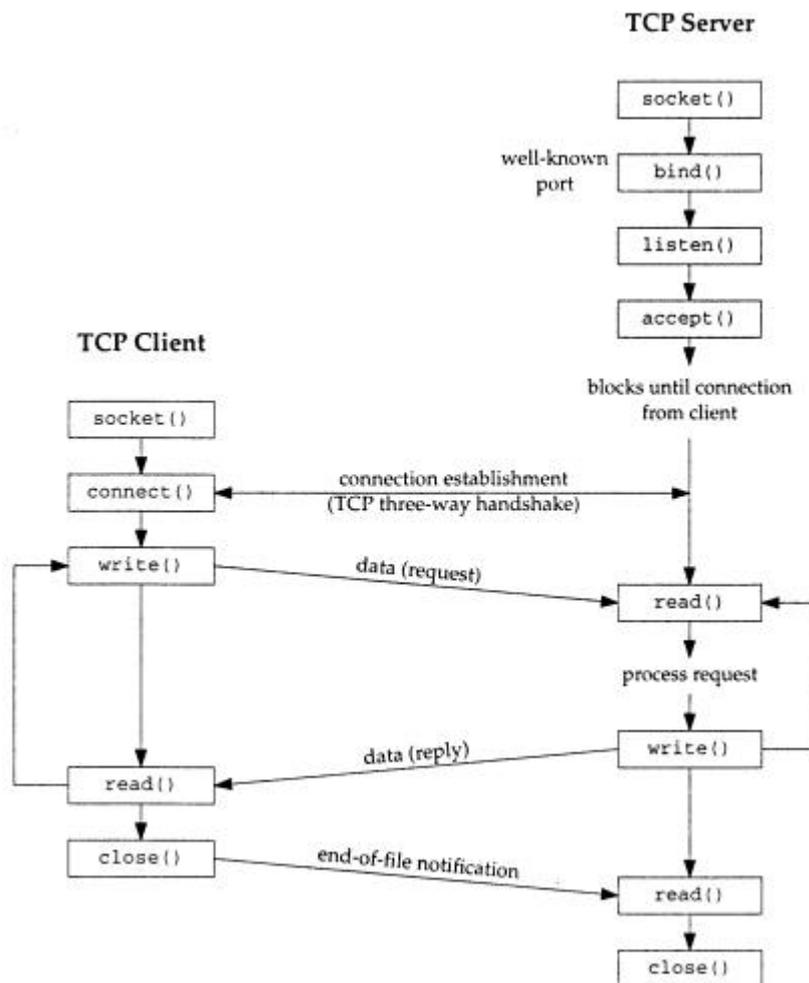


Socket per TCP: Fondamenti



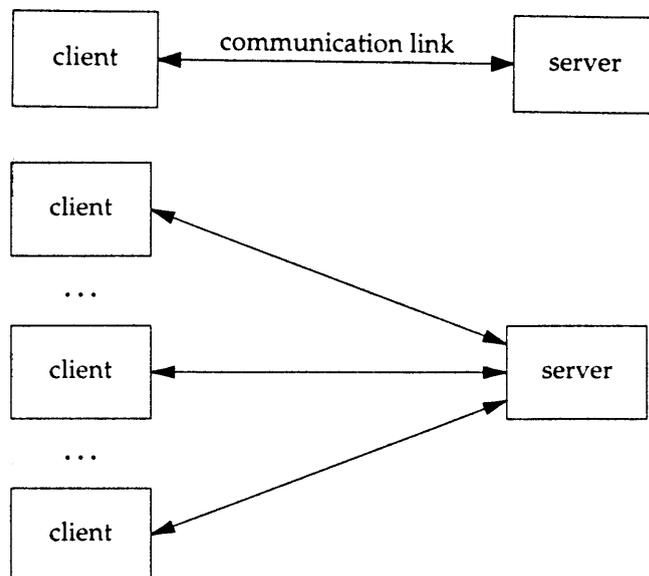
Network Applications

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

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.

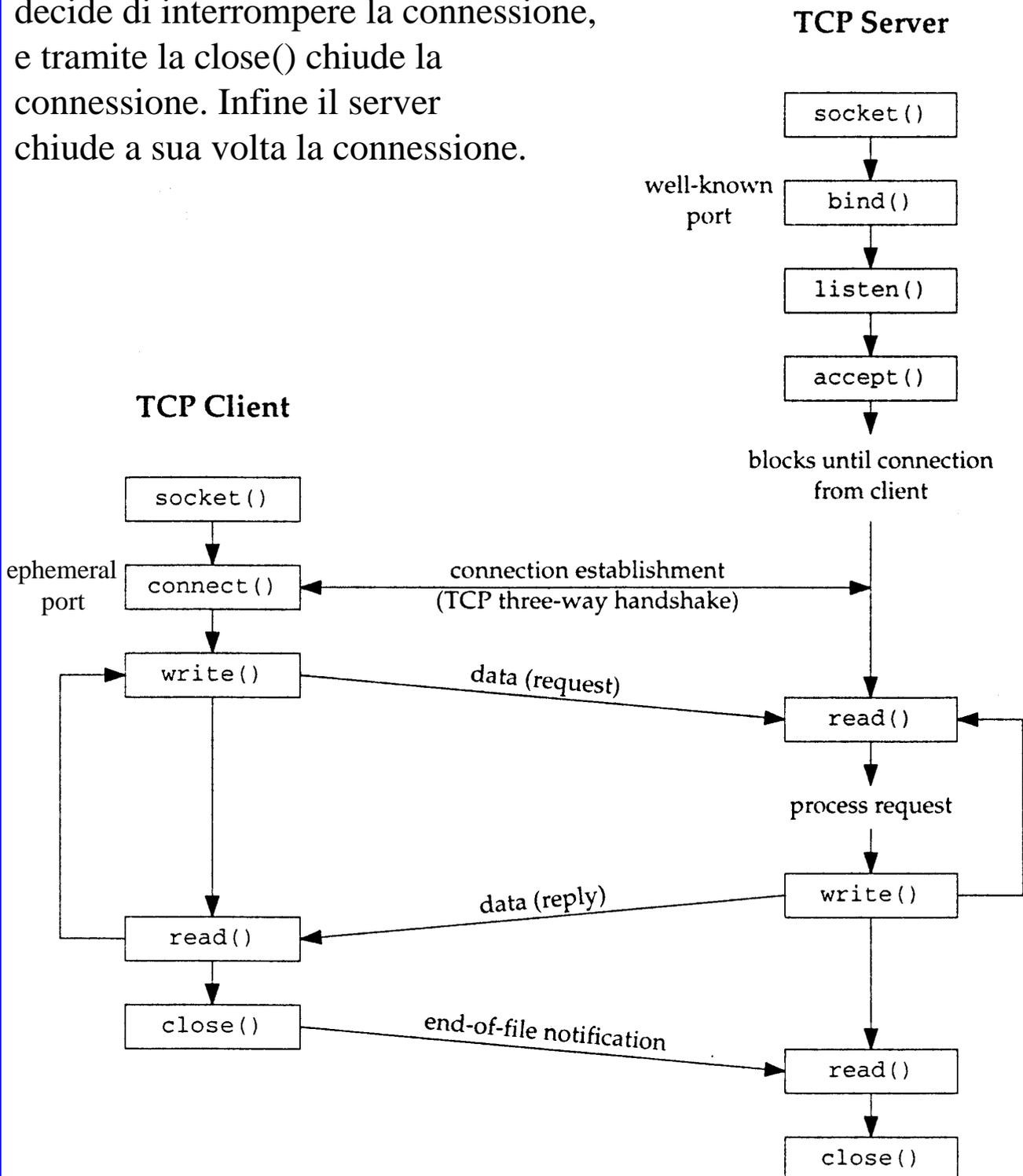
Nell'esempio (ma non è obbligatorio) il client spedisce una richiesta al server, questo risponde trasmettendo alcuni dati. Questa trasmissione bidirezionale continua fino a che uno dei due (il client nell'esempio)

decide di interrompere la connessione,

e tramite la `close()` chiude la

connessione. Infine il server

chiude a sua volta la connessione.

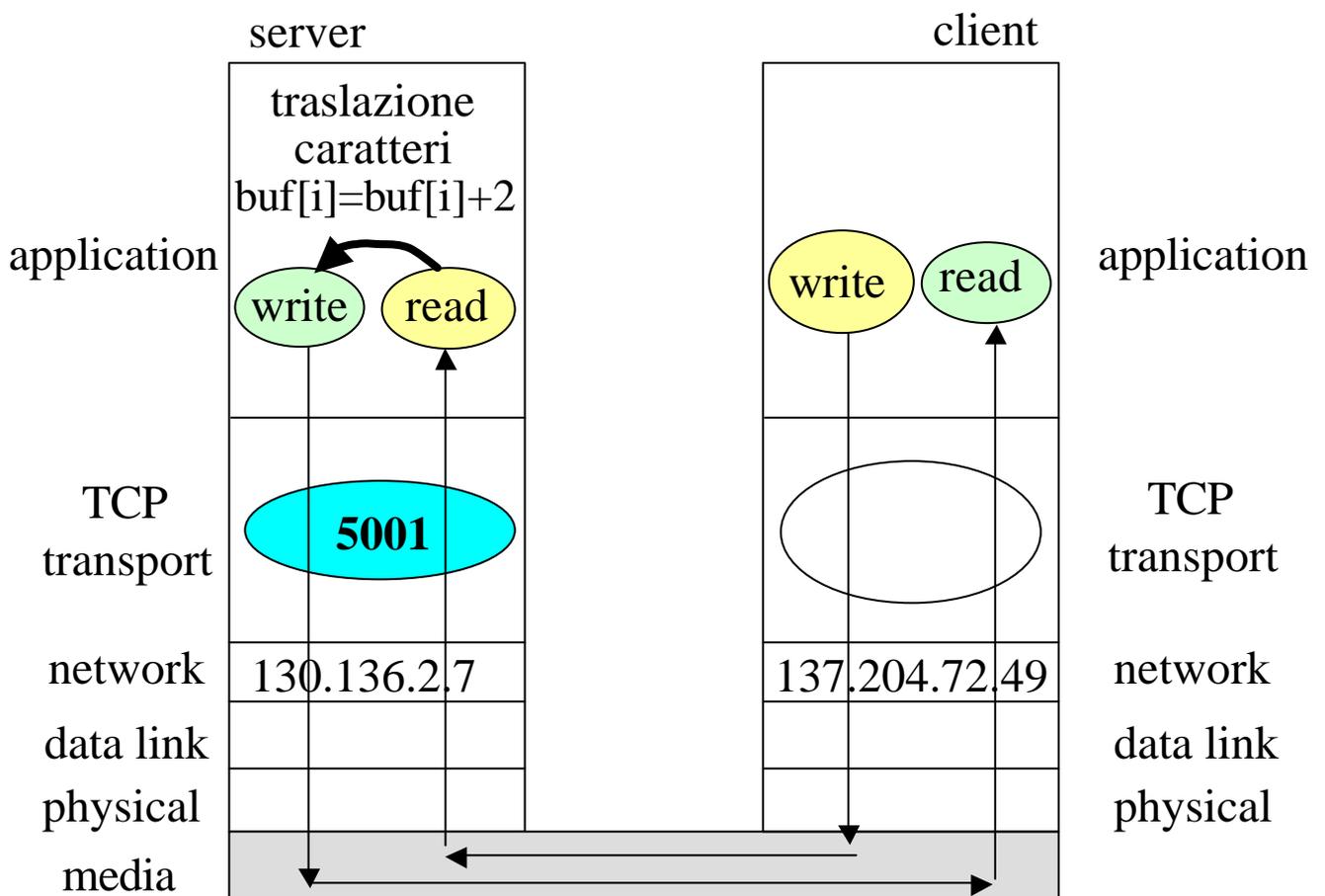


esempio di trasmissione con TCP

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, len;

/* 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 ( sockfd, (struct sockaddr*) &Serv, sizeof(Serv));
/* spedisce la stringa al server */
len = strlen(msg)+1; nwrite=0;
while((len>nwrite)&&(n=write(sockfd,&(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(sockfd);
}
```

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	<sys/socket.h> <i>AF_INET per IPv4, AF_INET6 per IPv6, AF_LOCAL per indir. locali unix (per pipe ecc..)</i>
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 ( sockfd, (struct sockaddr *)&server, sizeof(server) );
```

Socket Address Structures (2)

La generica struttura dell'indirizzo è dunque così 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. Restituisce 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’è 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 corrispondenti 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.

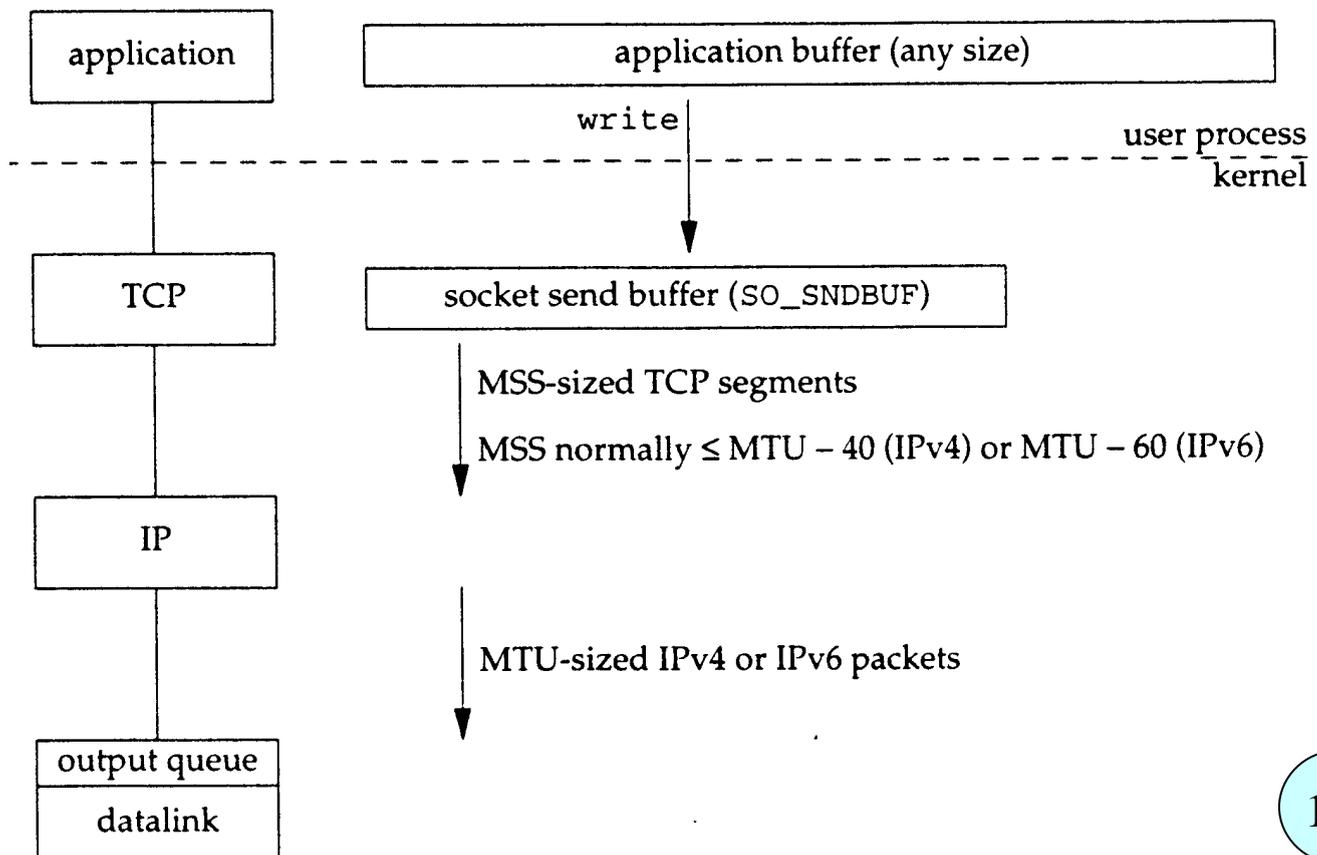
I/O su Socket TCP: (2)

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 byte sul socket TCP, il kernel cerca di copiare n byte dal buffer dell'appl. al buffer del socket. Se il buffer del socket è 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 fine 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 readn() e writen() 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 nwritten; 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);
}
```

funzione `socket()`

La prima azione per fare dell'I/O da rete è la chiamata alla funzione `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
<code>AF_INET</code>	IPv4 protocol
<code>AF_INET6</code>	IPv6 protocol
<code>AF_LOCAL</code>	Unix domain protocols (ex <code>AF_UNIX</code>)
<code>AF_ROUTE</code>	Routing socket
<code>AF_ROUTE</code>	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
<code>SOCK_STREAM</code>	socket di tipo stream (connesso affidabile)
<code>SOCK_DGRAM</code>	socket di tipo datagram
<code>SOCK_DRAW</code>	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.

	<code>AF_INET</code>	<code>AF_INET6</code>	<code>AF_LOCAL</code>	<code>AF_KEY</code> <code>AF_ROUTE</code>
<code>SOCK_STREAM</code>	TCP	TCP	esiste	
<code>SOCK_DGRAM</code>	UDP	UDP	esiste	
<code>SOCK_DRAW</code>	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 sockfd, const struct sockaddr *servaddr,  
             socklen_t addrlen);
```

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

- L'argomento **sockfd** 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'indirizzo 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 sockfd, int backlog );
```

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

- L'argomento **sockfd** 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 **codice 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 codice delle connessioni completate**.
- Quando il server chiama la `accept`, la prima delle entry nella **codice 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 sockfd, struct sockaddr *cli_addr,  
            socklen_t *ptraddrlen);
```

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

L'argomento **sockfd** 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 instaurare 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 **sockfd** passato come input. Il connected socket verrà utilizzato per scambiare i dati nella nuova connessione.

Il **listening socket** **sockfd** (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 sockfd);`

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

L'argomento `sockfd` 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 sockfd, struct sockaddr *Localaddr,  
                 socklen_t *ptr_addrlen );
```

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

Il primo argomento **sockfd** 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_addr**len 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 INADDR_ANY, cioè 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 sockfd, struct sockaddr *Remoteaddr,  
                 socklen_t *ptr_addrln );
```

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_addrln** è un puntatore ad intero in cui la funzione metterà la dimensione della struttura scritta.