funzione **fork**()

La funzione fork è usata per duplicare un processo.

```
#include <unistd.h>
pid_t fork (void);
```

• restituisce **-1 in caso di errore**. Se tutto va a buon fine **restituisce 0 nel processo figlio ed un valore maggiore di zero (il pid process identifier) nel processo padre**.

• Questa funzione viene chiamata nel processo (padre=parent), e restituisce il risultato in due diversi processi (padre e figlio).

• Se il figlio vuole conoscere il pid del padre userà la funzione getppid().

• **I descrittori di file e di socket aperti dal padre prima della fork sono condivisi col figlio, e possono perciò essere usati da entrambi per l'I/O.**

• Inoltre, per come funziona la funzione close(), **è possibile per uno dei processi (padre o figlio) chiudere una connessione aperta condivisa** (dai due processi) **senza con questo impedire all'altro processo di continuare ad utilizzare la connessione**.

• La fork viene usata per generare delle repliche del processo server, per gestire in parallelo le connessioni che via via vengono instaurate.

Server TCP Concorrenti (1)

• Un server banale in attesa su una porta TCP serializza le varie richieste di apertura di una connessione dei client permettendo la connessione ad un solo client per volta.
• Server TCP più evoluti invece, come i web server, una volta risvegliati dalla richiesta di una connessione da parte di un client, effettuano una fork() duplicando se stessi (il processo). Il processo figlio viene dedicato a servire la connessione appena instaurata, il processo padre attende nuove richieste di connessione sulla stessa porta.

A livello di interfaccia socket, questa situazione si ottiene così:
• Il server chiama la accept passando come argomento il **socket listening** (socket in ascolto), e subito dopo chiama la fork.
• il processo padre chiude il **connected socket**, e ripete la accept sul **listening socket**, in attesa della prossima richiesta di connessione.
• Invece il descrittore del **connected socket** (socket connesso) restituito dalla accept resta aperto per il figlio, e viene utilizzato da questo utilizzato per gestire l'I/O con la connessione. Quando infine il figlio termina il suo lavoro e chiude il connected socket con la close(), la connessione viene finalmente terminata con la sequenza di FIN.

```
pid_t pid;  int listenfd, connfd;
listenfd = socket (AF_INET, SOCK_STREAM, 0);
bind ( listenfd, (struct sockaddr*) &Local, sizeof(Local));
listen(listenfd, 10 );
for( ; ; )  {
   connfd = accept ( listenfd, (struct sockaddr*) &Cli, &len);
   pid = fork();
   if ( pid !=0)   close(connfd); /* processo padre */
   else  {   /* processo figlio */
      close(listenfd);
      usa_nuova_connessione_indicata_da_newsockfd(connfd);
      close(connfd);  exit(0);
   }
}
```

Server TCP Concorrenti (2)

Vediamo graficamente cosa capita a livello di TCP e di porte.
Entriamo un pò nei dettagli del programma appena visto, per quanto riguarda la scelta delle porte.
Consideriamo la situazione più complicata, quella di un'applicazione **server collocata su un host con più interfacce di rete, che vuole permettere le connessioni su una certa porta convenzionale (la 6001) da parte di client che accedono a una qualsiasi delle due interfacce del server**.

```
listenfd = socket (AF_INET, SOCK_STREAM, 0);
/* collega il socket ad un indirizzo IP locale e una porta TCP locale */
memset ( &Local, 0, sizeof(Local) );
Local.sin_family            =  AF_INET;

Local.sin_addr.s_addr    =  htonl(INADDR_ANY);  /* wildcard */
Local.sin_port              =  htons(6001);

bind ( listenfd, (struct sockaddr*) &Local, sizeof(Local));
/* accetta max 100 richieste simultanee di inizio conness., da adesso */
listen(listenfd, 100 );
/* accetta la prima conness. creando un nuovo socket per la conness. */
for( ; ; ){
        connfd = accept(listenfd, (struct sockaddr*) &Cli, &len);
        pid = fork();
        if ( pid !=0 )      /* processo padre */
                close ( connfd );
        else  {   /* processo figlio */
                close ( listenfd );   /* chiuso il listening socket */
                il figlio usa il connected socket ()
                close ( connfd );
                exit(0);
        }
}
```

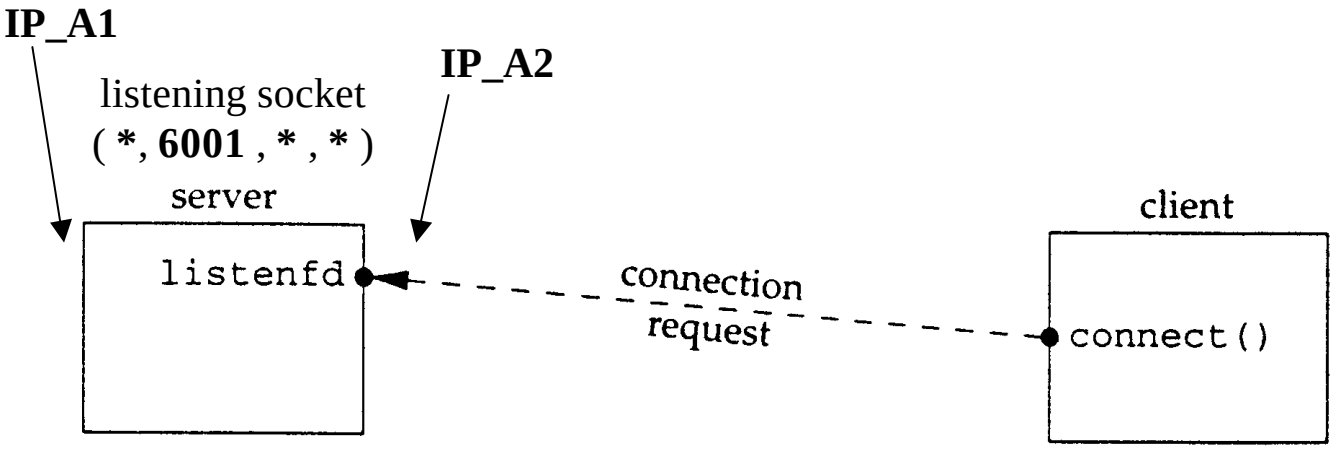
Server TCP Concorrenti (3)

La quaterna *(IP locale, Port Number locale , IP remoto, Port Number remoto)* che identifica univocamente una connessione TCP viene di solito chiamata **socket pair**.

SERVER
IP = IP_A1
IP = IP_A2

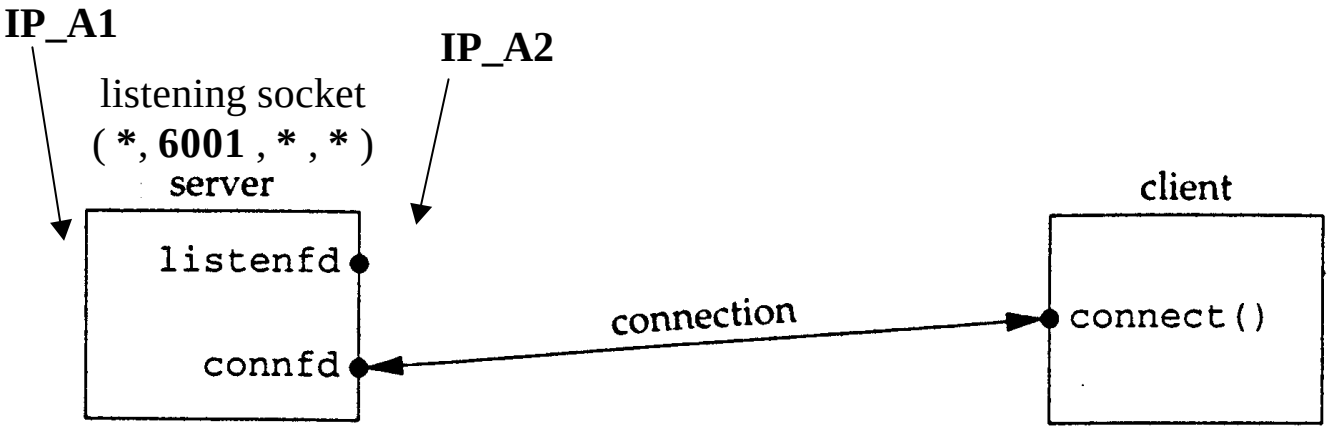
CLIENT
IP = IP_B
port = 2222

dopo la listen(), prima della accept()



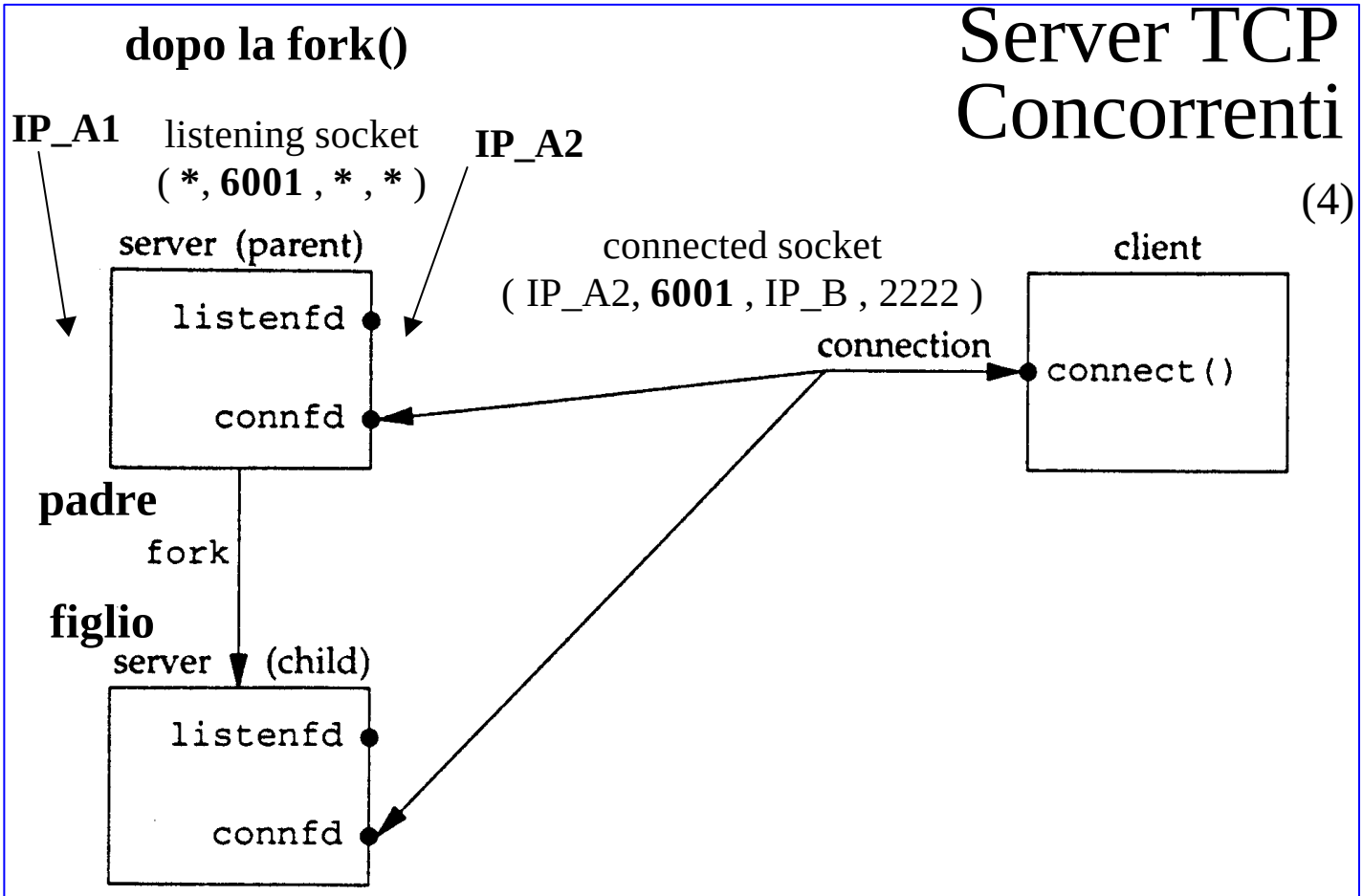
---

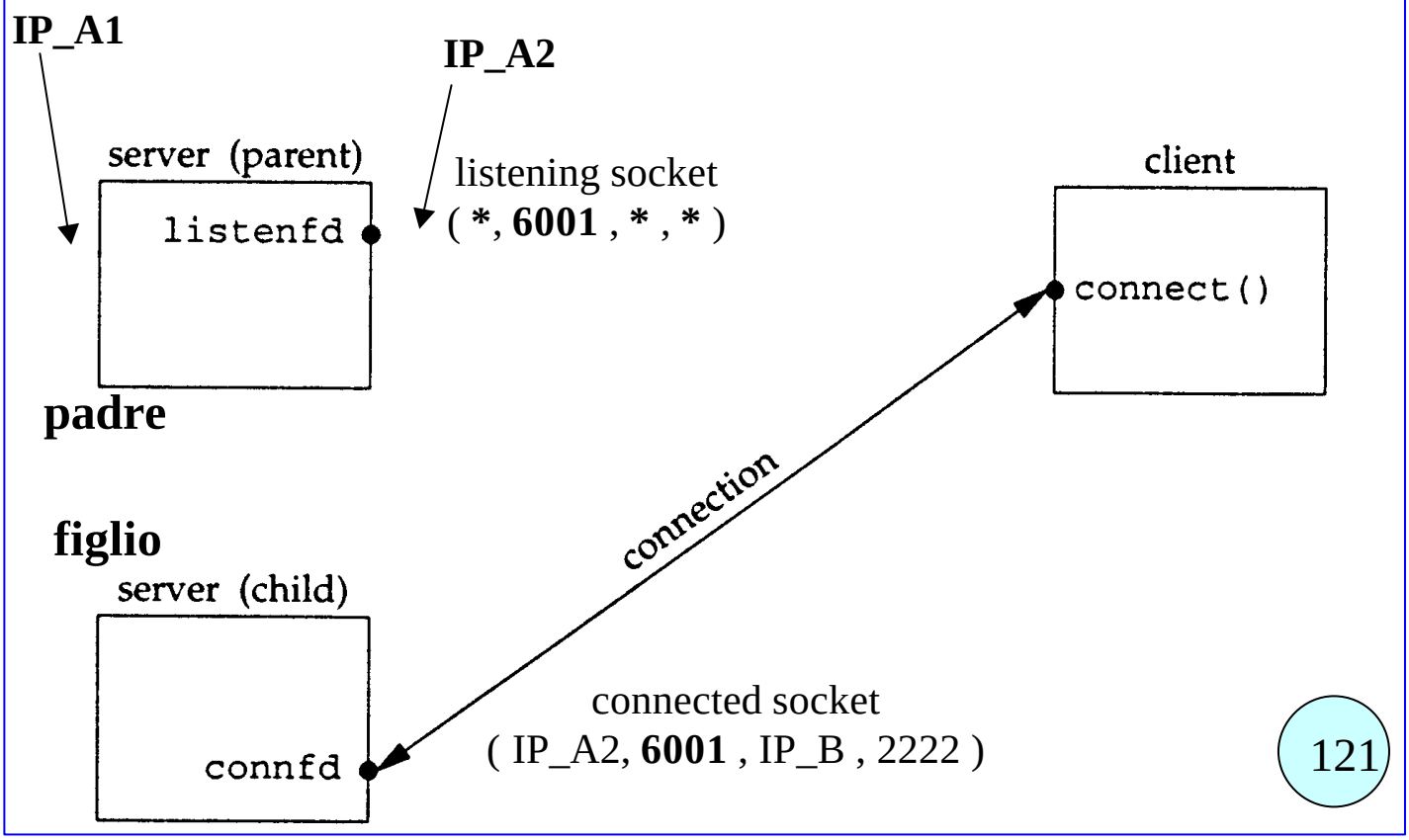
dopo la accept() , prima della fork()



connected socket
(IP_A2, **6001** , IP_B , 2222)

Server TCP Concorrenti (4)





I/O Multiplexing

Un’applicazione di rete può avere la necessità di accedere contemporaneamente a più tipi di input e di output, ad es. input di tipo stream dalla tastiera, input di tipo stream da rete eventualmente da più connessioni contemporaneamente, input di tipo datagram da rete anch’esso eventualmente da più socket contemporaneamente.
Esistono vari modelli di I/O disponibili in ambiente Unix:

- I/O Bloccante
- I/O Non Bloccante
- I/O tramite Multiplexing
- I/O guidato da signal
- I/O asincrono (per ora poco implementato)

Consideriamo per ora il **modello di I/O standard**, per cui quando viene effettuata una richiesta di I/O mediante una chiamata ad una primitiva di tipo read() o write(), la primitiva non restituisce il controllo al chiamante fino a che l’operazione di I/O non è stata effettuata, ovvero o la read() ha letto da un buffer del kernel dei dati, o la write() ha scritto dei dati dell’utente in un buffer del kernel.
Le funzioni di I/O finora analizzate sono state descritte nel loro funzionamento proprio secondo la modalità standard (bloccante).

• Il problema è che, quando l’applicazione effettua una read su un certo descrittore di file o di socket, se i dati non sono già presenti nella coda del kernel dedicata a quel descrittore, l’applicazione rimane bloccata fino a che i dati non sono disponibili, ed è impossibile leggere dati eventualmente già pronti sugli altri descrittori di file o socket. Analogamente per la write() se i buffer del kernel in cui si deve scrivere il dato è già occupato.

• Un problema analogo, caratteristico dei socket, si ha quando l’applicazione effettua una chiamata alla funzione accept(), che restituisce il controllo solo quando una richiesta di inizio connessione è disponibile (o meglio è già stata soddisfatta ed è nella coda delle connessioni stabilite).

I/O Multiplexing

Quello che serve è un modo di ordinare al kernek di avvertirci quando, in un insieme di canali di I/O, si verifica **una condizione di disponibilità all'I/O**, che può essere così definita:
1) o dei dati sono pronti alla lettura in una coda del kernel, e si può accedere mediante una read che restituirà immediatamente il controllo al chiamante con i dati letti,
2) o una coda di output del kernel si è svuotata ed è pronta ad accettare dati in scrittura mediante una write,
3) o si è verificato un errore in uno dei dispositivi di I/O e quindi una read() o write() restituirebbe il valore -1,
4) o quando un **socket listening** è disponibile a fornire immediatamente un **connected socket** in risposta ad una chiamata di tipo accept(), perchè ha ricevuto una richiesta di connessione da un client,

Posix.1g mette a disposizione una primitiva, detta **select(),** che:

1) permette di effettuare attesa contemporaneamente su più tipi di canali di I/O in modo da essere risvegliati quando uno di questi canali è disponibile all'I/O in lettura o scrittura o ha verificato un errore, o ancora nel caso dei socket quando sono disponibili i cosiddetti dati fuori banda (usati solo in casi particolarissimi perchè meno utili di quanto il nome farebbe presupporre),

2) e permette di fissare un limite all'attesa, in modo da essere risvegliati se non accade nulla allo scadere di un certo tempio limite. Quest'ultima possibilità può collassare in un'attesa di durata nulla, ovvero permette di non effettuare attesa alcuna, ma solo di controllare lo stato istantaneo dei vari canali e restituire subito il controllo al chiamante.

funzione **select**()

La funzione select permette di chiedere al kernel informazioni sullo stato di descrittori di tutti i tipi, riguardo a loro disponibilità in lettura scrittura o condizioni eccezionali, e di specificare quanto tempo al massimo aspettare.

```
#include <sys/select.h>
int select ( int maxfdp1, fd_set *readset, fd_set *writeset,
             fd_set *exceptset, const struct timeval *timeout);
```

La funzione restituisce -1 in caso di errore,
0 se il timeout fissato è scaduto,
altrimenti restituisce il numero di descrittori che hanno raggiunto la condizione di disponibilità loro richiesta.

L'ultimo argomento **timeout** dice al kernel quanto aspettare al massimo, ed è una struttura così fatta:

```
struct timeval {
        long    tv_sec;         /* secondi */
        long    tv_usec;        /* microsecondi */
}
```

con questa struttura noi possiamo specificare alla select:
attesa infinita: attesa fino a che una delle condizioni si è verificata. Si passa un puntatore timeout nullo.
attesa limitata: attesa il numero di secondi e microsecondi specificati nella struttura puntata dal puntatore timeout passato. In caso di timeout la select restituisce 0.
attesa nulla: ritorna subito al chiamante dopo avere fotografato la situazione dei descrittori. SI specifica settando a zero tv_sec e tv_usec.

NB: la timeval specifica microsecondi, ma i kernel non riescono a discriminare solitamente sotto i 10 msec.

funzione **select()** (2)

int **select** (int maxfdp1, fd_set *readset, fd_set *writeset,
fd_set *exceptset, const struct timeval *timeout);

I tre argomenti centrali di tipo fd_set*, **readset writeset exceptset** specificano i descrittori che si vuole controllare rispetivamente per verificare disponibilità alla lettura scrittura o eccezioni (out of band data only).

Il tipo fd_set (descriptor set) **è un array di interi, in cui ogni bit corrisponde ad un descrittore**. **Se il bit è settato il descrittore viene considerato appartenente al set, altrimenti non vi appartiene**.
Esistono delle macro per settare o resettare gli fd_set.

void FD_ZERO (fd_set *fdset);	clear di **tutti** i bit di fd_set
void FD_SET (int fd, fd_set *fdset);	**setta** il bit **fd** in fd_set
void FD_CLR (int fd, fd_set *fdset);	**clear** del bit **fd** in fd_set
int FD_ISSET (int fd, fd_set *fdset);	!=**0** se il bit fd è settato in fd_set **0** se il bit fd non è settato

Con queste macro posso settare pulire e controllare l'appartenenza o meno di un descrittore all'insieme. Es.:

fd_set readset;	dichiaro la variabile fd_set
FD_ZERO (&readset);	inizializzo, azzero tutto, insieme vuoto
FD_SET (1, &readset);	1 appartiene all'insieme
FD_SET (4, &readset);	4 "
FD_SET (7, &readset);	

FD_ISSET (4, &readset) restituisce != 0
FD_ISSET (3, &readset) restituisce 0

Ricordarsi di inizializzare il set (FD_ZERO) altrimenti risultati impredicibili.

funzione **select()** (3)

int **select** (int maxfdp1, fd_set *readset, fd_set *writeset,
fd_set *exceptset, const struct timeval *timeout);

Il primo argomento **maxfdp1**, specifica quali descrittori controllare, nel senso che deve avere il valore più alto tra i descrittori settati + 1.
Es.: se i descrittori settati sono **1, 4 , 7** maxfdp1 deve essere **8** = 7+1

I 3 descrittori di set passati per argomento contengono quindi i descrittori da controllare, e vengono passati per puntatore perchè la select li modifica scrivendoci sopra il risultato.
Quando la select termina si controllano ciascuno dei 3 fd_set, chiedendo tramite la macro FD_ISSET() quali descrittori sono settati.
Se un descrittore (es. 4) non è settato (es. FD_ISSET(4, &readset)== 0) significa che non è pronto.
Se invece il descrittore è settato (es. FD_ISSET(4, &readset) != 0) significa che è pronto.

- **Il valore restituito dalla select dice quanti descrittori sono stati settati.**
- **se la select restituisce 0 significa che è scaduto il timeout.**
- **se la select restituisce -1 c'e' stato un errore o è avvenuta una signal.**

N.B. esiste una define che specifica la costante FD_SETSIZE ovvero il numero di descrittori che può contenere la struttura fd_set.

Vediamo ora un esempio di uso della select, con cui implementiamo un web server, che lavora in parallelo senza dover fare delle fork().

esempio d'uso della select() server che non utilizza la fork() (1)

Prima parte del server, inizializzazione:

```
typedef SA struct sockaddr;

int main(int argc, char **argv)
{
int                     i, maxi, maxfd, listenfd, connfd, sockfd;
int                     nready, client[FD_SETSIZE];
ssize_t                 n;
fd_set                  rset, allset;
char                    line[MAXLINE];
socklen_t               clilen;
struct sockaddr_in      cliaddr, servaddr;

listenfd = socket(AF_INET, SOCK_STREAM, 0);

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family         = AF_INET;
servaddr.sin_addr.s_addr  = htonl(INADDR_ANY);
servaddr.sin_port           = htons(SERV_PORT);

bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

listen(listenfd, LISTENQ);

maxfd = listenfd;                       /* initialize */
maxi = -1;                              /* index into client[] array */
for (i = 0; i < FD_SETSIZE; i++)
            client[i] = -1;             /* -1 indicates available entry */
FD_ZERO(&allset);
FD_SET(listenfd, &allset);
```

server che non utilizza la fork() (1)

```
for ( ; ; ) {
   rset = allset;                          /* structure assignment */
   nready = select(maxfd+1, &rset, NULL, NULL, NULL);

   if ( FD_ISSET( listenfd, &rset) )  {          /* new client connection */
      clilen = sizeof(cliaddr);
      connfd = accept( listenfd, (SA *) &cliaddr, &clilen);

      for (i = 0; i < FD_SETSIZE; i++)
         if (client[i] < 0) {
            client[i] = connfd;         /* save descriptor */
            break;
         }
      if (i == FD_SETSIZE)  err_quit("too many clients");
      FD_SET(connfd, &allset);   /* add new descriptor to set */
      if (connfd > maxfd)      maxfd = connfd;                 /* for select */
      if (i > maxi)   maxi = i;         /* max index in client[] array */
      if (--nready <= 0)
         continue;         /* no more readable descriptors */
   }
   for (i = 0; i <= maxi; i++) {      /* check all clients for data */
      if ( (sockfd = client[i]) < 0)
          continue;
      if (FD_ISSET(sockfd, &rset)) {
         if ( (n = Readline(sockfd, line, MAXLINE)) == 0) {
             /*connection closed by client */
             close(sockfd);
             FD_CLR(sockfd, &allset);
             client[i] = -1;
         } else
         writen(sockfd, line, n);
         if (--nready <= 0)
              break;    /* no more readable descriptors */
      }
  }
}
}
```