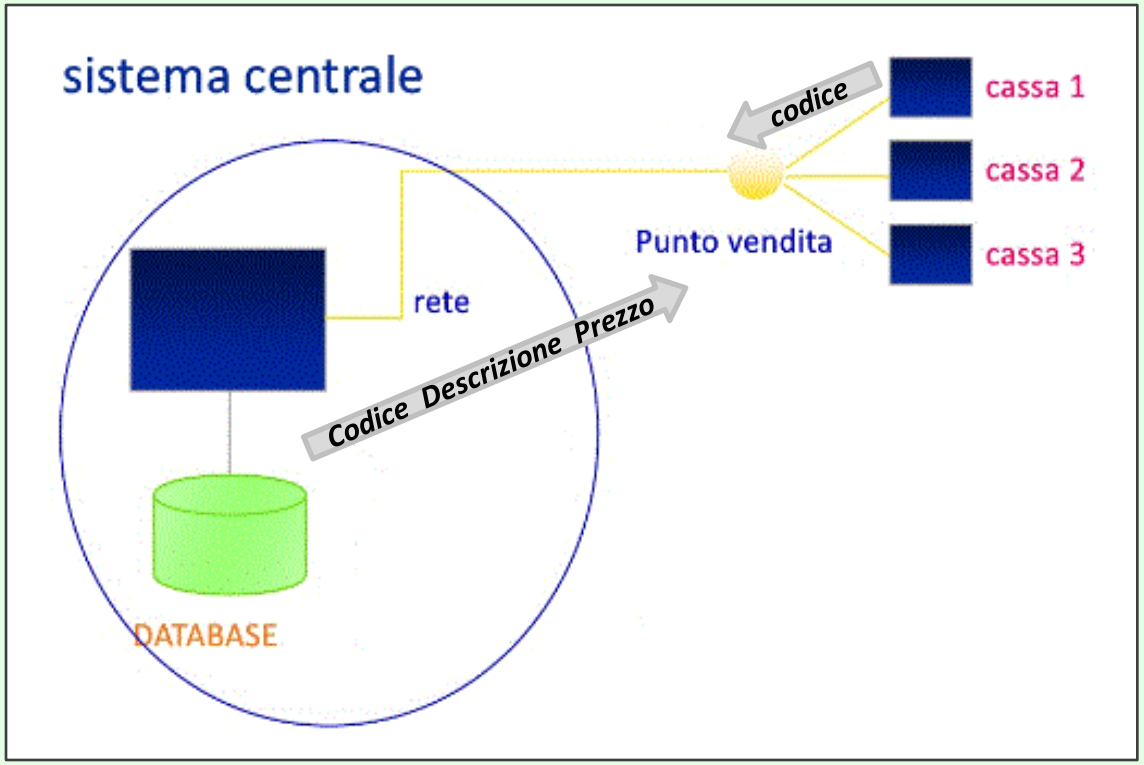
Ogni *nuovo socket* rappresenta un **servizio separato e separabile**



TCP concorrente in ambiente Windows

Esercizio (rif.: esempio pagg. 83÷87)

Le singole casse di un supermercato sono client TCP connesse a un unico server che riceve le richieste contenenti il *codice di un prodotto* e risponde con la *descrizione e il prezzo relativi a quel codice*. La gestione delle informazioni relative ai singoli prodotti in vendita viene effettuata da parte del server TCP.



TCP concorrente in ambiente Windows

Server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <process.h>
#include <winsock2.h>
#include  "product.c"   <-----// Dichiara la struttura PRODUCT  e  due funzioni: per caricare da file
                              // tutti i prodotti  in una tabella allocata dinamicamente e per
                              // ricercare un prodotto per codice

// thread per la gestione della comunicazione con il client
unsigned long WINAPI  client_service(void* arg)
{  …
   …
}

int main(int argc, char* argv[])
{
    WSADATA wsaData;
    SOCKET request_socket_id;            // socket richieste di connessione
    SOCKET communication_socket_id;      // socket comunicazione con client
    struct sockaddr_in server_add;       // struttura per indirizzo server
    struct sockaddr_in client_add;       // struttura per indirizzo client
    size_t   client_add_size;

    int   n_product;                     // numero prodotti caricati
```

Code, Description, Price

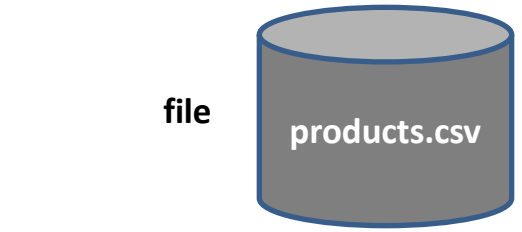
// Funzione eseguita dal thread che serve il generico client (cassa del supermercato) che richiede le informazioni relative ad un prodotto.

TCP concorrente in ambiente Windows

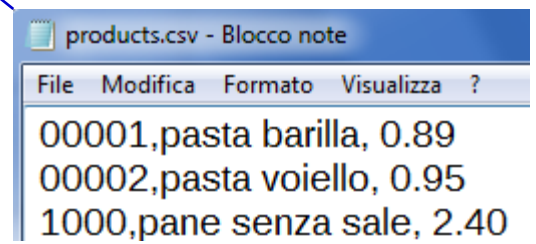
. . . Server.c

```
// caricamento prodotti da file
if ((  n_product  =  loadProducts("products.csv")) <= 0)
      {
        printf("Errore caricamento prodotti!\r\n");
        system("pause");
        return -1;
      }
// inizializzazione WinSock (versione 2.2)
if (WSAStartup(0x0202, &wsaData) != 0)
      {
        printf("Errore inizializzazione WinSock!\r\n");
        system("pause");
        return -1;
      }
// apertura del socket
if ((request_socket_id = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP)) == INVALID_SOCKET)
      {
        printf("Errore apertura socket!\r\n");
        WSACleanup();
        system("pause");
        return -1;
      }
```

. . .



Code, Description, Price



TCP concorrente in ambiente Windows

. . . Server.c

```
// costruzione della struttura per l'indirizzo del server
 memset(&server_add, 0, sizeof(server_add));
 server_add.sin_family = AF_INET;
 server_add.sin_addr.s_addr = 0;              // indirizzo IP locale
 server_add.sin_port = htons(23365);          // numero di porta server

 // associazione del numero di porta al socket
 if (bind (request_socket_id, (struct sockaddr *)&server_add, sizeof(server_add))   ==   SOCKET_ERROR)
        {
              printf("Errore associazione socket!\r\n");
              closesocket(request_socket_id);
              WSACleanup();
               system("pause");
              return -1;
         }

// impostazione del socket per ricevere le richieste di connessione
 if (listen(request_socket_id, 1)   ==   SOCKET_ERROR)
        {
              printf("Errore impostazione socket!\r\n");
              closesocket(request_socket_id);
              WSACleanup();
              system("pause");
              return -1;
        }
 printf("Servizio attivo (caricati %i prodotti)...\r\n", n_product);
```

. . .

TCP concorrente in ambiente Windows

. . . Server.c

```
while (1)
     {
       client_add_size = sizeof(client_add);
       communication_socket_id = accept (request_socket_id, (struct sockaddr*)&client_add, &client_add_size);
       if (communication_socket_id   !=   INVALID_SOCKET)
                {
                  // connessione effettuata
                  // creazione nuovo thread per la gestione della comunicazione con il processo client
                  CreateThread(NULL, 4096, &client_service, &communication_socket_id, 0, NULL);
                }
      }
closesocket (request_socket_id);
WSACleanup();
return 0;
}
```

TCP concorrente in ambiente Windows

. . . Server.c - funzione **client_service**(…)

```
// thread per la gestione della comunicazione con il client  (servizio al client)
unsigned long   WINAPI   client_service (void* arg)
{
    SOCKET socket_id = *(SOCKET*)arg;
    char buffer[64];                // buffer per ricezione
    char command[256];              // comando ricevuto dal client (codice prodotto)
    char price[16];
    int index;                      // indice per costruzione stringa di comando
    int i, n;
    struct PRODUCT product;         // variabile per ricerca prodotto

    index = 0;
    while (1)
                {
                   // il server è in attesa della richiesta del client che ora è connesso
                   if ( ( n = recv (socket_id, (void*)buffer, sizeof(buffer), 0))   <= 0)
                                {
                                   // chiusura della connessione da parte del client
                                   closesocket(socket_id);
                                   break;
                                }
```

. . .

TCP concorrente in ambiente Windows

. . . **Server.c** - funzione **client_service**(…)

```
else
    { // ricezione caratteri: costruzione del comando
      for (i=0; i<n; i++)
      if (buffer[i] == '\r' || buffer[i] == '\n')
        {
          // comando completato: inserimento del terminatore di stringa
          command[index] = '\0';
          if (strlen(command) > 0)
                {
                  // codice per richiesta informazioni prodotto
                  printf("... ricevuta richiesta per codice: %s.\r\n", command);
                  // ricerca prodotto  il cui codice è   in    command
                  if (findProduct(command, &product) > 0)
                       { // prodotto trovato: costruzione risposta
                          strcpy(buffer, product.code);
                          strcat(buffer, ",");
                          strcat(buffer, product.description);
                          strcat(buffer, ",");
                          sprintf(price, "%.2f\r\n", product.price);
                          strcat(buffer, price);
                          // trasmissione stringa di risposta
                          send(socket_id, (void*)buffer, strlen(buffer), 0);
                          printf("... risposta con prodotto: %s\r\n\r\n", buffer);
                       }
                    else
                       {  printf("... nessun prodotto trovato!\r\n\r\n");
                          strcpy(buffer, "Codice non trovato\r\n");
                          send(socket_id, (void*)buffer, strlen(buffer), 0);
                       }
                }
          index = 0;
          break;
        }
```

buffer (dati provenienti dal client)

00001 \n

Ricopiati

command

00001 \0

product (puntatore a record)

00001	pasta barilla	0.89

buffer (dati da inviare al client)

00001,pasta barilla,0.89 \r\n

. . .

TCP concorrente in ambiente Windows

. . . Server.c - funzione **client_service**(…)

```
            else
                {
                    // copia del carattere nel comando
                    if (index >= sizeof(command))
                            index = 0;
                    command[index] = buffer[i];
                    index++;
                 }

            } // fine else

        } // fine while

    closesocket(socket_id);
    ExitThread(0);
}
```

Processi e thread
in Windows

Funzione CreateThread

Sintassi

```
HANDLE WINAPI CreateThread( _In_opt_   LPSECURITY_ATTRIBUTES   lpThreadAttributes,
                            _In_       SIZE_T   dwStackSize,
                            _In_       LPTHREAD_START_ROUTINE   lpStartAddress,
                            _In_opt_   LPVOID   lpParameter,
                            _In_       DWORD   dwCreationFlags,
                            _Out_opt_  LPDWORD   lpThreadId );
```

Parameters

***lpThreadAttributes* [in, optional]**

A pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes

***dwStackSize* [in]**

The initial size of the stack, in bytes. The system rounds this value to the nearest page. If this parameter is zero, the new thread uses the default size for the executable.

***lpStartAddress* [in]**

A pointer to the application-defined function to be executed by the thread. This pointer represents the starting address of the thread.

***lpParameter* [in, optional]**

A pointer to a variable to be passed to the thread.

***dwCreationFlags* [in]**

The flags that control the creation of the thread. 0= The thread runs immediately after creation.

***lpThreadId* [out, optional]**

A pointer to a variable that receives the thread identifier. If this parameter is **NULL**, the thread identifier is not returned.

Return value

If the function succeeds, the return value is a handle to the new thread. If the function fails, the return value is **NULL**.

TCP concorrente in ambiente Windows

product.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "product.h"


struct PRODUCT* products;       // puntatore (globale) alla tabella prodotti che sarà allocato dinamicamente
int number = 0;                 // numero di prodotti nella tabella allocata

int loadProducts(char filename[])
{
      FILE* file;
      int i, n;
      char line[1024];
      char* token;
      float price;

     // apertura file dei prodotti in formato CSV
      file = fopen(filename, "r");
      if (file == NULL)
        { system("pause");
          return -1;
        }
```

```
// informazioni relative a un prodotto
struct  PRODUCT {
                        char code[16];
                        char description[32];
                         float price;
                       };
// caricamento da file dei prodotti in vendita
// restituisce il numero di prodotti caricati, -1 in caso di errore
int loadProducts(char filename[]);

// ricerca delle informazioni di prodotto relative a un codice
// restituisce 1 in caso di successo, 0 altrimenti
int findProduct(char code[], struct PRODUCT* product);
```

. . .

TCP concorrente in ambiente Windows

. . . product.c

```
// ciclo di conteggio delle linee del file  (quanti prodotti ci sono)
n = 0;

do {
      fgets (line, sizeof(line)-1, file);
      n++;
     } while ( !feof(file) );

if (n == 0)
   {
    fclose(file);
    system("pause");
    return 0;
   }

// allocazione memoria per il vettore prodotti
products = malloc (n*sizeof(struct PRODUCT));
if (products  ==  NULL)     // se non ha potuto allocare dinamicamente la memoria chiude il file  e restituisce -1
    {
     fclose(file);
     return -1;
    }
```

char ***fgets**(char *s, int size, FILE *stream)

La funzione **fgets()** legge una linea dallo **stream** immagazzinandola nel buffer puntato da **s**.
Sono letti *al più* (**size**-1) caratteri, oppure fino al raggiungimento del carattere di new-line '\n' o di EOF.
Viene immagazzinato nel buffer anche l'eventuale carattere di new-line '\n' che venisse incontrato.
Dopo l'ultimo carattere letto, viene inserito nel buffer il carattere '\0'.

La funzione **feof()** testa il flag *end-of-file* per lo **stream**.
Restituisce 0 se il flag non è stato settato ed un valore diverso da zero altrimenti.

. . .

TCP concorrente in ambiente Windows

. . . product.c

```
// riposizionamento all'inizio del file dei prodotti
rewind(file);
i= 0;
// ciclo di lettura delle informazioni dei prodotti dal file
do {
        // lettura di una line dal file
        fgets(line, sizeof(line)-1, file);
        // estrazione delle informazioni del prodotto (separate da ",") e copia nella posizione corrente del vettore
        token = strtok(line, ",\r\n\0");     // ricerca  uno dei 4 simboli :   ,   \r   \n   \0
        if (token != NULL)
                {  strncpy(products[i].code, token, 15);
                   token = strtok(NULL, ",\r\n\0");
                   if (token != NULL)
                           {  strncpy(products[i].description, token, 31);
                              token = strtok(NULL, ",\r\n\0");
                              if (token != NULL)
                                      {  price = atof(token);
                                         if (price != 0)
                                                 { products[i].price = price;
                                                   i++;
                                                  }
                                      }
                           }
                }
      } while ( !feof (file));

fclose(file);
number = i;
return i;
}
```

double atof (const char *str)
converte la stringa **str** in un numero floating-point di tipo double

. . .

TCP concorrente in ambiente Windows

. . . product.c

```
int findProduct(char code[], struct PRODUCT* product)
{
   int i;

   // ciclo di ricerca sequenziale del codice di prodotto nel vettore
  for (i=0; i<number; i++)
        if (strcmp(code, products[i].code) == 0)
        {
            *product = products[i];
            return 1;
         }
   return 0;
}
```