Advanced TCP Socket

Le Opzioni per i Socket

Le opzioni per i socket sono controllate mediante tre tipi di primitive:
1) le funzioni **getsockopt()** e **setsockopt(),** che permettono di configurare alcune **caratteristiche proprie solo dei socket**, quali dimensioni dei segmenti, controlli sulla funzionalità della connessione, **modalita' di utilizzo degli indirizzi**;
2) la funzione **fcntl()**, che invece consente di settare caratteristiche comuni a tutti i descrittori di I/O, quali **comportamento bloccante o non bloccante** e **I/O guidato dai signal**.
3) la funzione **ioctl()**, che ripete operazioni delle fcntl ed inoltre effettua operazioni riguardanti ARP e routing.

Le opzioni tipiche per i socket possono essere settate solo quando il socket è reso disponibile all'interfaccia di programmazione. Consideriamo ad es. il caso di un **connected socket** ottenuto da un server in risposta ad una chiamata alla accept() a partire da un **socket listening**. Il connected socket viene creato dopo la listen() quando arriva una richiesta di connessione dal client, e **tutto il procedimento di instaurazione della connessione avviene senza che il programmatore possa intervenire, in quanto il socket creato verrà reso disponibile al programmatore solo quando questo farà eseguire la accept()**.

Per rendere possibile configurare il socket anche in questa situazione temporanea, l'interfaccia socket implementa la politica seguente: **il connected socket eredita dal listening socket alcune opzioni, invece di assumere le opzioni di default**. Queste opzioni sono: SO_DEBUG, SO_DONTROUTE, **SO_KEEPALIVE**, **SO_LINGER**, SO_OOBINLINE, **SO_RCVBUF** e **SO_SNDBUF**. In tal modo se vogliamo che il connected socket abbia queste opzioni gia' durante la fase del thee-way-handshake dobbiamo settare in quel modo il listening socket.
In http://www.cs.unibo.it/~ghini/didattica/sistemi3/SOCKOPTS/checkopts.c e' implementato un esempio di lettura delle opzioni di default di un socket.

funzioni **getsockopt**() e s**etsockopt**()

Il formato di queste due funzioni è il seguente:
#include <sys/socket.h>
int **getsockopt** (int **sockfd**, int **level**, int **optname**,
void ***optval**, socklen_t ***optlen**);
int **setsockopt** (int **sockfd**, int **level**, int **optname**,
const void ***optval**, socklen_t **optlen**);
restituiscono 0 se tutto OK, -1 in caso di errore.

- Il primo argomento **socketfd** è un descrittore di socket aperto con una socket(), su cui operano le funzioni.
- L'argomento **level** indica a che livello di protocollo deve agire l'opzione: a livello di **socket generale** (level=SOL_SOCKET), a livello **IPv4**(IPPROTO_IP), IPv6(IPPROTO_IPV6), ICMPv6 (IPPROTO_ICMPV6), a livello **TCP** (level=IPPROTO_TCP).
- **optname** specifica l'opzione da leggere /settare.
- **optval** è un puntatore ad una variabile di tipo dipendente dall'opzione specificata, che contiene il valore nuovo da settare dell'opzione da configurare (caso setsockopt) o che conterrà il valore attuale dell'opzione (caso getsockopt).
- **optlen** contiene la dimensione della variabile puntata da optval.

Le opzioni possono effettuare due diverse operazioni:
- settare o resettare un flag,
- assegnare o leggere un valore più complesso.

Nel caso dei flags, il valore restituito o passato (optval) punta ad un intero, che vale zero se l'opzione e' disabilitata, vale nonzero se l'opzione e' abilitata.
Nel caso non flags optval punta ad un dato di tipo diverso, come indicato nella seguente tabella:

Tabella delle Opzioni Socket Generiche, cioè di livello Socket, per **getsockopt**() e **setsockopt**()

Per queste opzioni, nelle funzioni getsockopt() e setsockopt() deve essre utilizzato come secondo argomento level=SOL_SOCKET.

optname	get	set	Description	Flag	Datatype
SO_BROADCAST	•	•	permit sending of broadcast datagrams	•	int
SO_DEBUG	•	•	enable debug tracing	•	int
SO_DONTROUTE	•	•	bypass routing table lookup	•	int
SO_ERROR	•		get pending error and clear		int
SO_KEEPALIVE	•	•	periodically test if connection still alive	•	int
SO_LINGER	•	•	linger on close if data to send		linger{}
SO_OOBINLINE	•	•	leave received out-of-band data inline	•	int
SO_RCVBUF	•	•	receive buffer size		int
SO_SNDBUF	•	•	send buffer size		int
SO_RCVLOWAT	•	•	receive buffer low-water mark		int
SO_SNDLOWAT	•	•	send buffer low-water mark		int
SO_RCVTIMEO	•	•	receive timeout		timeval{}
SO_SNDTIMEO	•	•	send timeout		timeval{}
SO_REUSEADDR	•	•	allow local address reuse	•	int
SO_REUSEPORT	•	•	allow local address reuse	•	int
SO_TYPE	•		get socket type		int
SO_USELOOPBACK	•	•	routing socket gets copy of what it sends	•	int

Opzioni Socket Generiche (1)

Consideriamo le principali opzioni caratteristiche di tutti i socket, quelle identificate dal livello **SOL_SOCKET**.

SO_BROADCAST questa opzione **abilita o disabilita la possibilità per un socket di spedire messaggi broadcast**. Viene applicato solo ai socket datagram (DGRAM) e solo se la rete sottostante lo permette (es: ethernet, non punto a punto). **Per default questa opzione è disabilitata**, per impedire ad un processo di spedire accidentalmente un datagram in broadcast, ad es. se l'indirizzo IP di destinazione viene preso a linea di comando e si digita per errore un indirizzo di broadcast. In tal caso il kernel si accorge di avere a che fare con un datagram di broadcast il cui invio è disabilitato, e restituisce un errore di tipo EACCES.

SO_DEBUG questa opzione è supportata solo da TCP, e ordina al kernel di mantenere informazioni su tutti i pacchetti spediti o ricevuti da/a un certo socket, in una coda circolare. Il programma trcp esaminera' questa coda.

SO_DONTROUTE questa opzione e' applicata per bypassare il normale meccanismo di routing dei pacchetti in uscita, ad es: **per farli uscire da un'interfaccia di rete diversa da quella prevista dalle tabelle interne di routing**. Viene usata ad es. dai processi daemon del routing (routed o gated) per instradare un pacchetto sull'interfaccia giusta quando le tabelle di routing sono sbagliate.

SO_KEEPALIVE questa **opzione è applicata solo agli stream TCP, per verificare se una connessione che da molto tempo non scambia dati debba essere chiusa o no**. Il motivo per cui una connessione deve essere chiusa da un end system e' che l'altro end system a) e' down, b) non e' raggiungibile (rete partizionata o problema nel routing), c) non e' piu' interessato a quella connessione.

Opzioni Socket Generiche (2)

SO_KEEPALIVE (continuazione) Quando un socket TCP ha questa opzione settata, **se nessun dato viene scambiato in una delle due direzioni** della connessione per **2 ore**, il TCP spedisce un segmento, detto **keepalive probe**, all'altro end-system, per verificarne la situazione, e si aspetta di ricevere un ACK.

1) se il peer risponde con un ACK tutto e' OK, ed il TCP mandera' un nuovo probe dopo altre due ore di inattivita'.

2) se il peer risponde con un segmento RST (reset), significa che e' andato in crash e poi ha effettuato il reboot. Il socket allora viene chiuso, ed la variabile d'errore del socket settata a ECONNRESET.

3) se non c'e' risposta dal peer, il TCP riprova a mandare altri 8 segmenti keepalive probe, ogni 75 secondi, aspettando risposta.

3.1) Se non viene ricevuta alcuna risposta il socket e' chiuso e la var. d'errore settata a ETIMEOUT.

3.2) se invece viene ricevuta un ICMP error in risposta ad uno dei keepalive probe, il socket viene chiuso, e viene restituito l'errore indicato dall'ICMP, che sara' di tipo EHOSTUNREACH, ovvero l'host non e' raggiungibile.

La specifica Posix.1g stabilisce anche le modalita' per settare lintervallo di attesa (le 2 ore) ad un valore diverso, ma tale opzione e' implementata raramente.

Questa Opzione **SO_KEEPALIVE serve a stabilire se il peer host e' andato in crash**. Invece **il crash dell'applicazione peer viene individuato e notificato dal TCP peer**. Cioe' **quando nell'altro end system il processo che gestiva il socket va in crash, il TCP di quell'host spedisce un segmento FIN per chiudere la connessione, e questa chiusura può essere individuata con una read() o una select() settata per verificare la possibilita' di leggere da quel socket**. Non esiste altro modo di accorgersi di un crash se non cercando di fare un test per lettura.

Opzioni Socket Generiche (3)

SO_RCVBUF e SO_SNDBUF Queste due opzioni servono a modificare la dimensione dei buffer di ricezione e trasmissione del TCP e dell'UDP.
Per il TCP la dimensione del buffer di ricezione viene mandata all'atto dell'instaurazione della connessione. E' quindi necessario che per il server questa opzione sia settata prima della chiamata alla listen(), mentre per il client deve essere settata prima della chiamata alla connect().
Invece per UDP il buffer di ricezione determina la dimensione massima dei datagram che possono essere accettati.

SO_REUSEADDR e SO_REUSEPORT Queste due opzioni servono a permettere di effettuare la bind() su porte e indirizzi IP gia' utilizzati da qualche altro processo. La SO_REUSEADDR ad es. puo' essere utile nei seguenti casi:

a) si cerca di fare la bind per un listening socket che e' stato chiuso e si vuol fare ripartire, quando ancora esiste un connected socket nato dal listening socket appena chiuso.

b) ci sono piu connected socket che lavorano sulla stessa porta di un host ma con IP diversi. E' il caso dei web server che devono lavorare sulla stessa well know port 80 ma su interfacce diverse.

SO_TYPE Questa opzione puo' essere usata solo in lettura, e restituisce il tipo del socket, SOCK_STREAM o SOCK_DGRAM.

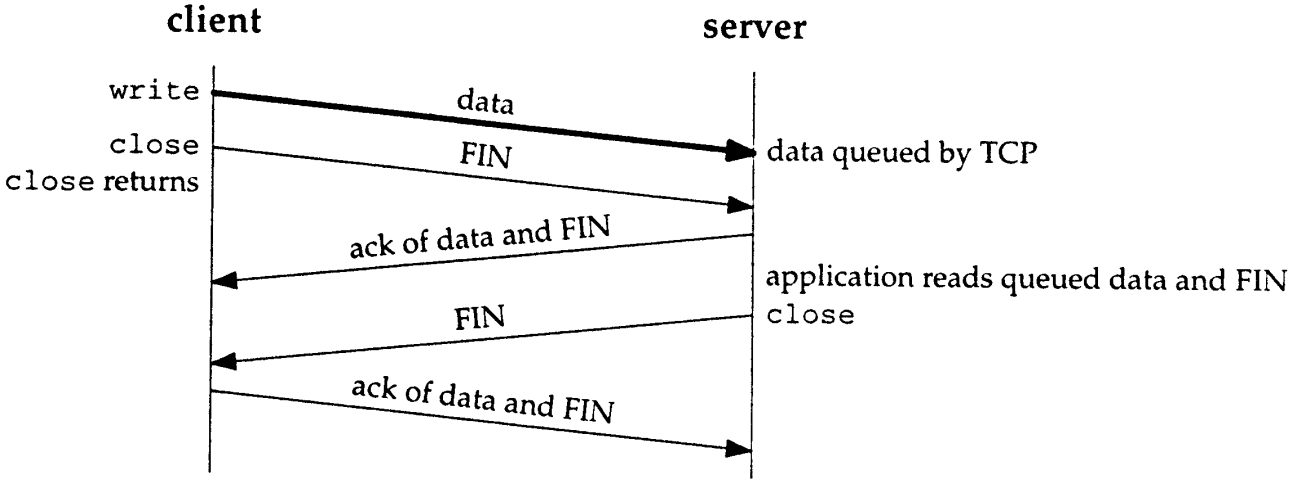
Opzioni Socket Generiche (4)

SO_LINGER Questa opzione determina le modalita' di chiusura realizzate dalla funzione close(). Viene usato come argomento **optval** della setsockopt() un puntatore ad una struttura di tipo:

```
struct linger { int   l_onoff;   /* 0=off , nonzero=on */
                int   l_linger;  /* linger time, Posix.1g vuole secondi */
}
```

Per default la close() restituisce subito il controllo al chiamante, ma se alcuni dati rimangono nei buffer di spedizione il TCP tenta di spedirli.

1) **se l_onoff==0** l'opzione SO_LINGER e' disabilitata, quindi viene settato il modo di default appena visto.



• In questo modo non sappiamo se il TCP peer ha ricevuto tutti i dati, e a maggior ragione non sappiamo se l'application peer li ha ricevuti.

2) se **l_onoff != 0** e **l_linger==0** quando un socket chiama la close(), il TCP chiude la connessione in modo traumatico, non spedendo i dati eventualmente bufferizzati per la spedizione, e mandando un segment RST (reset) all'altro end-system. Non si va nello stato TIME_WAIT e si rischia di danneggiare l'apertura di una nuova connessione con gli stessi indirizzi IP e di porta.

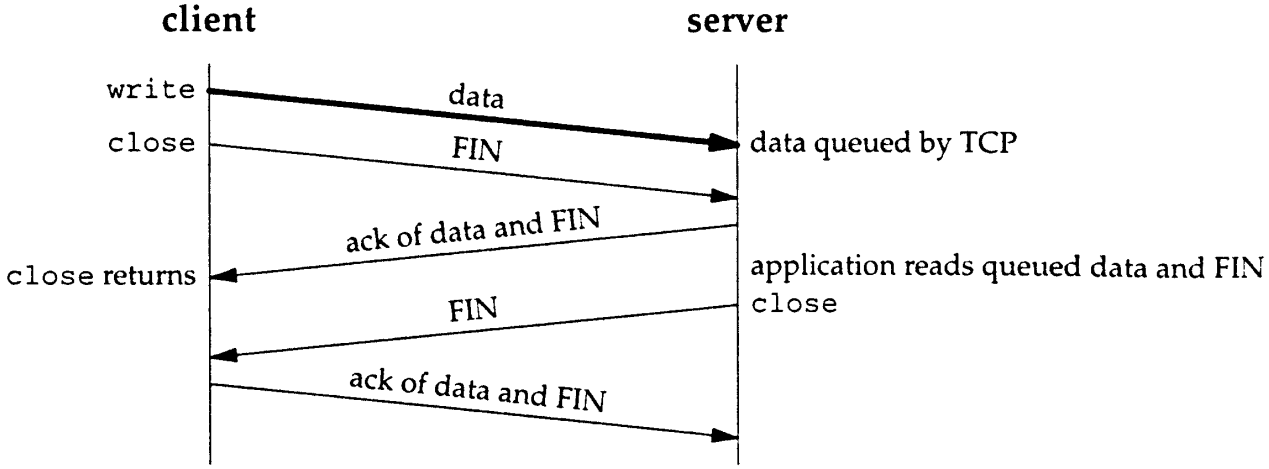
• Anche in questo caso non sappiamo se il TCP peer ha ricevuto tutti i dati, e nemmeno se li ha ricevuti l' application peer.

Opzioni Socket Generiche (5)

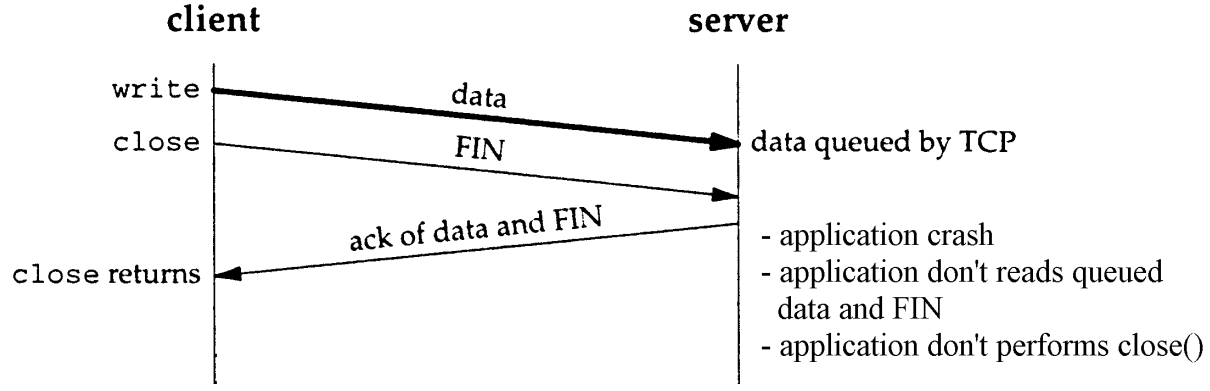
SO_LINGER (continuazione)

3) se **l_onoff != 0** e **l_linger !=0** quando un socket chiama la close(), se il socket e' di tipo bloccante (e' il default) il TCP tenta di spedire i dati eventualmente bufferizzati per la spedizione, fino a che si verifica una di queste condizioni:

3.1) tutti i dati sono trasmessi e riscontrati dal TCP peer, ed allora la funzione close() restituisce il controllo al chiamante con risultato 0, passando dallo stato TIME_WAIT.



Anche in questo caso però, anche se sappiamo che il TCP ha ricevuto i dati non abbiamo garanzie che l'application peer riceva i dati. Puo' capitare infatti che dopo che il TCP peer ha spedito l'ACK per i dati ed il FIN, e quindi la close() e' terminata, l'application peer vada in crash e non riesca a leggere dalla coda del TCP peer.



3.2) oppure scade il tempo assegnato di attesa l_linger e la funzione close restituisce -1 mandando un segment RST (reset) all'altro end-system, e non passa dallo stato TIME_WAIT.

funzione **shutdown()** (1)

La funzione shutdown() e' utilizzata per chiudere una connessione in modo differente rispetto alla close().Infatti:

- mentre **la close() decrementa solo il contatore dei processi** che utilizzano quel socket e **quando il contatore e' zero spedisce il segmento FYN**, **la shutdown spedisce subito il segmento di FIN** (anche se il contatore e' maggiore di zero) ovviamente senza passare avanti agli altri dati bufferizzati per la spedizione.
- mentre **la close() chiude entrambe le direzioni della connessione**, e quindi impedisce di usare con quel socket sia primitive di input sia di output (no read no write dopo close()), la shutdown da' la possibilita' di effettuare chiusure asimmetriche di una connessione, specificando quale direzione deve essere interrotta.

int **shutdown** (int sockfd, int howto);

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

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

L'argomento **howto** specifica l'azione che deve essere effettuata sul socket sockfd, ed e' una delle seguenti:

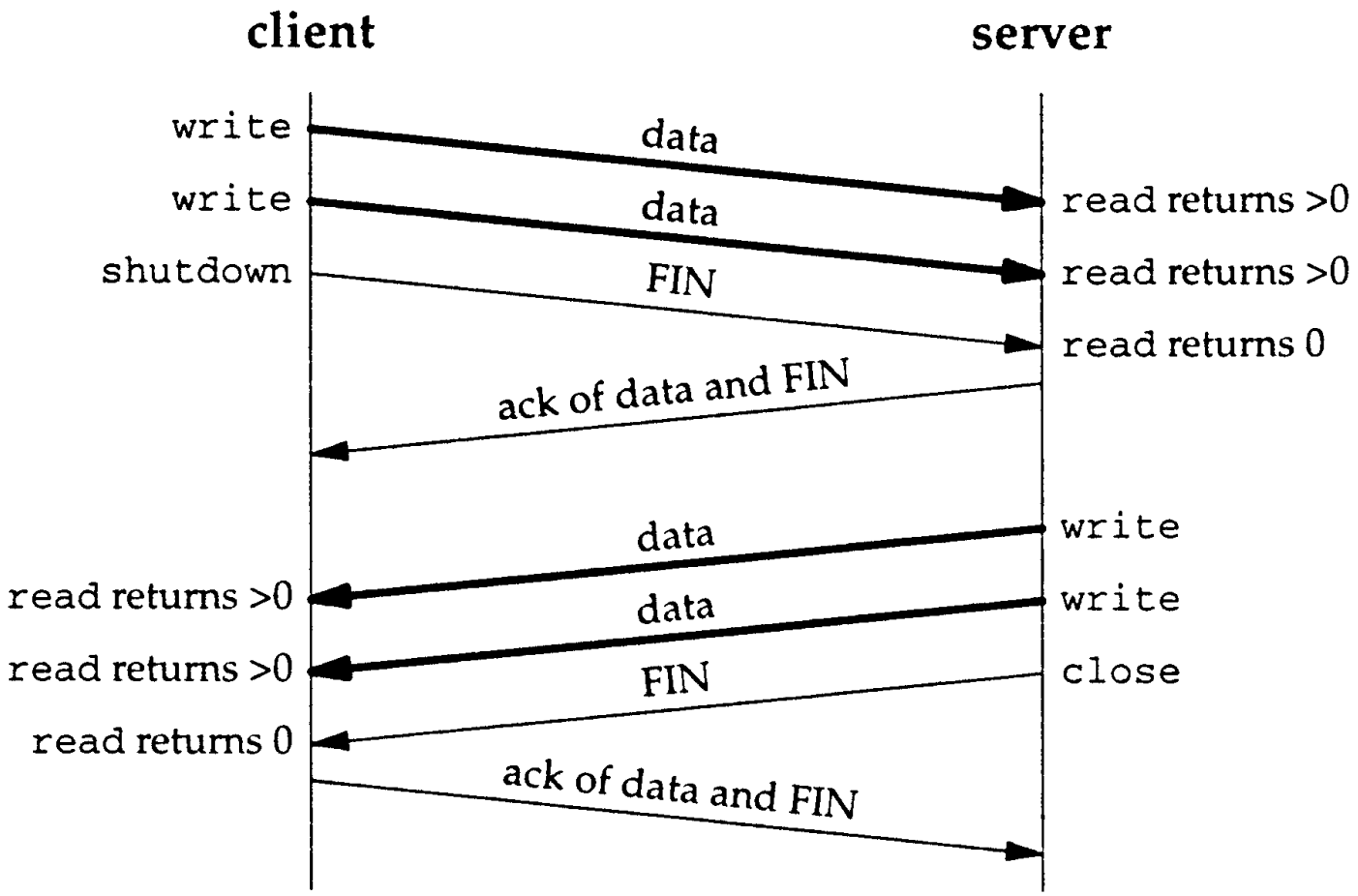
- SHUT_WR l'applicazione chiude il socket sia in scrittura che in lettura,, senza badare al contatore di processi per quel socket. Tutti i dati eventualmente presenti nelle code di output verranno spediti, e poi verra' spedito un SYN segment, per terminare il lato di output della connessione. Rimane possibile effettuare delle read() fino a che l'altro end-system non effettua a suo volta un\a close() che fa inviare il SYN segment verso chi aveva effettuato la chiamata alla shutdown(). Questo tipo di chiusura viene detta **half-close**.

- SHUT_RD l'applicazione chiude il socket in lettura, resta possibile effettuare le write(). Non è piu' possibile effettuare le letture, e tutti i dati eventualmente gia' ricevuti dal TCP e presenti nelle code per l'input vengono scartati. Vengono scartati anche eventuali dati giunti dopo la shutdown().

funzione **shutdown()** (2)

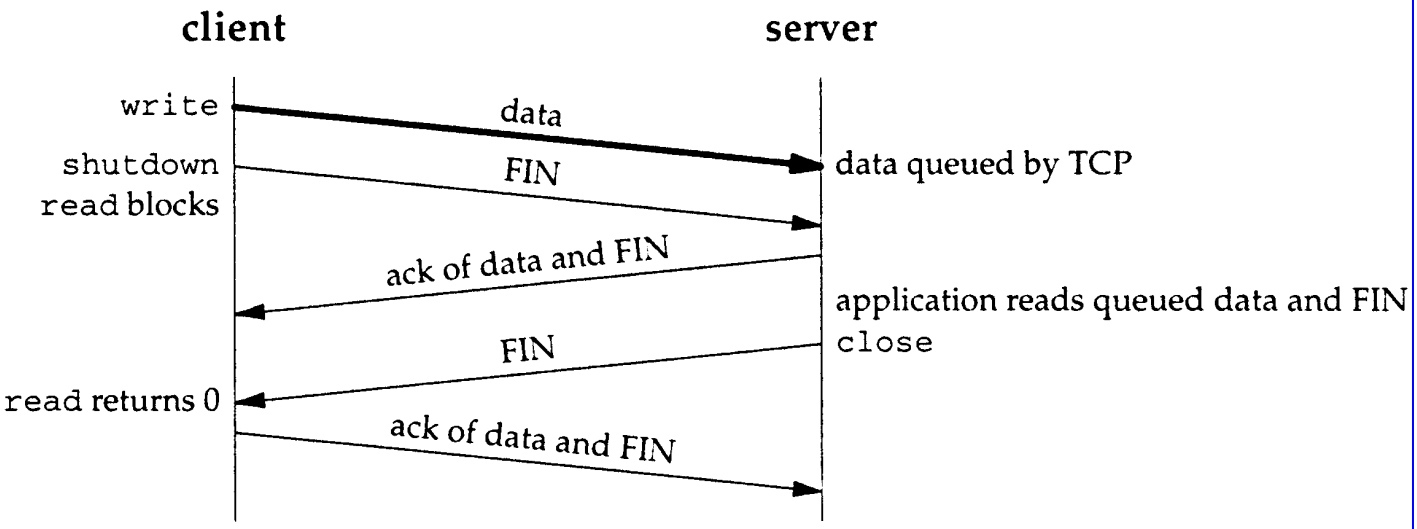
- SHUT_RDWR l'applicazione chiude il socket sia in scrittura che in lettura, come se effettuasse due chiamate alla shutdown, con parametro SHUT_RD e poi con parametro SHUT_WR.

vediamo qui una rappresentazione di una half close (SHUT_WR).



Garanzie di Trasmissione Completata (1)

Garanzia per il Client che il Server ha ricevuto tutti i dati.
Un modo sicuro per sapere se l'applicazione dell'altro end system (non solo il TCP) ha ricevuto i dati è sostituire la chiamata alla close() con una chiamata alla shutdown() usando come secondo argomento la costante SHUT_WR, e procedere poi con una chiamata alla read().



In questo modo la shutdown manda il segmento di FIN, ma lascia aperto il socket in lettura, permettendo di effettuare la chiamata alla read() che altrimenti restituirebbe immediatamente un errore. La read() rimane bloccata fino a che l'application peer termina la lettura di tutti i dati e legge il FIN, quindi effettua la close() che manda il segmento FIN di risposta.
La read() allora riceve l'end-of-file e termina restituendo 0. Si ha allora la garanzia che la applicazione peer ha ricevuto tutti i dati.

Garanzie di Trasmissione Completata (2)

Garanzia per il Client che il Server ha ricevuto tutti i dati.
Un altro modo per garantire che l'application peer ha ricevuto tutti i dati e' usare il cosiddetto **application-level acknowledgment** o **application ACK**.
Il client dopo avere spedito tutti i suoi dati si blocca su una read aggiuntiva, che rappresenta l'attesa per un ACK a livello di applicazione. Il server, dopo avere ricevuto tutti i dati effettua una write di un byte, che rappresenta l'ACK a livello di applicazione. Quando il client ritorna dalla read() effettua la close() che manda il FIN segment. Solo allora l'application peer effettua la close().
In questo modo si ha la garanzia che il processo server ha letto i dati che gli sono stati inviati.

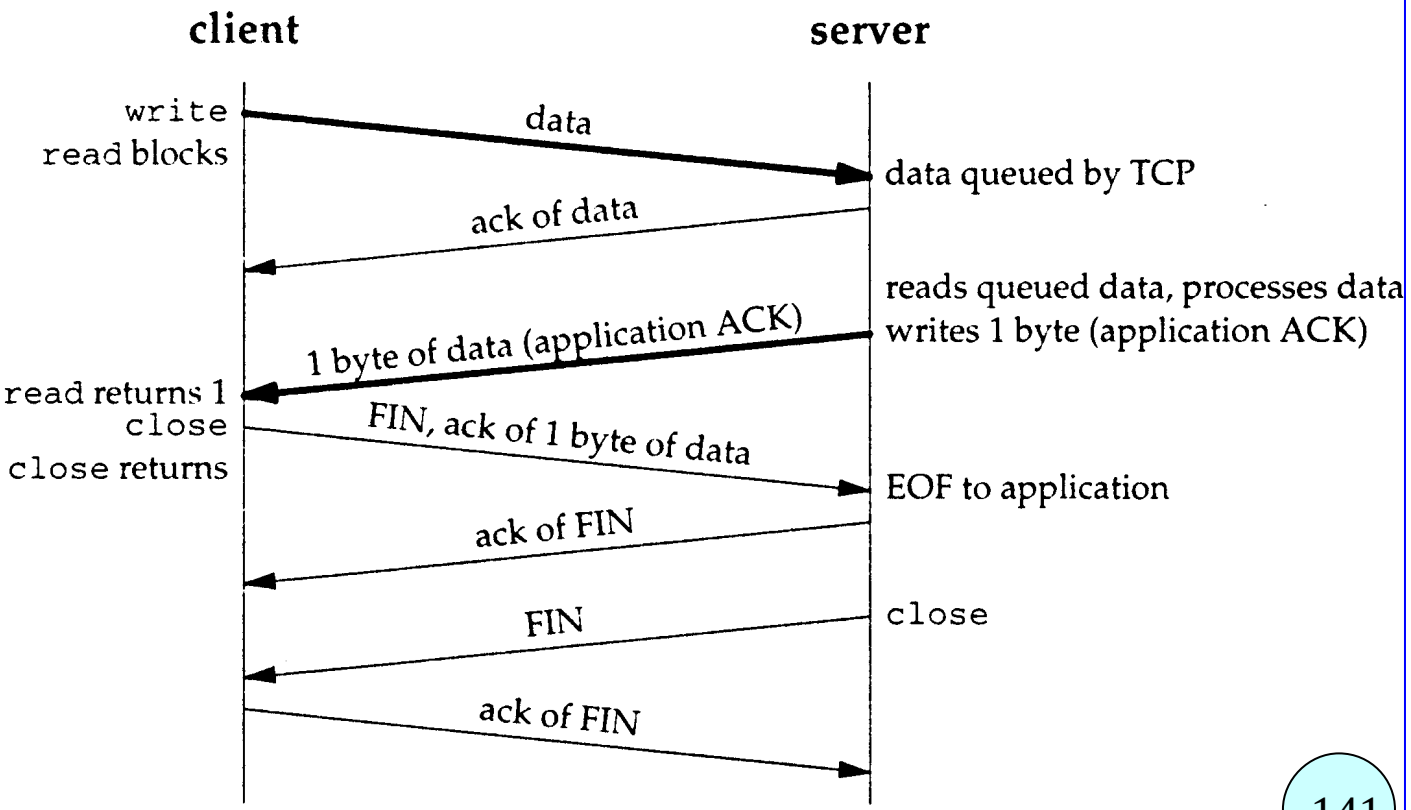


Tabella delle Opzioni Socket per TCP, cioè di livello IPPROTO_TCP, per **getsockopt**() e **setsockopt**()

Per queste opzioni, nelle funzioni getsockopt() e setsockopt() deve essere utilizzato come secondo argomento level=IPPROTO_TCP.

level	*optname*	get	set	Description	Flag	Datatype
IPPROTO_TCP	TCP_KEEPALIVE	•	•	seconds between keepalive probes		int
	TCP_MAXRT	•	•	TCP maximum retransmit time		int
	TCP_MAXSEG	•	•	TCP maximum segment size		int
	TCP_NODELAY	•	•	disable Nagle algorithm	•	int
	TCP_STDURG	•	•	interpretation of urgent pointer	•	int

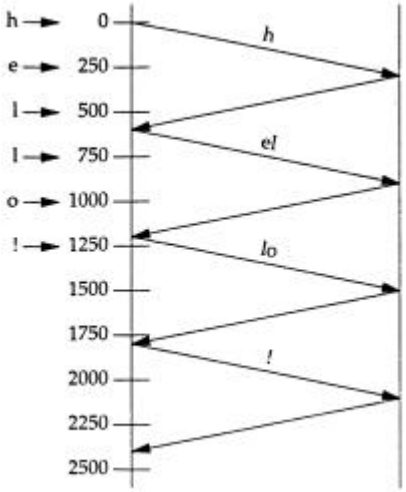
Opzioni Socket per TCP

Queste opzioni si usano specificando il livello **IPPROTO_TCP**.

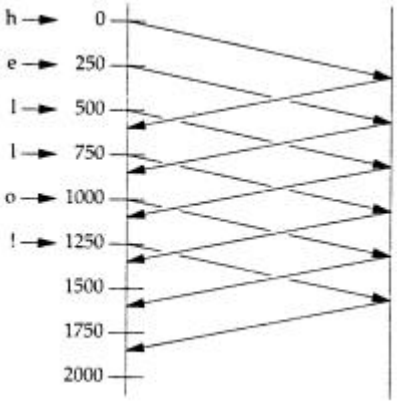
TCP_KEEPALIVE Questa opzione specifica il tempo (in sec.) di inattivita' della connessione prima che venga fatto partire il segmento di keepalive probe. Il valore di default e' di 7200 sec. (2 ore). Questa opzione e' realmente attivata solo quando l'opzione SO_KEEPALIVE e' abilitata.

TCP_MAXSEG Questa opzione restituisce o setta il maximum segment size (MSS) per la connessione TCP.

TCP_NODELAY Per default il TCP abilita il cosiddetto Algoritmo di Nagle (Nagle Algorithm) il cui scopo e' di diminuire il numero di segmenti piccoli trasmessi, come nel caso di client telnet che prevedono l'ACK per ogni singolo carattere battuto a tastiera. Questo algoritmo è legato all'algoritmo dell'ACK ritardato (delayed ACK algorithm) che fa aspettare il TCP per circa 50-200 msec) prima di dare l'ack ad un segmento, sperando di poter accodare l'ACK ad un segmento di dati.
Questi due algoritmi assieme cercano di minimizzare il numero di segmenti trasmessi, ma producono ritardo per applicazioni che scambiano piccoli dati e quasi solo in una direzione.
L'opzione TCP_NODELAY disabilita l'uso di questi algoritmi.



Nagle algorithm Enabled



Nagle algorithm DISABLED
settata TCP_NODELAY opt.

Opzioni Socket per IP: Multicast (1)

Consente di inviare uno stesso datagram UDP a piu' destinatari, i quali devono essersi preventivamente registrati come membri del gruppo di multicast caratterizzato da un certo indirizzo di multicast.

Indirizzi di multicast:

Classe D, 224.0.0.0 - 239.255.255.255

Indirizzi di Multicast Speciali:

224.0.0.1 (all-host group) vi si devono registrare tutti gli host di una sottorete, che implementano il multicast.

224.0.0.2 (all-router group) vi si devono registrare tutti gli host di una sottorete, che implementano il multicast.

Operazioni necessarie per ricevere datagram UDP via multicast:

creazione di un socket UDP

collegamento ad una porta del protocollo UDP (bind)

join ad un indirizzo di multicast (setsockopt).

Operazioni necessarie per smettere di ricevere via multicast:

leave del gruppo multicast (setsockopt).

N.B. Se con la bind non specifico un indirizzo IP, posso ricevere datagram UDP sia originati da multicast che da unicast, per la porta UDP specificata.

N.B. Se con la bind specifico l'indirizzo IP di multicast a cui faro' il join, potro' ricevere solo datagram UDP di multicast per quello indirizzo di multicast.

Operazioni opzionali per spedire datagram UDP via multicast:

Specificare il TTL dei datagram UDP multicast.

Se non si specifica, il default e' 1, non si esce dalla subnet.

Specificare se il mittente deve ricevere copia del datagram (se fa parte del gruppo di multicast.

Opzioni per il Multicast in IPv4

Opzioni di livello IPPROTO_IP

Le opzioni utilizzate per configurare un socket UDP per la trasmissione/ricezione di datagram per un certo indirizzo di multicast, vengono settate mediante la primitiva setsockoption, utilizzando il livello IPPROTO_IP, ed i comandi e le strutture dati specificati in tabella.

opzione	tipo di dato	uso
IP_ADDR_MEMBERSHIP	struct ip_mreq	collega il socket UDP ad uno specifico gruppo di multicast
IP_DROP_MEMBERSHIP	struct ip_mreq	toglie il socket UDP dal gruppo di multicast
IP_MULTICAST_IF	struct in_addr	indica l'interfaccia di rete da usare per spedire i datagram di multicast
IP_MULTICAST_TTL	u_char	specifica il TTL dei datagram di multicast da spedire
IP_MULTICAST_LOOP	u_char	indica se i datagram di multicast spediti devono andare anche al mittente, se questo appartiene al gruppo

esempio: receiver di datagram UDP per gruppo di Multicast

```
struct ip_mreq Mreq;
.....
socketfd = socket (AF_INET, SOCK_DGRAM, 0);
OptVal = 1;
setsockopt (socketfd, SOL_SOCKET, SO_REUSEADDR,
                (char *)&OptVal, sizeof(OptVal) );
Local.sin_family            =  AF_INET;
Local.sin_addr.s_addr    =  htonl(INADDR_ANY);
Local.sin_port              =  htons( 6001); // local_port_number
bind ( socketfd, (struct sockaddr*) &Local, sizeof(Local));

/* join the multicast group. */
Mreq.imr_multiaddr.s_addr = inet_addr("234.5.6.7");
Mreq.imr_interface.s_addr = INADDR_ANY;
setsockopt(socketfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                (char *)&Mreq, sizeof(Mreq)  );

for (j=1;j<=1000;j++) {
    Fromlen=sizeof(struct sockaddr);
    msglen = recvfrom ( socketfd, msg, (int)SIZEBUF, 0,
                            (struct sockaddr*)&From, &Fromlen);
    sprintf(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 per gruppo di Multicast

```
struct ip_mreq Mreq; int ttl, int loopback;
.....
socketfd = socket (AF_INET, SOCK_DGRAM, 0);
OptVal = 1;

/* set TTL to traverse up to multiple routers */
ttl = TTL_VALUE;   // per default 1
setsockopt(socketfd,  IPPROTO_IP,  IP_MULTICAST_TTL,
            (char *)&ttl, sizeof(ttl));

/* join the multicast group, NON NECESSARIO per SPEDIRE */
Mreq.imr_multiaddr.s_addr = inet_addr("234.5.6.7");
Mreq.imr_interface.s_addr = INADDR_ANY;
setsockopt(socketfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
            (char *)&Mreq, sizeof(Mreq)  );

/* disable loopback , NON NECESSARIO se non si fa join */
loopback =0;
setsockopt(socketfd, IPPROTO_IP, IP_MULTICAST_LOOP,
                (char *)&loopback, sizeof(loopback));

for (j=1;j<=1000;j++) {
        To.sin_family    =  AF_INET;
        To.sin_addr.s_addr  =  inet_addr("234.5.6.7" );
        To.sin_port        =  htons( 6001 );  // remote_port_number
        addr_size = sizeof(struct sockaddr_in);
         /* send to the address */
         sendto ( socketfd, msg, strlen(msg) , 0,
                    (struct sockaddr*)&To, addr_size);
}
```

Opzioni Socket tramite **fcntl** (1)

La funzione fcntl significa “file control” e serve ad effettuare alcune operazioni su vari descrittori non solo di socket ma in generale di file. In particolare la funzione fcntl permette di:

• configurare l’I/O di un descrittore come non bloccante o bloccante usando nella fcntl il comando F_SETFL con argomento O_NONBLOCK (O_BLOCK),

• configurare l’I/O di un descrittore come guidato dai signal usando nella fcntl il comando F_SETFL con argomento O_ASYNC, ottenendo che in corrispondenza di ogni modifica della situazione del socket venga scatenato un SIGIO signal.

• settare un proprietario per il descrittore (comando F_SETOWN), ovvero definire qual’e’ il processo che viene avvisato con il signal SIGIO quando il descrittore e’ disponibile all’I/O, ma questo solo se e’ stato settato il socket come guidato dai signal. Si ricorda che un socket appena creato con la socket() non ha proprietario, un connected socket ha lo stesso proprietario del listenung socket

• conoscere il proprietario corrente per il descrittore (F_GETOWN).

```
#include <fcntl.h>
int  fcntl (int fd, int cmd, ...  /* int arg */   );
```

restituisce -1 in caso di errore,
un altro valore dipendente dal comando in caso tutto OK.

Le proprieta’ di un descrittore di file (e di socket) sono definite mediante alcuni flags, che la funzione fcntl permette di leggere (usando come secondo argomento cmd=F_GETFL) e di settare (usando come secondo argomento cmd=F_SETFL).

I due flags che ci interessano per i socket sono:

O_NONBLOCK	non blocking I/O
O_ASYNC	signal-driven I/O notification

Opzioni Socket tramite **fcntl** (2)

In generale la procedura corretta per settare un flags consiste nel leggere i flags del descrittore mediante il comando F_GETFL, modificare solo il flag che interessa con un OR logico, settare i nuovi flags per il socket mediante il comando F_SERFL.

Nel seguente esempio viene settato il flag O_NONBLOCK, quindi il socket diventa "non bloccante".

Procedura corretta

```
int flags, sockfd;
sockfd = socket(....);
if (  (flags=fcntl(sockfd,F_GETFL,0)) <0 )
   exit(1);
flags  |= O_NONBLOCK;
if (  fcntl(sockfd,F_SETFL,flags) <0 )
   exit(1);
```

Procedura errata, vengono azzerati tutti gli altri flags.

```
int sockfd;
sockfd = socket(....);
if ( fcntl(sockfd,F_SETFL, O_NONBLOCK )<0)
 exit(1);
```

Infine in questo esempio viene resettato il flag O_NONBLOCK, quindi il socket diventa "bloccante".

```
int flags, sockfd;
sockfd = socket(....);
if (  (flags=fcntl(sockfd,F_GETFL,0)) <0 )
   exit(1);
flags &= ~O_NONBLOCK;
if (  fcntl(sockfd,F_SETFL,flags) <0 )
   exit(1);
```

Opzioni Socket tramite **fcntl** (3)

Se per un socket viene settato il flags O_ASYNC, quando il socket diventa disponibile all'I/O, o e' in errore, viene lanciato un signal SIGIO al processo proprietario del socket, o ai processi del gruppo proprietario del socket, se il proprietario esiste.

Per settare il proprietario si usa la fcntl con secondo argomento F_SETOWN passando come terzo parametro o il numero positivo pid del processo proprietario, o il numero negativo ottenuto cambiando di segno il numero di gruppo.

Per conoscere il proprietario si usa la fcntl con secondo argomento F_GETOWN. La funzione restituisce un intero positivo (il pid del proprietario) o un numero negativo diverso da -1, che e' l'identificatore del gruppo cambiato di segno.

settare il proprietario di un socket

```
int sockfd,pid;
sockfd = socket(....);
if (  fcntl(sockfd,O_SETOWN,  pid ) <0 )
   exit();
```

settare il gruppo di un socket

```
int sockfd, gid;
sockfd = socket(....);
if (  fcntl(sockfd,O_SETOWN ,  -gid ) <0 )
   exit();
```

I/O Non Bloccante (1)

Per default un socket e' bloccante. Cio' significa che quando il socket deve effettuare un'operazione di I/O, se questa non puo' essere terminata immediatamente il processo si mette in attesa aspettando la fine dell'operazione. Se invece, mediante la fcntl vista prima, il socket viene settato Nonblocking il comportamento delle funzioni cambia:

Operazioni di input (read, recvfrom). Con un socket Non Bloccante, se l'operazione di input non puo' essere terminata (cioe' non c'e' nemmeno un byte per il socket TCP o non c'e' nemmeno un datagram per il socket UDP) la funzione ritorna immediatamente al chiamante restituendo il codice d'errore EWOULDBLOCK.

Operazioni di output (write, sendto).

• Con un socket **Non Bloccante di tipo TCP**, se l'operazione di scrittura non puo' essere effettuata nemmeno in parte perche' manca spazio nei buffer del TCP in cui andare a scrivere i dati, la funzione ritorna immediatamente al chiamante restituendo il codice d'errore EWOULDBLOCK. Se invece e' rimasto spazio, viene effettuata una scrittura di una porzione di dati minore o uguale alla dimensione del buffer libero, e la write restituisce il numero di byte scritti.

• Con un socket **Non Bloccante di tipo UDP** invece il problema non sussiste perche' nessun datagram viene mantenuto in un buffer perche' non c'e' ritrasmissione, quindi il datagram UDP viene immediatamente spedito e vada come vada. Quindi una primitiva di output di un socket UDP non blocca mai il processo che l'effettua.

Operazioni di Accettazione Richiesta Connessione (accept). Con un socket Non Bloccante, se l'operazione di accept non puo' restituire immediatamente una nuova connessione perche' la coda delle connessioni accettate e' vuota, la funzione ritorna immediatamente al chiamante restituendo il codice d'errore EWOULDBLOCK.

I/O Non Bloccante (2)

Operazioni di Richiesta Connessione (connect).

Con **un socket UDP** la funzione connect non e' realmente bloccante, perche' deve solo scrivere nel kernel l'indirizzo IP e il numero di porta dell'altro end-system, quindi anche nel caso di socket Non Bloccante la connect per l'UDP effettua sempre l'operazione interamente e restituisce il controllo al chiamante.

Invece **per il TCP** la chiamata del client alla connect() di un socket bloccante deve aspettare che, all'interno del three way handshake, il client abbia ricevuto l'ACK per il SYN segment, sia stato ricevuto, e quindi puo' veramente bloccare il processo.

Se il socket TCP e' di tipo Non Bloccante, la chiamata del client alla connect() fa iniziare il procedimento di inizio connessione, cioe' fa spedire il primo SYN segment, ma se la connessione non puo' immediatamente essere instaurata la connect() termina con un codice d'errore **EINPROGRESS** ad indicare che l'operazione continua a livello inferiore.
Con una select() sara' possibile accorgersi di quando il socket impegnato nella richiesta di connessione ha stabilitpo la connessione, e quindi puo' essere ripetuta la connect().
Si noti che puo' capitare che la connect() restituisca OK perche' e ' riuscita ad avere immediatamente risposta ed ha instaurato la connessione richiesta.

Connect Non Bloccante (1)

```
socketfd = socket(AF_INET, SOCK_STREAM, 0);
/* SETTO IL SOCKET COME NON BLOCCANTE  */
set_socket_non_blocking(socketfd); /* ora socket e' non bloccante */
    ... qui va messo il necessario per la bind  .....
bind(socketfd, (struct sockaddr*) &Local, sizeof(Local));
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);
/* connection request, NON BLOCCANTE */
ris = connect(socketfd, (struct sockaddr*) &Serv, sizeof(Serv));
if (ris != SOCKET_ERROR)  {    /* connessione riuscita subito */
      set_socket_blocking(socketfd); /* ora socket e' bloccante */
        .....
}
else {    /*  (ris == SOCKET_ERROR) */
      if(errno!=EINPROGRESS)  exit(1); /* conness. non terminata */
      FD_ZERO(&fdr); FD_SET(socketfd,&fdr);
      fdw=fdr;
        .....
      ris=select(socketfd+1,&fdr,&fdw,NULL,NULL);
      if ((FD_ISSET(socketfd,&fdr))||(FD_ISSET(socketfd,&fdw))) {
            ris=connect(socketfd,(struct sockaddr*) &Serv, sizeof(Serv));
            if (ris == SOCKET_ERROR)  {
                  if(errno==EISCONN) {
                        printf ("connessione gia' esistente, OK\n");
                        set_socket_blocking(socketfd); /* socket e‘ bloc*/
                  }
                  else  exit(1); /* connessione NON riuscita */
            }
      }
}
```

Connect Non Bloccante (2)

```
/* SETTO IL SOCKET COME NON BLOCCANTE  */
int set_non_blocking(int socketfd) {
    int flags;
    if ( (flags=fcntl(socketfd,F_GETFL,0)) <0 ) return(0);  /* errore */
    flags |= O_NONBLOCK;
    if ( fcntl(socketfd,F_SETFL,flags) <0 ) return(0);  /* errore */
    return(1);
}


/* SETTO IL SOCKET COME BLOCCANTE  */
int set_blocking(int socketfd) {
    int flags;
    if ( (flags=fcntl(socketfd,F_GETFL,0)) <0 ) return(0);  /* errore */
    flags &= (~O_NONBLOCK);
    if ( fcntl(socketfd,F_SETFL,flags) <0 ) return(0);  /* errore */
 return(1);
}
```

N.B.
per l’esempio completo vedere nella home page