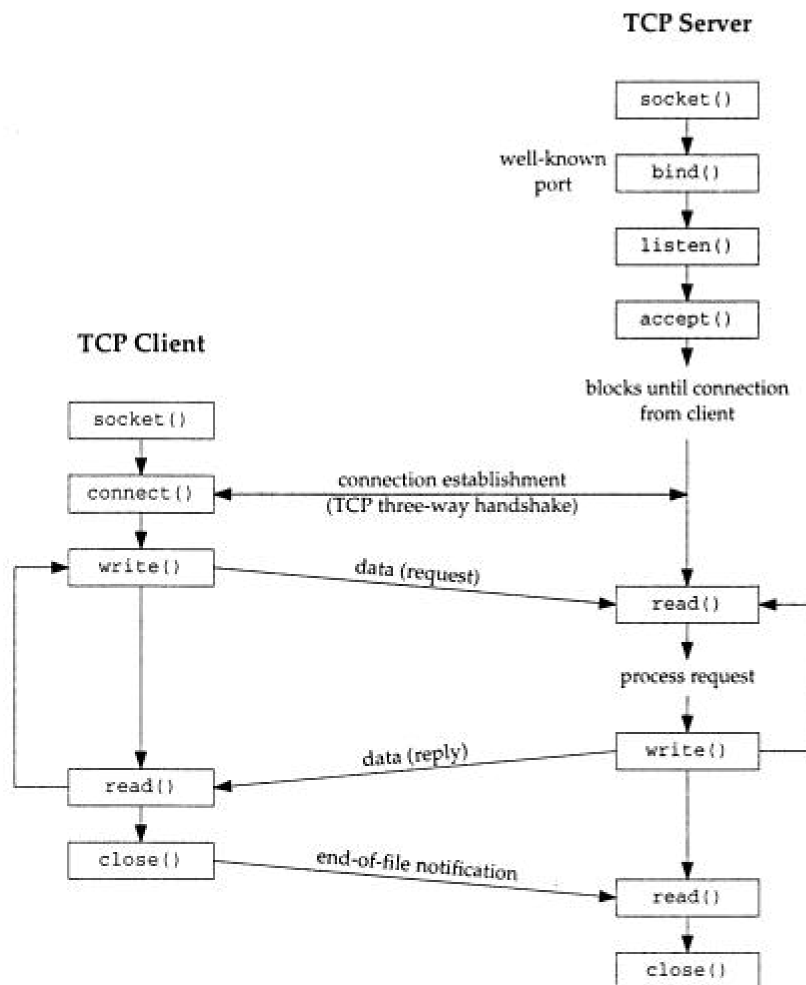
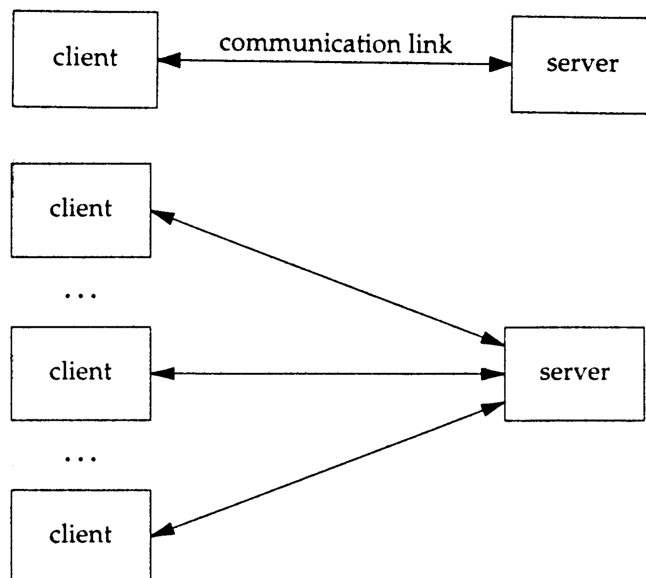
Socket per TCP: Fondamenti



Network Applications

Molte applicazioni di rete sono formate da due programmi distinti (che lavorano su due diversi host) uno detto server ed uno detto client.



Il server si mette in attesa di una richiesta da servire, il client effettua tale richiesta.

Tipicamente il client comunica con un solo server, mentre un server usualmente comunica con più client contemporaneamente (su connessioni diverse nel caso tcp).

Inoltre spesso client e server sono processi utente, mentre i protocolli della suite TCP/IP fanno solitamente parte del sistema operativo.

Nel seguito faremo riferimento al termine IP nel senso di IPv4.

Unix Standards

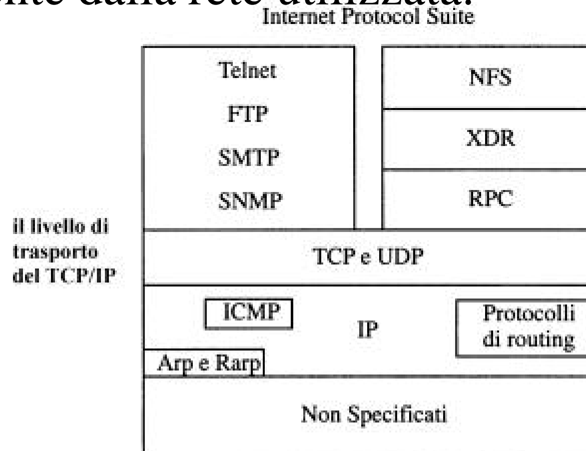
Posix = Portable Operating System Interface è una famiglia di standard (vedi http://www.pasc.org/standing/sd11.html) sviluppata da IEEE e adottata da ISO. Posix comprende **IEEE Std 1003.1 (1996)** (una raccolta di alcune specifiche precedenti) che contiene al suo interno una parte detta “Part1: System Application Program Interface (API)” che specifica l’interfaccia C per le chiamate di sistema del kernel Unix, relative a processi (fork, exec, signal, timer, user ID, gruppi), files e directory (I/O function), I/O da terminale, password, le estensioni per il realtime, execution scheduling, semaphores, shared memory, clock, message queues. In particolare comprende **IEEE Std 1003.1g: Protocol Independent Interface (PII)** che è lo standard per l’interfaccia di programmazione delle reti, e definisce due standard chiamati **DNI** (**Detailed Network Interfaces**):

1) **DNI/Socket** basato sulle API socket del 4.4BSD, di cui ci occuperemo

2) DNI/XTI, basato sulle specifiche XPG4 del consorzio X/Open

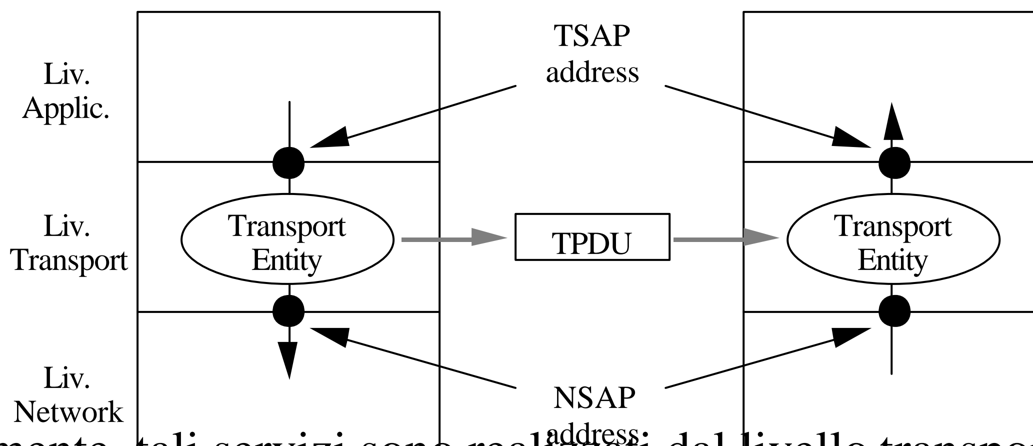
Il livello di Trasporto del TCP/IP

Il compito del livello transport (livello 4) è di fornire un trasporto efficace dall'host di origine a quello di destinazione, indipendentemente dalla rete utilizzata.



Questo è il livello in cui si gestisce per la prima volta (dal basso verso l'alto) una conversazione diretta, cioè senza intermediari, fra una transport entity su un host e la sua peer entity su un altro host.

.



Naturalmente, tali servizi sono realizzati dal livello transport per mezzo dei servizi ad esso offerti dal livello network.

Così come ci sono due tipi di servizi di livello network, ce ne sono due anche a livello transport:

• servizi affidabili orientati alla connessione, detti di tipo stream, offerti dal **TCP** (Transmission Control Protocol);

• servizi senza connessione detti di tipo datagram offerti dall' **UDP** (User Datagram Protocol).

Indirizzi a livello di Trasporto per il TCP/IP

Quando si vuole trasferire una o più **TPDU** (Transport Protocol Data Unit) da una sorgente ad una destinazione di livello 4, occorre specificare mittente e destinatario di livello 4. Il protocollo di livello 4 deve quindi decidere come deve essere fatto l'indirizzo di livello transport, detto ***TSAP address*** (***Transport Service Access Point address***).

Poichè l'indirizzo di livello 4 serve a trasferire informazioni tra applicazioni che lavorano su hosts diversi, deve poter individuare un host e una entità contenuta nell'host. Per questo motivo i protocolli di livello 4 tipicamente definiscono come indirizzo di livello 4 una coppia formata da un indirizzo di livello network che identifica l'host, e da un'altra informazione che identifica un punto di accesso in quell'host **(NSAP address, informazione supplementare)**.

Ad esempio, in TCP/IP un TSAP address (ossia un indirizzo TCP o UDP) ha la forma:

(IP address : port number)

Port number è un intero a 16 bit, che identifica un servizio o un punto di accesso e/o smistamento di livello 4.

Ad es. la coppia (137.204.72.49 : 23) indica la porta 23 dell'host poseidon.csr.unibo.it, cioè l'entry point per il demone telnet, cioè il punto di accesso all'applicazione che permette ad un utente di collegarsi mediante telnet all'host poseidon.

• Anche se l'indirizzo di livello trasporto è formato da questa coppia, **il TCP/IP**, per ridurre l'overhead causato dagli header dei vari livelli, **nella trasmissione effettua una violazione della stratificazione tra i livelli 4 e 3** (Trasporto e Network). Infatti, come vedremo TCP e UDP contengono nei loro header solo i numeri delle porte e riutilizzano gli indirizzi IP di mittente e destinaz. contenuti nel pacchetto IP.

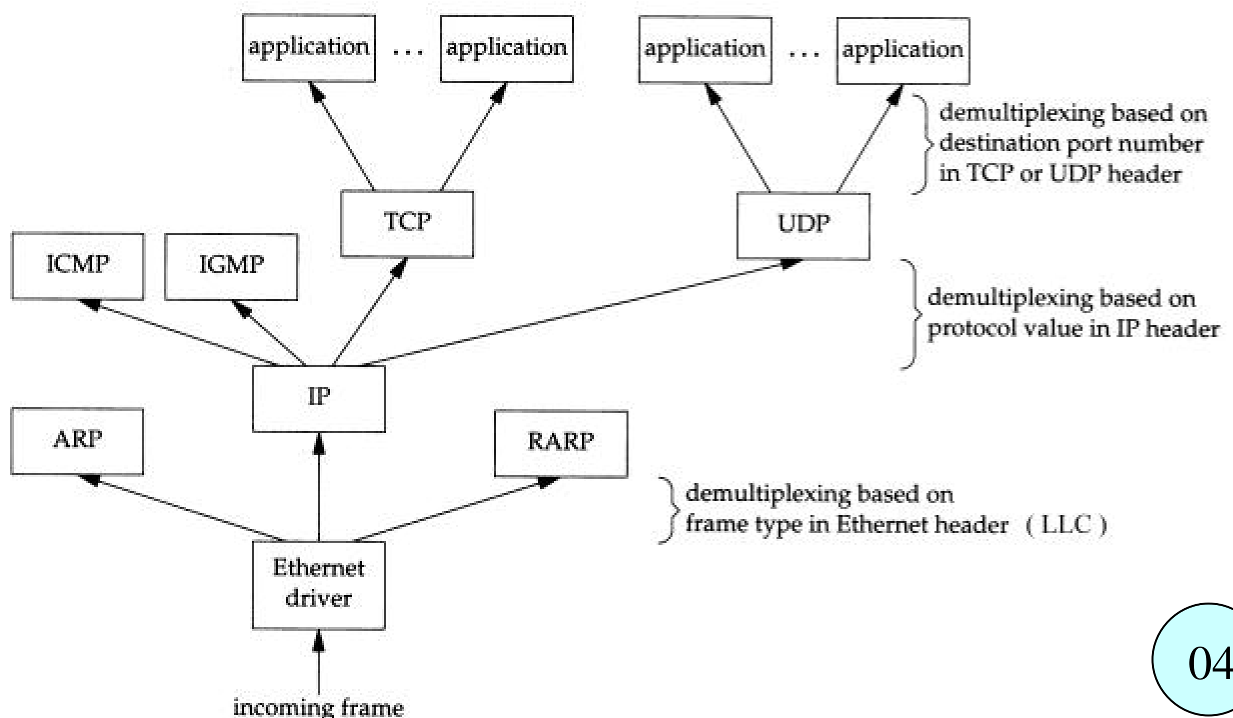
Multiplexing a livello Trasporto

• Gli **identificatori di porta** (port number) permettono di effettuare la **demultiplazione** dei pacchetti di livello 4, ovvero di discriminare l'applicazione destinazione dei pacchetti in funzione del port number contenuto nell'header del pacchetto di livello transport, sia esso di tipo TCP che UDP.

• E' ovvio che mittente e destinatario devono essere d'accordo sul valore della porta del destinatario per poter effettuare la trasmissione.

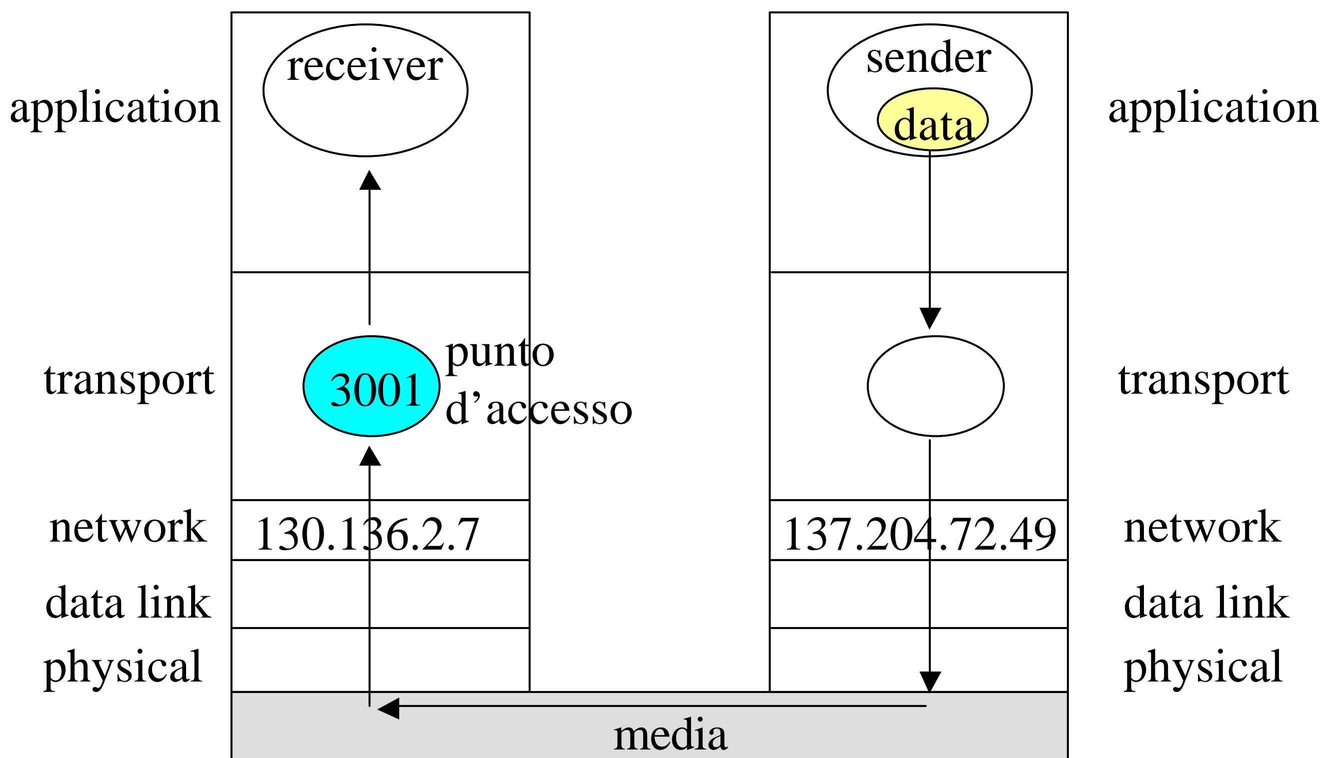
• Il mittente scrive questo numero di porta come indirizzo del destinatario, ed il destinatario si deve mettere in attesa dei pacchetti che giungono all'host del destinatario e che posseggono come identificatore proprio quel port number.

• Alcune primitive fornite dall'interfaccia socket permettono di specificare il numero di porta di cui interessa ricevere i pacchetti (stream=flussi di dati nel caso TCP). E' quindi il sistema operativo che si fa carico di effettuare le operazioni di demultiplexing dei pacchetti ricevuti dal livello network.



esempio di trasmissione di datagram UDP

Senza per ora entrare nei dettagli riguardanti i socket, vediamo un semplice esempio di programma che sfrutta i socket per trasmettere un datagram UDP contenente una stringa di testo “pippo” da un host **sender 137.204.72.49** ad un host **receiver 130.136.2.7** .
Il punto di accesso stabilito dal programmatore è nel receiver, nella porta UDP caratterizzata dal numero 3001.
Il receiver si mette in attesa sulla porta **3001** fino a che il sender invia un datagram all’host receiver su quella porta, e stampa il contenuto del datagram ricevuto.
Quindi sender e receiver devono essersi messi d’accordo sulla porta da usare, ed il sender deve conoscere l’indirizzo IP del receiver.



Il codice completo (con la gestione degli errori) dei due programmi che realizzano l’esempio qui mostrato è disponibile allo indirizzo http://www.cs.unibo.it/~ghini/didattica/sistemi3/UDP1/UDP1.html

esempio: receiver di datagram UDP

```
/* eseguito sull'host 130.136.2.7 */
#define SIZEBUF 10000
void main(void)    {
struct sockaddr_in Local, From;   short int local_port_number=3001;
char string_remote_ip_address[100];  short int remote_port_number;
int socketfd, OptVal, msglen, Fromlen;  char msg[SIZEBUF];

/* prende un socket per datagram UDP */
socketfd = socket (AF_INET, SOCK_DGRAM, 0);
/* impedisce l'errore di tipo EADDRINUSE nella bind() */
OptVal = 1;
setsockopt (socketfd, SOL_SOCKET, SO_REUSEADDR,
                (char *)&OptVal, sizeof(OptVal) );
/* assegna l'indirizzo IP locale e una porta UDP locale al socket */
Local.sin_family            =  AF_INET;
Local.sin_addr.s_addr       =  htonl(INADDR_ANY);
Local.sin_port              =  htons(local_port_number);
bind ( socketfd, (struct sockaddr*) &Local, sizeof(Local));

/* wait for datagram */
Fromlen=sizeof(struct sockaddr);
msglen = recvfrom ( socketfd, msg, (int)SIZEBUF, 0,
                            (struct sockaddr*)&From, &Fromlen);
sprintf((char*)string_remote_ip_address,"%s",inet_ntoa(From.sin_addr);

remote_port_number = ntohs(From.sin_port);
printf("ricevuto msg: \"%s\" len %d, from host %s, port %d\n",
       msg, msglen, string_remote_ip_address, remote_port_number);
}
```

esempio: sender di datagram UDP

```
/* eseguito sull'host 137.204.72.49 */
int main(void)   {
struct sockaddr_in Local, To;   char msg[]="pippo";
char string_remote_ip_address[]="130.136.2.7";
short int remote_port_number = 3001;
int socketfd, OptVal, addr_size;

/* prende un socket per datagram UDP */
socketfd = socket(AF_INET, SOCK_DGRAM, 0);
/* impedisce l'errore di tipo EADDRINUSE nella bind() */
OptVal = 1;
setsockopt(socketfd, SOL_SOCKET, SO_REUSEADDR,
                (char *)&OptVal, sizeof(OptVal));
/* assegna l'indirizzo IP locale e una porta UDP locale al socket */
Local.sin_family          =  AF_INET;
/* il socket verra' legato all'indirizzo IP dell'interfaccia che verrà
   usata per inoltrare il datagram IP, e ad una porta a scelta del s.o.  */
Local.sin_addr.s_addr  =  htonl(INADDR_ANY);
Local.sin_port             =  htons(0); /* il s.o. decide la porta locale */
bind( socketfd, (struct sockaddr*) &Local, sizeof(Local));
/* assegna la destinazione */
To.sin_family             =  AF_INET;
To.sin_addr.s_addr  =  inet_addr(string_remote_ip_address);
To.sin_port                =  htons(remote_port_number);
addr_size = sizeof(struct sockaddr_in);
/* send to the address */
sendto(socketfd, msg, strlen(msg) , 0,
          (struct sockaddr*)&To, addr_size);
}
```

Socket Address Structures (1)

Cominciamo la descrizione delle Socket API (Application program Interface) dalla descrizione delle **strutture usate per trasferire indirizzi** dall'applicazione al kernel (nelle funzioni bind, connect, sendto) e dal kernel alle applicazioni (nelle funzioni accept, recvfrom, getsockname e getpeername).

• I dati definiti per Posix.1g sono quelli della seguente tabella:

int8_t	signed 8-bit integer <sys/types.h>	
uint8_t	unsigned 8-bit integer	<sys/types.h >
int16_t	signed 16-bit integer	<sys/types.h >
uint16_t	unsigned 16-bit integer	<sys/types.h>
int32_t	signed 32-bit integer	<sys/types.h>
uint32_t	unsigned 32-bit integer	<sys/types.h>
sa_family_t	famiglia di indirizzi socket *AF_INET per IPv4, AF_INET6 per IPv6,* *AF_LOCAL per indir. locali unix (per pipe ecc..)*	<sys/socket.h>
socklen_t	lunghezza della struttura che contiene l'indirizzo, di solito è un uint32_t	<sys/socket.h>
in_addr_t	indirizzo IPv4, = uint32	<netinet/in.h>
in_port_t	porta TCP o UDP, = uint16	<netinet/in.h>

• Poichè i socket devono fornire un'interfaccia per diverse famiglie di protocolli (IPv4, IPv6 e Unix), e poichè tali strutture vengono passate per puntatore, le funzioni di libreria presentano un argomento che è il **puntatore alla generica struttura** (**struct sockaddr***), ma essendo diversa la struttura passata a seconda della famiglia di indirizzi usata, l'argomento passato deve essere convertito mediante il cast alla struttura (struct sockaddr*), ad es:

```
struct sockaddr_in server;  /* IPv4 socket address structure */
memset ( &server, 0, sizeof(server) ); /* azzero tutta la struttura */
... riempimento dei dati della struttura server ...
bind ( socketfd, (struct sockaddr *)&server, sizeof(server) );
```

Socket Address Structures (2)

La generica struttura dell'indirizzo è dunque cosi definita:

```
struct sockaddr {
        uint8_t          sa_len;
        sa_family_t      sa_family;
        char             sa_data[14];
        };
```

La famiglia di indirizzi Ipv4 (sa_family=AF_INET) usa la struttura:

```
struct sockaddr_in {
   uint8_t            sin_len;       /* lunghezza struttura */
   sa_family_t        sin_family;  /* = AF_INET */
   in_port_t          sin_port;    /*16-bit TCP UDP port, network byte ordered */
   struct in_addr  sin_addr;    /* 32-bit IPv4 address, network byte ordered */
   char               sin_zero[8];    /* unused */
};
```

con

```
struct in_addr {                    /* e' una struttura per ragioni storiche */
        in_addr_t    s_addr ;    /* 32-bit IPv4 address network byte ordered */
        };
```

• sa_len e sa_family si sovrappongono perfettamente a sin_len e sin_family rispettivamente, permettendo di leggere la costante di tipo sa_family_t e di capire che tipo di struttura si sta utilizzando.
• il campo sin_len non è richiesto espressamente da Posix.1g, e anche quando è presente non è necessario settarlo, se non per applicazioni di routing, in quanto le principali funzioni in cui si passano indirizzi prevedono già un argomento in cui si passa (o riceve) la lunghezza della struttura indirizzo.
• Il campo sin_zero non è usato, ma va sempre settato tutto a zero prima di passare una struttura che lo contiene. Di più, **per convenzione, bisogna sempre settare TUTTA la struttura indirizzo tutta a zero prima di riempire i vari campi, usando la funzione memset().**
• **memset (&server, 0, sizeof(server));**

Socket Address Structure (3)

Confrontiamo alcune delle strutture usate per gli indirizzi:

IPv4

sockaddr_in{}

length	AF_INET
16-bit port#	
32-bit IPv4 address	
(unused)	

fixed length (16 bytes)

IPv6

sockaddr_in6{}

length	AF_INET6
16-bit port#	
32-bit flow label	
128-bit IPv6 address	

fixed length (24 bytes)

Unix

sockaddr_un{}

length	AF_LOCAL
pathname (up to 104 bytes)	

variable length

Datalink

sockaddr_dl{}

length	AF_LINK
interface index	
type	name len
addr len	sel len
interface name and link-layer address	

variable length

Funzioni di Ordinamento dei Byte

Poichè alcuni campi delle strutture di indirizzo (i numeri di porta o gli indirizzi IPv4 ad esempio) devono essere memorizzati secondo l'ordine per i bytes stabilito per la rete (network byte order), prima di assegnare alla struttura un valore di porta (16-bit) o un indirizzo IPv4 (32-bit) è necessario convertirlo dall'ordine dei byte per l'host all'ordine per la rete, utilizzando delle funzioni di conversione, i cui prototipi sono definiti nell'include <netinet/in.h>:

uint16_t **htons** (uint16_t host16bitvalue); /* Host TO Network Short */
uint32_t **htonl** (uint32_t host32bitvalue); /* Host TO Network Long */

Viceversa, per convertire il valore di una porta o di un indirizzo IPv4, preso da una struttura di indirizzo, in un valore intero secondo l'ordinamento dell'host si devono utilizzare le funzioni:

uint16_t **ntohs** (uint16_t net16bitvalue); /* Network TO Host Short */
uint32_t **ntohl** (uint32_t net32bitvalue); /* Network TO Host Long */

Se l'ordinamento dell'host è corrispondente all'ordinamento di rete, queste funzioni sono implementate con delle macro nulle, cioè non modificano il dato.

Funzioni di Manipolazione dei Byte

Vediamo solo le funzioni portabili ovunque perche sono ANSI C.

void ***memset*** (void *dest, int c, size_t n_bytes);
setta al valore c un numero len di byte a partire da dest

void *memcpy (void *dest, const void *src, size_t n_bytes);
copia n_bytes byte da src a dest, problemi se c'e' sovrapposizione, nel caso usare memmove. Resituisce dest.

void ***memcmp*** (const void ptr1, const void *ptr2, size_t n_bytes);
confronta due vettori di n_bytes ciascuno, restituisce 0 se sono uguali, diverso da zero se diversi.

Funzioni di Conversione di Indirizzi IP dalla forma dotted-decimal ASCII string alla forma 32-bit network byte ordered

Queste funzioni sono definite in <arpa/inet.h>

Le funzioni **inet_aton** e **inet_addr** convertono gli indirizzi IP da una forma di stringa di caratteri ASCII decimali separati da punti del tipo “255.255.255.255”, nella forma di interi a 32-bit ordinati secondo l’ordinamento di rete.

int **inet_aton** (const char *str, struct in_addr *addrptr);

scrive nella locazione puntata da addrptr il valore a 32-bit, nell’ordine di rete, ottenuto dalla conversione della stringa zero-terminata puntata da str. Restituisce zero in caso di errore, 1 se tutto va bene.

in_addr_t inet_addr (const char *str);   **NON VA USATA**

restituisce il valore a 32-bit, nell’ordine di rete, ottenuto dalla conversione della stringa zero-terminata puntata da str.
In caso di errori restituisce INADDR_NONE, e questo è un casino, perchè INADDR_NONE è un intero a 32 bit di tutti 1, che sarebbe ottenuto come risultato della chiamata di inet_addr passandogli la stringa “255.255.255.255” che è l’indirizzo valido di broadcast.
Per evitare confusione non deve essere usata.

Infine c’e’ una funzione che effettua la conversione inversa, da interi a 32-bit network ordered verso stringhe ASCII decimali separate da punti.

char ***inet_ntoa** (struct in_addr addr);

scrive in una locazione di memoria statica (di cui restituisce un puntatore) la stringa ASCII null-terminata di caratteri decimali separati da punti corrispondeni all’indirizzo IP a 32-bit, nell’ordine di rete, contenuto nella struttura addr (che stranamente non è un puntatore). Occhio, questa funzione non è rientrante, perchè memorizza il risultato in una locazione statica.

I/O su Socket TCP (1)

I socket TCP, una volta che la connessione TCP sia stata instaurata, sono accedibili come se fossero dei file, mediante un descrittore di file (un intero) ottenuto tramite una socket() o una accept() o una connect(). Con questo descrittore è possibile effettuare letture tramite la funzione read, che restituisce i byte letti dal flusso in entrata, e scritture tramite la funzione write, che spedisce i byte costituendo il flusso in uscita.

ssize_t **read** (int fd, void *buf, size_t count);

> cerca di leggere count byte dal file descriptor fd, scrivendoli nel buffer puntato da buf. Se count è zero restituisce zero. Se count è maggiore di zero viene effettuata la lettura e viene restituito il numero di byte letti. Se viene restituito zero significa end-of-file (fine stream). Se viene restituito -1 è accaduto un errore e viene settato la variabile globale errno definita in <errno.h>.

La funzione read presenta una particolarità quando è applicata ad un socket. Può accadere che la read() restituisca meno byte di quanti richiesti, anche se lo stream è ancora aperto. Ciò accade se il buffer a disposizione del socket nel kernel è stato esaurito. Sarà necessario ripetere la read (richiedendo il numero dei byte mancanti) fino ad ottenerli tutti. Oss.: ssize_t è definito in <unistd.h> ed è un long.

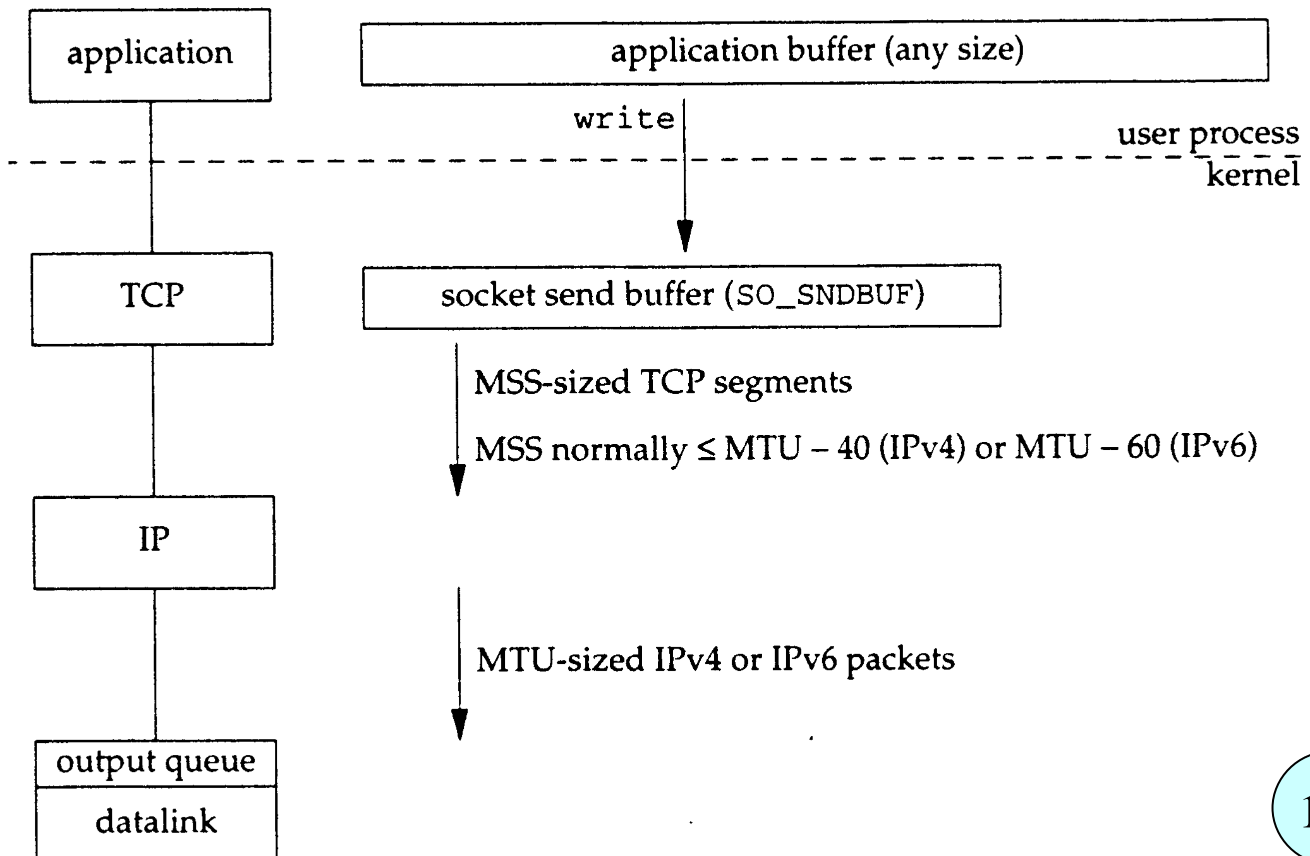
ssize_t **write** (int fd, const void *buf, size_t count);

> cerca di scrivere fino a count byte sul file descriptor fd, leggendoli dal buffer puntato da buf. Se count è zero restituisce zero. Se count è maggiore di zero viene effettuata la scrittura e viene restituito il numero di byte scritti. Se viene restituito -1 è accaduto un errore e viene settato errno.

Analogamente alla read() anche la write presenta una particolarità quando è applicata ad un socket. Può accadere che la write() scriva meno byte di quanto richiesto, anche se lo stream è ancora aperto. Ciò accade se il buffer a disposizione del socket nel kernel è stato esaurito. Sarà necessario ripetere la write (con i soli byte mancanti) fino a scriverli tutti.

I/O su Socket TCP: (2)
TCP Output

Ogni socket TCP possiede un buffer per l’output (send buffer) in cui vengono collocati temporaneamente i dati che dovranno essere trasmessi mediante la connessione instaurata. La dimensione di questo buffer può essere configurata mediante un’opzione SO_SNDBUF.
Quando un’applicazione chiama write() per n bite sul socket TCP, il kernel cerca di copiare n byte dal buffer dell’appl. al buffer del socket.
Se il buffer del socket e’ più piccolo di n byte, oppure è già parzialmente occupato da dati non ancora trasmessi e non c’è spazio sufficiente, verranno copiati solo nc<n byte, e verrà restituito dalla write il numero nc di byte copiati.
Se il socket ha le impostazioni di default, cioè è di tipo bloccante, la fne della routine write ci dice che sono stati scritti sul buffer del socket quegli nc byte, e possiamo quindi riutilizzare le prime nc posizioni del buffer dell’applicazione. Ciò non significa affatto che già i dati siano stati trasmessi all’altro end-system.



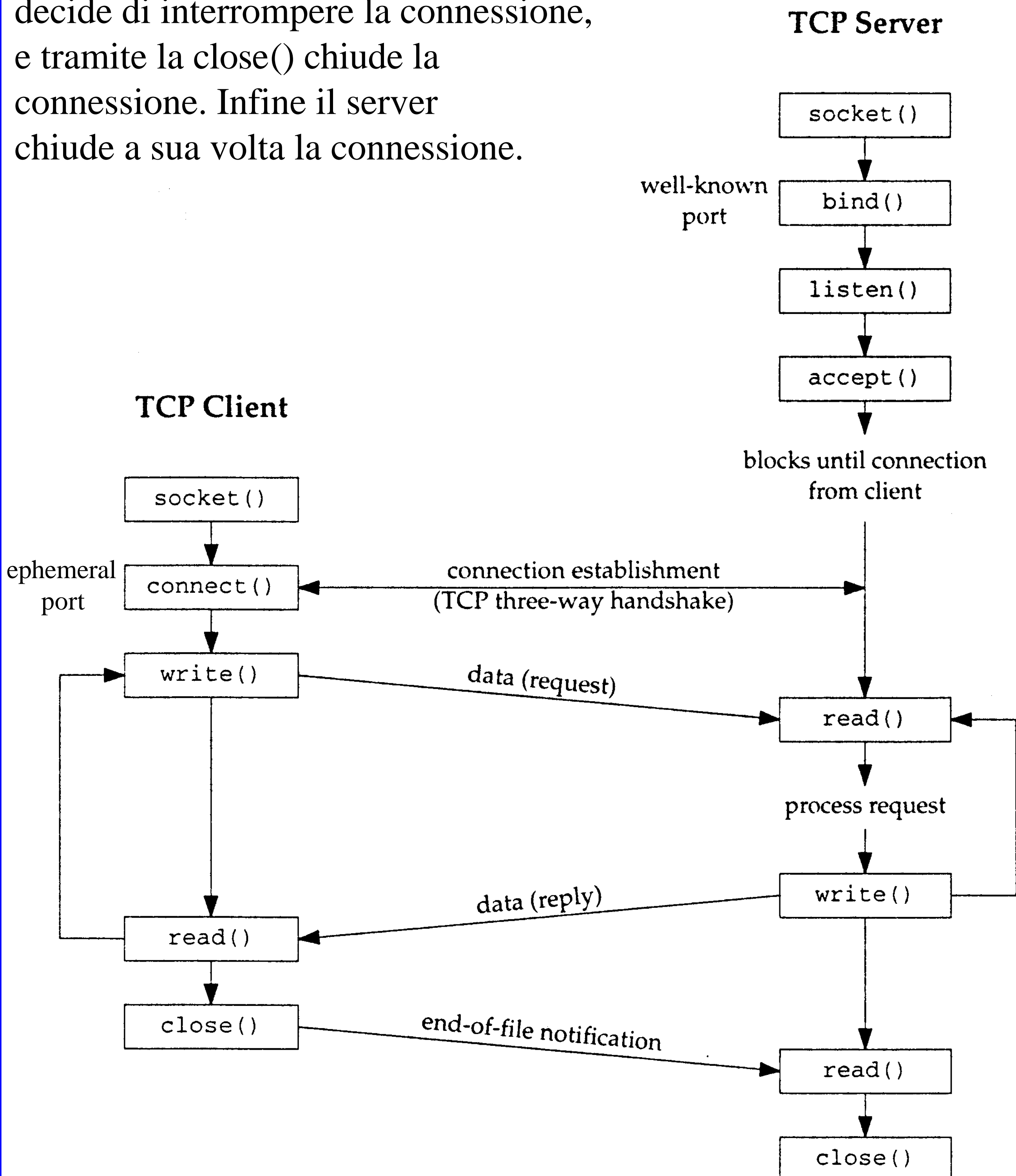
I/O su Socket TCP (3)

Per semplificarsi la vita ed usare delle funzioni per l'I/O con i socket che si comportano esattamente come le read() e write() su file, si può scrivere due proprie funzioni read**n**() e write**n**() che effettuano un loop di letture/scritture fino a leggere/scrivere tutti gli n byte come richiesto, o incontrare l'end-of-file o riscontrare un errore.

```
ssize_t  readn (int fd, void *buf, size_t n)
{   size_t  nleft;     ssize_t  nread;  char   *ptr;
    ptr = buf;  nleft = n;
    while (nleft > 0) {
        if ( (nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)   nread = 0;    /* and call read() again */
            else                  return(-1);
        }
        else if (nread == 0)
              break;    /* EOF, esce */
        nleft -= nread;   ptr   += nread;
    }
    return(n - nleft);              /* return >= 0 */
}
ssize_t  writen (int fd, const void *buf, size_t n)
{   size_t  nleft;     ssize_t  nread;  char   *ptr;
    ptr = buf;  nleft = n;
    while (nleft > 0) {
        if ( (nwritten = write(fd, ptr, nleft)) <= 0) {
            if (errno == EINTR)   nwritten = 0;   /* and call write() again*/
            else                  return(-1);      /* error */
        }
        nleft -= nwritten;
        ptr   += nwritten;
    }
    return(n);
}
```

Interazioni tra Client e Server TCP

Per primo viene fatto partire il server, poi viene fatto partire il client che chiede la connessione al server e la connessione viene instaurata.
Nell'esempio (ma non è obbligatorio) il client spedisce una richiesta al server, questo risponde trasmettendo alcuni dati. Questa trasmissione bidirezionale continua fino a che uno dei due (il client nell'esempio) decide di interrompere la connessione, e tramite la close() chiude la connessione. Infine il server chiude a sua volta la connessione.



funzione **socket**()

La prima azione per fare dell'I/O da rete è la chiamata alla funziona socket() specificando il tipo di protocollo di comunicazione da utilizzare (TCP con IPv4, UDP con IPv6, Unix domain stream protocol per usare le pipe).

```
#include <sys/socket.h>
int  socket (int family, int type, int protocol);
```

restituisce un descrittore di socket maggiore o uguale a zero, oppure -1 in caso di errore, e setta errno.

L'argomento family specifica la famiglia di protocolli da utilizzare.

family	descrizione
AF_INET	IPv4 protocol
AF_INET6	IPv6 protocol
AF_LOCAL	Unix domain protocols (ex AF_UNIX)
AF_ROUTE	Routing socket
AF_ROUTE	Key socket (sicurezza in IPv6)

L'argomento type specifica quale tipo di protocollo vogliamo utilizzare all'interno della famiglia di protocolli specificata da family.

type	descrizione
SOCK_STREAM	socket di tipo stream (connesso affidabile)
SOCK_DGRAM	socket di tipo datagram
SOCK_DRAW	socket di tipo raw (livello network)

L'argomento protocol di solito è settato a 0, tranne che nel caso dei socket raw.

Non tutte le combinazioni di family e type sono valide. Quelle valide selezionano un protocollo che verrà utilizzato.

	AF_INET	AF_INET6	AF_LOCAL	AF_KEY AF_ROUTE
SOCK_STREAM	TCP	TCP	esiste	
SOCK_DGRAM	UDP	UDP	esiste	
SOCK_DRAW	IPv4	IPv6		esiste

funzione **connect**()

La funzione connect() è usata dal client TCP per stabilire la connessione con un server TCP.

```
#include <sys/socket.h>
int connect (int socketfd, const struct sockaddr *servaddr,
                 socklen_t addrlen);
```

restituisce 0 se la connessione viene stabilita, -1 in caso di errore.

• L'argomento **socketfd** è un descrittore socket ottenuto da una chiamata alla funzione socket().

• L'argomento **servaddr** come visto in precedenza è in realtà per IPv4 un puntatore alla struttura sockaddr_in, e **deve specificare l'indirizzo IP e il numero di porta del server da connettere**.

• L'argomento **addrlen** specifica la dimensione della struttura dati che contiene l'indirizzo del server servaddr, viene di solito assegnata mediante la sizeof(servaddr).

• Il client non deve di solito specificare il proprio indirizzo IP e la propria porta, perchè queste informazioni non servono a nessuno. Quindi può chiedere al sistema operativo di assegnargli una porta TCP qualsiasi, e come indirizzo IP l'indirizzo della sua interfaccia di rete, o dell'interfaccia di rete usata se ne ha più di una. Quindi NON SERVE la chiamata alla bind() prima della connect().

• Nel caso di connessione TCP la connect inizia il protocollo three way handshake spedendo un segmento SYN. La funzione termina o quando la connessione è stabilita o in caso di errore.

• In caso di errore la connect restituisce -1 e la variabile errno è settata a:

- ETIMEDOUT nessuna risposta al segmento SYN
- ECONNREFUSED il server risponde con un segmento RST (reset) ad indicare che nessun processo server è in attesa (stato LISTEN) su quella porta
- EHOSTUNREACH o ENETUNREACH host non raggiungibile
- ed altri ancora.

funzione **bind()** (1)

La funzione bind() collega al socket un indirizzo locale. Per TCP e UDP ciò significa assegnare un indirizzo IP ed una porta a 16-bit.

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

restituisce 0 se tutto OK, -1 in caso di errore.

- L'argomento **sockfd** è un descrittore ottenuto da una socket().
- L'argomento **myaddr** è un puntatore alla struttura sockaddr_in, e specifica l'eventuale indirizzo IP **locale** e l'eventuale numero di porta **locale** a cui il sistema operativo deve collegare il socket.
- L'argomento **addrlen** specifica la dimensione della struttura myaddr.

- L'applicazione può collegarsi o no ad una porta.
  - Di solito il server si collega ad una porta nota (well know port). Fa eccezione il meccanismo delle RPC.
  - I client di solito non si collegano ad una porta con la bind.
  - In caso non venga effettuato il collegamento con una porta, il kernel effettua autonomamente il collegamento con una porta qualsiasi (ephemeral port) al momento della connect (per il client) o della listen (per il server).
- L'applicazione può specificare (con la bind) per il socket un indirizzo IP di un'interfaccia dell'host stesso.
  - Per un TCP client ciò significa assegnare il source IP address che verrà inserito negli IP datagram, spediti dal socket.
  - Per un TCP server ciò significa che verranno accettate solo le connessioni per i client che chiedono di connettersi proprio a quell'IP address.
  - Se il TCP client non fa la bind() o non specifica un IP address nella bind(), il kernel sceglie come source IP address, nel momento in cui il socket si connette, quello della interfaccia di rete usata.
  - Se il server non fa il bind con un IP address, il kernel assegna al socket come indirizzo IP locale quello contenuto nell'IP destination address del datagram IP che contiene il SYN segment ricevuto

funzione **bind()** (2)

Chiamando la bind() si può specificare o no l'indirizzp IP e la porta, assegnando valori ai due campi **sin_addr** e **sin_port** della struttura sockaddr_in passata alla bind come secondo argomento.
A seconda del valore otteniamo risultati diversi, che sono qui elencati, nella tabella che si riferisce solo al caso:

IP_address sin_addr	port sin_port	Risultato
wildcard	0	il kernel sceglie IP address e porta
wildcard	nonzero	il kernel sceglie IP address, porta fissata
Local IP Address	0	IP address fissato, kernel sceglie la porta
Local IP Address	non zero	IP address e porta fissati dal processo

Specificando il numero di porta 0 il kernel sceglie collega il socket ad un numero di porta temporaneo nel momento in cui la bind() è chiamata.
Specificando la wildcard (mediante la costante **INADDR_ANY** per IPv4) il kernel non sceglie l'indirizzo IP locale fino a che o il socket è connesso (se TCP) o viene inviato il primo datagram per quel socket (se UDP).

L'assegnazione viene fatta con le istruzioni:
struct sockaddr_in localaddr;
localaddr.sin_addr.s_addr = htonl(INADDR_ANY);
localaddr.sin_port = htons(port_number);

Se con la bind si lascia al kernel la scelta di IP address locale o port number locale, una volta che il kernel avrà scelto, si potrà sapere quale IP address e quale port number è stato scelto mediante la funzione **getsockname()**.

funzione **listen**()

La funzione listen **è chiamata solo dal TCP server** e esegue due azioni:
1) ordina al kernel di far passare il socket dallo stato iniziale CLOSED allo stato LISTEN, e di accettare richieste di inizio connessione per quel socket, accodandole in delle code del kernel.
2) specifica al kernel quante richieste di inizio connessione può accodare al massimo per quel socket.

```
#include <sys/socket.h>
int listen (int socketfd, int backlog );
```

restituisce 0 se tutto OK, -1 in caso di errore.

- L'argomento **socketfd** è un descrittore ottenuto da una socket().
- L'argomento **backlog** è un intero che specifica quante richieste di inizio connessione (sia connessioni non ancora stabilite, cioè che non hanno ancora raggiunto lo stato ESTABLISHED, sia connessioni stabilite) il kernel può mantenere in attesa nelle sue code.
- Quando un segmento SYN arriva da un client, se il TCP verifica che c'è un socket per quella richiesta, crea una nuova entry in una **coda delle connessioni incomplete**, e risponde con il suo FIN+ACK secondo il 3-way handshake. L'entry rimane nella coda fino a che il 3-way è terminato o scade il timeout.
- Quando il 3-way termina normalmente, la connessione viene instaurata, e la entry viene spostata in **una coda delle connessioni completate**.
- Quando il server chiama la accept, la prima delle entry nella **coda delle connessioni completate** viene consegnata alla accept() che ne restituisce l'indice come risultato, ovvero restituisce un nuovo socket che identifica la nuova connessione.
- Se quando il server chiama la accept(), la coda delle connessioni completate è vuota, la accept resta in attesa.
- L'argomento backlog specifica il numero totale di entry dei due tipi di code.
- Solitamente si usa 5, per http daemon si usano valori molto grandi.

funzione **accept**()

La funzione accept **è chiamata solo dal TCP server** e restituisce la prima entry nella **coda delle connessioni già completate** per quel socket. Se la coda è vuota la accept resta in attesa.

```
#include <sys/socket.h>
int accept (int socketfd, struct sockaddr *cli_addr,
                 socklen_t *ptraddrlen);
```

restituisce un descrittore socket >=0 se tutto OK, -1 in caso di errore.

L'argomento **socketfd** è un descrittore ottenuto da una socket() e in seguito processato da bind() e listen(). E' il cosiddetto **listening socket**, ovvero il socket che si occupa di insturare le connessioni con i client che lo richiedono, secondo le impostazioni definite dalla bind() e dalla listen(). Tale listening socket viene utilizzato per accedere alla coda delle connessioni instaurate come visto per la listen().

- L'argomento **cli_addr** è un puntatore alla struttura sockaddr_in, su cui la funzione accept scrive l'indirizzo IP **del client** e il numero di porta **del client**, con cui è stata instaurata la connessione a cui si riferisce il socket che viene restituito come risultato .
- L'argomento **ptraddrlen** è un puntatore alla dimensione della struttura cli_addr che viene restituita.

Se accept termina correttamente restituisce un nuovo descrittore di socket che è il **connected socket**, cioè si riferisce ad una connessione instaurata con un certo client secondo le regole del listening socket socketfd passato come input. Il connected socket verrà utilizzato per scambiare i dati nella nuova connessione.

Il **listening socket** socketfd (il primo argomento) mantiene anche dopo la accept le impostazioni originali, e può essere riutilizzato in una nuova accept per farsi affidare dal kernel una nuova connessione.

funzione **close**()

La funzione close è utilizzata normalmente per chiudere un descrittore di file, è utilizzata per chiudere un socket e terminare una connessione TCP.

int **close** (int socketfd);

restituisce 0 se tutto OK, -1 in caso di errore.

L'argomento **socketfd** è un descrittore di socket.

• Normalmente la chiamata alla close() fa marcare "closed" il socket, e la funzione ritorna il controllo al chiamante. Il socket allora non può più essere usato dal processo, ovvero non può più essere usato come argomento di read e write.

• **Però il TCP continua ad utilizzare il socket trasmettendo i dati** che eventualmente stanno nel suo buffer interno, fino a che non sono stati trasmessi tutti. **In caso di errore (che impedisce questa trasmissione) successivo alla close l'applicazione non se ne accorge e l'altro end system non riceverà alcuni dei dati**.

• Esiste un'opzione però (la **SO_LINGER socket option**) che modifica il comportamento della close, facendo in modo che la **close restituisca il controllo al chiamante solo dopo che tutti i dati nei buffer sono stati correttamente trasmessi** e riscontrati.

• Se un **socket connesso sockfd è condiviso da più processi** (padre e figlio ottenuto da una fork), il **socket mantiene il conto di quanti sono i processi a cui appartiene**. In tal caso la chiamata alla close(sockfd) per prima cosa **decrementa di una unità questo contatore**, e **non innesca la sequenza FIN+ACK+FIN+ACK di terminazione della connessione fino a che tale contatore è maggiore di zero**, perchè **esiste ancora un processo che tiene aperta la connessione**.

• Per innescare veramente la sequenza di terminazione, anche se ci sono ancora processi per quella connessione si usa la funzione **shutdown()**.

funzione **getsockname**()

La funzione getsockname serve a **conoscere l'indirizzo** di protocollo (IP e port number) **dell'host locale** associato ad un certo descrittore di socket connesso.

int **getsockname** (int socketfd, struct sockaddr ***Localaddr**,
socklen_t ***ptr_addrlen**);

restituisce 0 se tutto OK, -1 in caso di errore.

Il primo argomento **socketfd** è un descrittore di socket connesso.
Il secondo argomento **Localaddr** è un puntatore ad una struttura di tipo sockaddr, in cui la funzione metterà l'indirizzo **locale** della connessione.
Il terzo argomento **ptr_addrlen** è un puntatore ad intero in cui la funzione metterà la dimensione della struttura scritta.

Questa funzione viene utilizzata in varie situazioni:

In un client, **dopo una connect se non è stata effettuata la bind**, e quindi non si è specificato nessun indirizzo: in tal caso getsockname permette di conoscere l'indirizzo IP e la porta assegnati dal kernel alla connessione.

In un client dopo una bind in cui come port number è stato specificato il valore 0, con il quale si è informato il kernel di scegliere lui la porta. In tal caso la getsockname restituisce il numero di porta locale assegnato dal kernel.

In un server multihomed, dopo una accept preceduta da una bind in cui come indirizzo IP LOCALE è stata messa la wildcard INADD_ANY, ciòè una volta che si sia stabilita una connessione, la getsockname permette al server di sapere quale indirizzo IP ha la propria interfaccia di rete utilizzata per la connessione.

funzione **getpeername**()

La funzione getpeername serve a **conoscere l'indirizzo** di protocollo (IP e port number) **dell'host remoto** associato ad un certo descrittore di socket connesso.

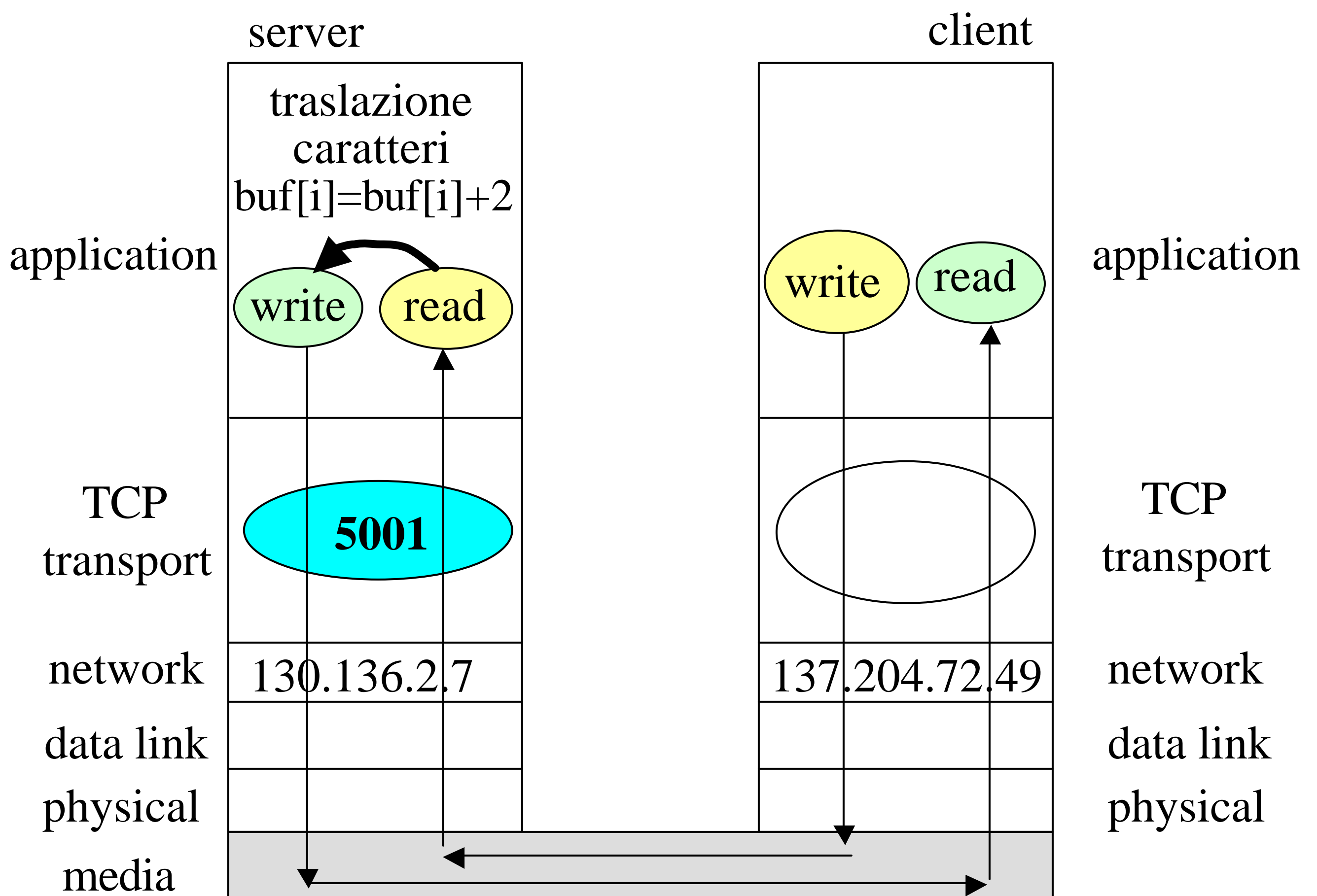
int **getpeername** (int socketfd, struct sockaddr ***Remoteaddr**,
socklen_t ***ptr_addrlen**);

restituisce 0 se tutto OK, -1 in caso di errore.

Il primo argomento **socketfd** è un descrittore di socket connesso.
Il secondo argomento **Remoteaddr** è un puntatore ad una struttura di tipo sockaddr, in cui la funzione metterà l'indirizzo **remoto** della connessione.
Il terzo argomento **ptr_addrlen** è un puntatore ad intero in cui la funzione metterà la dimensione della struttura scritta.

esempio di trasmissione con TCP

Vediamo un semplice esempio di programma che sfrutta i socket TCP per instaurare una connessione tra un client e un server, trasmettere dal client al server una stringa di caratteri, aspettare che il server modifichi questi caratteri (tranne l'ultimo, lo'\0' che delimita la stringa) shiftandoli di due posizioni (es: 'a' diventa 'c', '2' diventa '4') e li rispedisca indietro così traslati, infine stampare il risultato.
Il server è l'host **130.136.2.7**, mentre il client è l'host **137.204.72.49**.
Il punto di accesso del servizio di traslazione è la porta TCP 5001.



Il codice completo (con la gestione degli errori) dei due programmi che realizzano l'esempio qui mostrato è disponibile allo indirizzo http://www.cs.unibo.it/~ghini/didattica/sistemi3/TCP1/TCP1.html

server TCP per l'esempio

```
/* servTCP.c     eseguito sull'host 130.136.2.7 */
void main(void)    {
struct sockaddr_in Local, Client; short int local_port_number=5001;
char buf[SIZEBUF];  int sockfd, newsockfd, n, nread, nwrite, len;

/* prende un socket per stream TCP */
sockfd = 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);
Local.sin_port              =  htons(local_port_number);
bind ( sockfd, (struct sockaddr*) &Local, sizeof(Local));
/* accetta max 10 richieste simultanee di inizio conness., da adesso */
listen(sockfd, 10 );
/* accetta la prima conness. creando un nuovo socket per la conness. */
newsockfd = accept(sockfd, (struct sockaddr*) &Cli, &len);
/* riceve la stringa dal client */
nread=0;
while( (n=read(newsockfd, &(buf[nread]), MAXSIZE )) >0) {
    nread+=n;
    if(buf[nread-1]=='\0')  break; /* fine stringa */
    }
/* converte i caratteri della stringa */
for( n=0; n<nread  -1  ; n++) buf[n] = buf[n]+2;
/* spedisce la stringa traslata al client */
nwrite=0;
while((n=write ( newsockfd, &(buf[nwrite]),nread-nwrite)) >0 )
    nwrite+=n;
/* chiude i socket */
close(newsocketfd); close(socketfd);
}
```

client TCP per l’esempio

```
/* cliTCP.c     eseguito sull'host 137.204.72.49 */
void main(void)    {
struct sockaddr_in Local, Serv; short int remote_port_number=5001;
char msg[]="012345ABCD";  int sockfd, newsockfd, n, nread, nwrite, le
/* prende un socket per stream TCP */
sockfd = socket (AF_INET, SOCK_STREAM, 0);
/* collega il socket senza specificare indirizzo IP e porta TCP locali */
memset ( &Local, 0, sizeof(Local) );
Local.sin_family            =  AF_INET;
Local.sin_addr.s_addr   =  htonl(INADDR_ANY);
Local.sin_port               =  htons(0);
bind ( sockfd, (struct sockaddr*) &Local, sizeof(Local));
/* specifica l'indirizzo del server, e chiede la connessione */
memset ( &Serv, 0, sizeof(Serv) );
Serv.sin_family              =  AF_INET;
Serv.sin_addr.s_addr      =  inet_addr ( string_remote_ip_address);
Serv.sin_port                 =  htons(remote_port_number);
connect ( socketfd, (struct sockaddr*) &Serv, sizeof(Serv));
/* spedisce la stringa al server */
len = strlen(msg)+1;   nwrite=0;
while((len>nwrite)&&(n=write(socketfd,&(msg[nwrite]),len-nwrite))>0))
    nwrite+=n;
nread=0; /* riceve la stringa traslata dal server */
while( (n=read(newsockfd, &(msg[nread]), MAXSIZE )) >0) {
    nread+=n;
    if(buf[nread-1]=='\0')  break; /* fine stringa */
    }
printf("%s\n", msg);  /* stampa la stringa traslata */
/* chiude i socket e termina*/
close(socketfd);
}
```

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**.

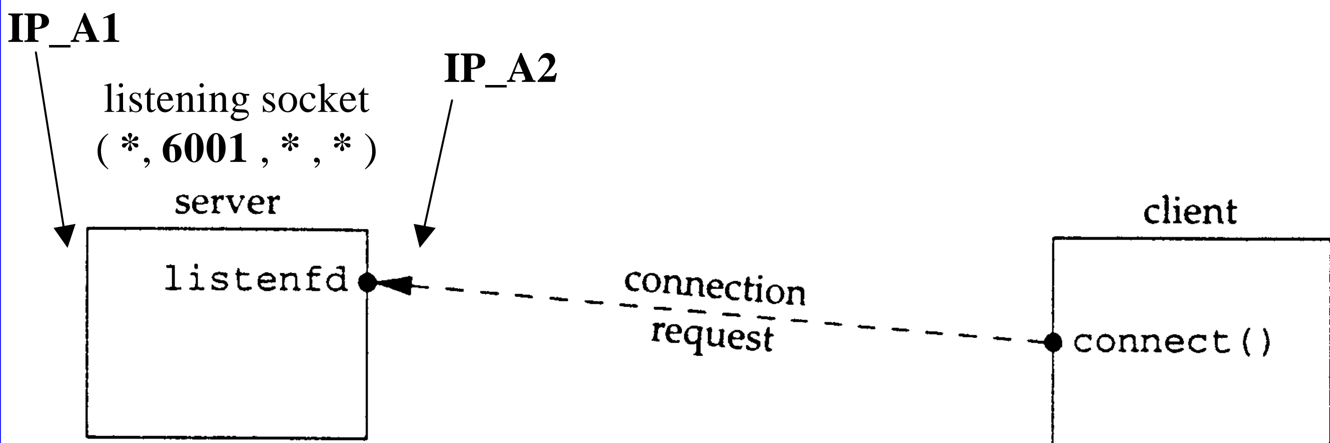
```
listenfd = socket (AF_INET, SOCK_STREAM, 0);
/* collega il socket ad un indirizzo IP locale e una porta TCP locale */
memset ( &Local, 0, sizeof(Local) );
Local.sin_family          =  AF_INET;

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)

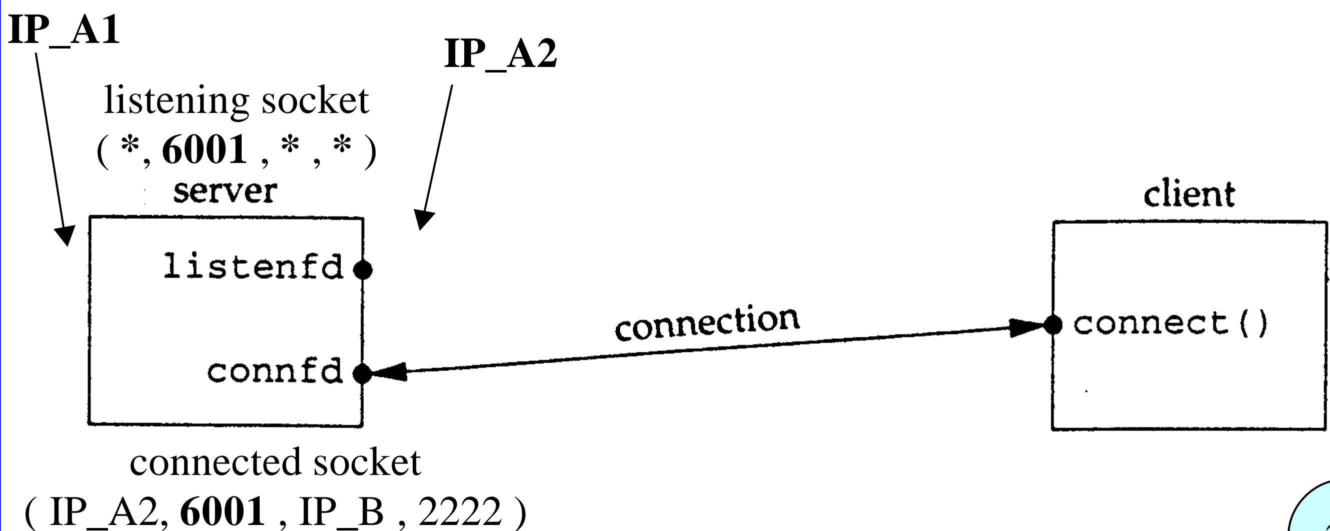
La quaterna *(IP locale, Port Number locale , IP remoto, Port Number remoto)* che identifica univocamente una connessione TCP viene di solito chiamata **socket pair**.

SERVER	CLIENT
IP = IP_A1	IP = IP_B
IP = IP_A2	port = 2222

dopo la listen(), prima della accept()



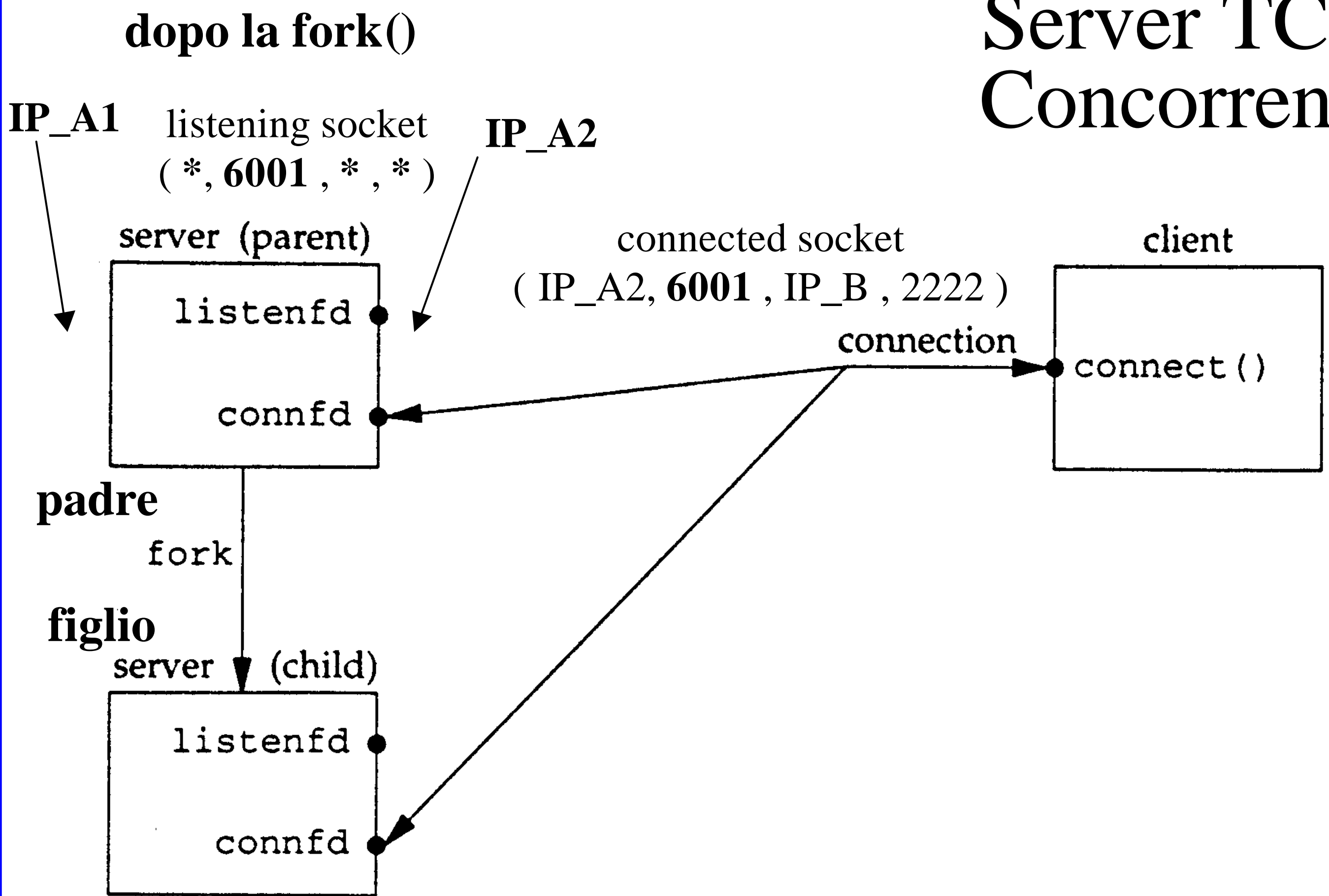
dopo la accept() , prima della fork()



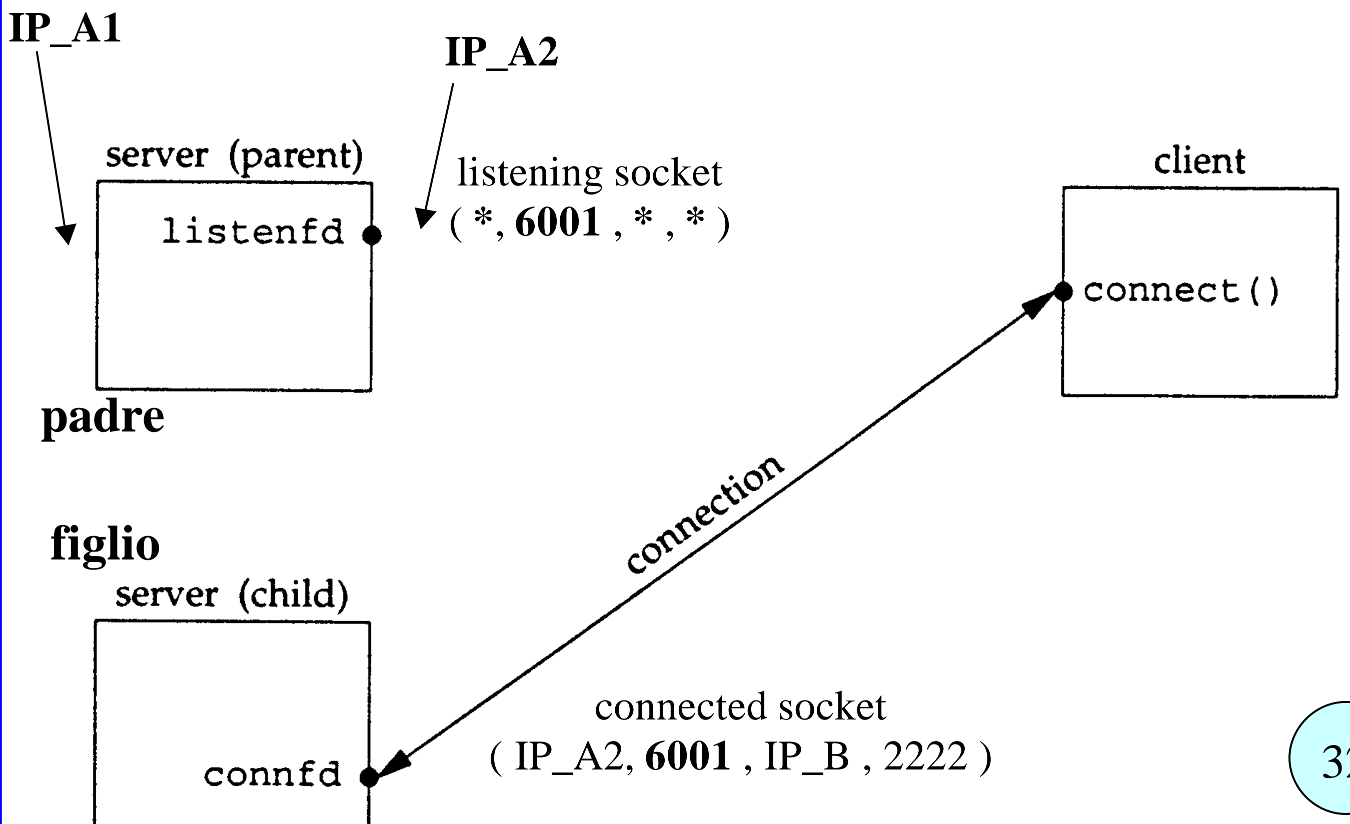
Server TCP Concorrenti

(4)

dopo la fork()



dopo la close(connfd) del padre, e la close(listenfd) del figlio



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 */
      }
  }
}
}
```

funzione **poll**()

La funzione poll è una specie di select. Prevede un insieme di argomenti più facili da usare, e prevede zone separate per l'input (la descrizione dei descrittori da considerare e le condizioni) ed il risultato restituito (l'elenco dei descrittori disponibili per l'I/O). Utilizza una struttura dati per indicare i descrittori da considerare ed il risultato ottenuto:

```
struct pollfd {
        int       fd;        /* descrittore da controllare */
        short     events;    /* eventi di interesse per fd */
        shorts    revents;   /* eventi che sono accaduti per fd */
}
#include <poll.h>
int poll ( struct pollfd *fdarray, unsigned long nfds, int timeout_msecs);
```

La funzione restituisce -1 in caso di errore,
0 se il timeout fissato è scaduto,
>0 se ci sono dei descrittori che hanno verificato una delle condizioni richieste, ed il numero restituito è il numero dei descrittori pronti.