[successivo] [precedente] [inizio] [fine] [indice generale]

Capitolo 3. Le funzioni dei socket di rete

Prima di illustrare le funzioni di gestione dei socket definiamo il *socket pair* la cui importanza è fondamentale in tale contesto; con tale termine si intende la combinazione di quattro valori che identificano i due estremi della comunicazione: *IP_locale:porta_locale, IP_remoto:porta_remota*.

A proposito delle porte è importante anche ricordare quelle identificate da un valore minore di 1024 possono essere usate solo da programmi che siano eseguiti con i privilegi dell'utente *root*.

La sequenza di operazioni da svolgere per gestire un socket TCP è:

1. creazione del socket;
2. assegnazione dell'indirizzo;
3. connessione o attesa di connessione;
4. invio o ricezione dei dati;
5. chiusura del socket.

Di queste la creazione è già stata esaminata in 2.2.

3.1 Assegnazione dell'indirizzo a un socket

La funzione per l'assegnazione di un indirizzo ad un socket è *bind()* con la quale si si assegna un indirizzo locale ad un socket (quindi la prima metà di un *socket pair*).

La usa solitamente un programma servente per stabilire da quale «IP:porta» si metterà in ascolto.

Un cliente invece di solito non la usa in quanto il suo indirizzo per una connessione viene scelto automaticamente dal *kernel* (almeno per quanto riguarda la porta, visto che l'IP sarà quello dell'interfaccia di rete usata).

Se accade che un servente non specifichi il suo indirizzo locale, il *kernel* lo determinerà in base all'indirizzo di destinazione specificato dal segmento *SYN* del cliente (cioè il primo segmento inviato durante il processo di attivazione della connessione).

La funzione ha il seguente prototipo:

```
int bind(int sd, const struct sockaddr *serv_ind, socklen_t indlen)

//sd è l'identificativo (o file descriptor) del socket ottenuto
//     dalla creazione con la funzione socket();
//serv_ind è l'indirizzo;
//indlen è la lunghezza dell'indirizzo.
```

Ritorna 0 in caso di successo e -1 se c'è errore, nel qual caso la variabile *errno* viene valorizzata nel seguente modo:

- EBADF e ENOTSOCK: *sd* non è valido;
- EINVAL: il socket ha già un indirizzo assegnato;
- EACCES: si sta cercando di usare una porta non avendo sufficienti privilegi;
- EADDRNOTAVAIL: il tipo di indirizzo indicato non è disponibile;
- EADDRINUSE: l'indirizzo è già usato da un altro socket.

Se si devono indicare indirizzi IPv4 particolari (locale, broadcast) si possono usare le seguenti costanti tutte valorizzate in formato macchina (e quindi da convertire):

- INADDR_ANY: indirizzo generico (0.0.0.0);
- INADDR_BROADCAST: indirizzo di broadcast;
- INADDR_LOOPBACK: indirizzo di loopback (127.0.0.1);
- INADDR_NONE: indirizzo errato.

Per avere le stessa possibilità in IPv6 sono definite in <netinet/in.h> le variabili esterne *in6addr_any*, e *in6addr_loopback* inizializzate dal sistema rispettivamnete con i valori IN6ADRR_ANY_INIT e IN6ADDR_LOOPBACK_INIT.

3.2 Connessione

La connessione di un cliente TCP ad un servente TCP si effettua con la funzione *connect()*, usata dal cliente e il cui prototipo è:

```
int connect(int sd, const struct sockaddr *serv_ind, socklen_t indlen)

//sd è l'identificativo del socket;
//serv_ind è l'indirizzo del cliente;
//indlen  è la lunghezza dell'indirizzo.
```

Ritorna 0 in caso di successo e -1 se c'è errore, nel qual caso *errno* assume i valori (i più significativi):

- ECONNREFUSED: nessun processo è in ascolto all'indirizzo remoto;
- ETIMEDOUT: scaduto il timeout durante il tentativo di connessione;
- ENETUNREACH: rete non raggiungibile;
- EAFNOSUPPORT: indirizzo specificato con famiglia di indirizzi non corretta;
- EACCES, EPERM: tentativo di connessione a indirizzo broadcast ma socket non abilitato al broadcast;
- EINPROGRESS: socket non bloccante ma la connessione non può essere conclusa immediatamente;
- EALREADY: socket non bloccante e un tentativo precedente di connessione non si è ancora concluso.

La funzione *connect()* per il TCP attiva il meccanismo di attivazione e ritorna a connessione stabilita o se c'è un errore.

Fra le situazioni di errore, quelle dovute alla rete sono:

- il cliente non riceve risposta al SYN: (si tratta dell'errore ETIMEDOUT); in GNU/Linux ciò avviene normalmente dopo un tempo di 180 secondi in quanto il sistema invia nuovi SYN ad intervalli di 30 secondi per un massimo di 5 tentativi; questo valore di tentativi può comunque essere cambiato o usando la funzione *sysctl()* (vedere il manuale in linea) o scrivendo il valore in `'/proc /sys/net/ipv4/tcp_syn_retries'`.
- il cliente riceve come risposta al SYN un RST (è l'errore ECONNREFUSED) perché il SYN era per una porta che non ha nessun processo in ascolto, oppure perché il TCP ha abortito la connessione in corso, oppure perché il servente ha ricevuti un segmento per una connessione inesistente.
- la risposta al SYN è un messaggio ICMP di destinazione non raggiungibile (è l'errore ENETUNREACH); la condizione errata può essere transitora e superata da tentativi successivi fino allo scadere del timeout come illustrato sopra.

In caso di esito positivo della funzione *connect()*, la connessione è completata e i processi possono comunicare.

È importante ribadire che il cliente non deve preoccuparsi dell'altra metà del *socket pair*, cioè il proprio «IP:porta», in quanto viene assegnata automaticamente dal *kernel*.

3.3 Attesa di connessione

In un servente TCP, quindi orientato alla connessione, è necessario indicare che il processo è disposto a ricevere le connessioni e poi mettersi in attesa che arrivino le relative richieste da parte dei clienti.

Le due operazioni devono essere svolte dopo la *bind()* e sono realizzate grazie alle funzioni *listen()* e *accept()*.

La funzione *listen()* ha il seguente prototipo:

```
int listen(int sd, int backlog)

//pone il socket sd in attesa di una connessione;
//backlog è il numero massimo di connessioni accettate.
```

Ritorna 0 in caso di successo e -1 se c'è errore con *errno* che in tal caso assume i valori:

- EBADF o ENOTSOCK: socket non valido;
- EOPNOTSUPP: il socket non supporta questa funzione.

Si può applicare solo a socket di tipo SOCK_STREAM o SOCK_SEQPACKET e pone il socket in modalità passiva (in ascolto) predisponendo una coda per le connessioni in arrivo di lunghezza pari al valore indicato nel parametro *backlog*.

Se tale valore viene superato, al cliente che ha inviato la richiesta dovrebbe essere risposto con un errore ECONNREFUSED, ma siccome il TCP prevede la ritrasmissione, la richiesta viene semplicemente ignorata in modo che la connessione possa essere ritentata.

Riguardo alla lunghezza della coda delle connessioni si deve osservare che essa riguarda solo le connessioni completate (cioè quelle per cui il processo di attivazione è concluso); in effetti per ogni socket in ascolto ci sono in coda anche quelle non completate (l'attivazione è ancora in corso) e nei vecchi *kernel* (fino al 2.2) il parametro *backlog* considerava anche queste.

Il cambiamento ha lo scopo di evitare gli attacchi *syn flood* che consistono nell'invio da parte di un cliente di moltissime richieste di connessione (segmenti SYN), lasciate volutamente incomplete grazie al mancato invio del segmento ACK in risposta al segmento SYN - ACK del servente, fino a saturare la coda delle connessioni di quest'ultimo.

La funzione *accept()* ha il seguente prototipo:

```
int accept(int sd, struct sockaddr *ind, socklen_t *indlen)

//accetta una connessione sul socket sd;
//ind e indlen sono l'indirizzo, e relativa lunghezza, del cliente
//      che ha inviato la richiesta di connessione
```

Ritorna un numero di socket positivo in caso di successo oppure -1 se c'è errore; in tal caso *errno* può assumere gli stessi valori visti nel caso di *listen()* e anche:

- EPERM: un firewall non consente la connessione;

- EAGAIN o EWOULDBLOCK: socket non bloccante e non ci sono connessioni da accettare;
- ENOBUFS e ENOMEM: memoria limitata dai limiti sui buffer dei socket.

La funzione può essere usata solo con socket che supportino la connessione (cioè di tipo SOCK_STREAM, SOCK_SEQPACKET o SOCK_RDM) e di solito viene invocata da un processo servente per gestire la connessione dopo la conclusione del meccanismo di attivazione della stessa.

L'effetto consiste nella creazione di un nuovo socket, detto «socket connesso», il cui descrittore è quello ritornato dalla funzione, che ha le stesse caratteristiche del socket *sd* e sul quale avviene la comunicazione; il socket originale resta invece nello stato di ascolto.

Se non ci sono connessioni completate in coda, il processo che ha chiamato la funzione può:

- essere messo in attesa, se, come avviene normalmente, il socket è bloccante;
- continuare, se il socket è non bloccante; in tal caso, come detto, la funzione ritorna -1 con errore EAGAIN o EWOULDBLOCK.

3.4 Invio e ricezione dati

Per l'invio e la ricezione dei dati si possono usare le stesse funzioni usate per la scrittura e lettura dei file a basso livello, *write()* e *read()*:

```
ssize_t write(int sd, void *buf, size_t cont)
ssize_t read(int sd, void *buf, size_t cont)

//sd è il socket usato;
//buf l'area di transito dei dati;
//cont la quantità di byte da leggere o scrivere.
```

Le due funzioni ritornano la quantità di byte effettivamente scritti o letti oppure -1 in caso di errore, nel qual caso *errno* può valere (tra l'altro):

- EINTR: la funzione è stata interrotta da un segnale;
- EAGAIN: non ci sono dati da leggere o scrivere e il socket è non bloccante.

Con i socket avviene molto più frequentemente che con i file che il numero di byte letti o scritti non coincida con quanto indicato nel parametro *cont*.

Per questo motivo è opportuno definire delle funzioni di lettura e scrittura personalizzate che usino rispettivamente la funzione *read()* e *write()* in modo iterativo, fino al raggiungimento del numero di byte richiesti in lettura o scrittura.

All'interno di queste funzioni personalizzate si deve avere l'accortezza di testare eventuali errori: se si tratta di EINTR il ciclo deve essere continuato (non è un vero e proprio errore sul socket), altrimenti interrotto.

Nel caso della lettura, se il numero di byte letti è zero (situazione simile all'EOF per i file), significa che il socket è stato chiuso dal processo all'altro estremo della comunicazione e quindi non si deve continuare al leggere.

Esistono anche altre due funzioni per scrivere o leggere i dati, la *send()* e la *recv()* che hanno i prototipi:

```
int recv(int sd, void *buf, int lun, int opzioni)

//riceve dati dal socket sd;
//buf area di transito dei dati;
//lun dimensione dati da ricevere;
//opzioni può essere impostato a 0.

int send(int sd, void *buf, int lun, int opzioni)

//invia dati sul socket sd;
//buf area di transito dei dati;
//lun dimensione dati da inviare;
//opzioni può essere impostato a 0.
```

Le due funzioni ritornano -1 in caso di errore oppure il numero di byte effettivamente scritti o letti.

3.5 Invio e ricezione dati con socket UDP

Il protocollo UDP non supporta le connessioni e non è affidabile; i dati vengono inviati in forma di pacchetti chiamati anche «datagrammi», senza alcuna assicurazione circa l'effettiva ricezione o l'arrivo nel giusto ordine.

Il vantaggio rispetto al TCP risiede nella velocità e fa preferire il trasporto UDP nei casi in cui questa caratteristica è fondamnetale come nel trasferimento di dati multimediali.

Un altro caso adatto all'uso di UDP è quello in cui la comunicazione consiste in un semplice processo di interrogazione/risposta con pochissimi dati da trasferire; l'esempio tipico è il servizio DNS che infatti si appoggia su UDP.

I socket UDP non supportano la comunicazione di tipo *stream* tipica del TCP, in cui si ha a disposizione un flusso continuo di dati che è possibile leggere un po' alla volta, ma piuttosto una comunicazione di tipo *datagram*, in cui i dati arrivano in singoli blocchi da leggere integralmente.

Quindi i socket UDP devono essere aperti con la funzione *socket* utilizzando per lo stile di comunicazione il valore SOCK_DGRAM; inoltre, non esistendo il concetto di connessione non è ovviamente necessario alcun meccanismo di attivazione e non servono le funzioni *connect()*, *listen()* e *accept()*.

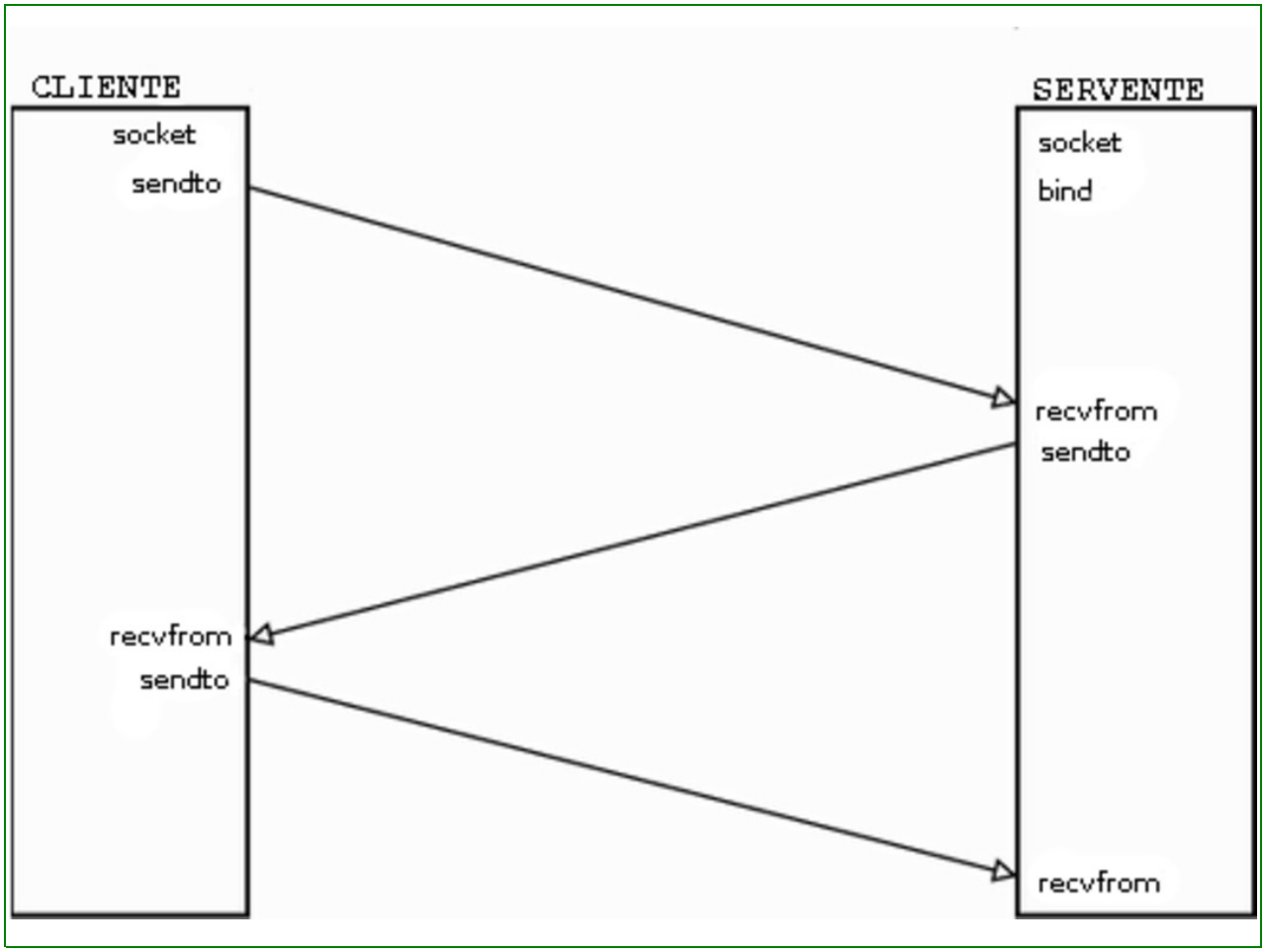
La funzione *bind()* invece serve ancora:

- ad entrambi i processi che comunicano, se si tratta di una comunicazione paritetica;

- solo al servente se c'è ancora la presenza di un cliente e un servente.

Lo scambio di dati avviene in modo molto semplice come schematizzato in 3.1 dove si ipotizza una comunicazione fra cliente e servente.

Figura 3.1



Il *kernel* si limita a ricevere i pacchetti ed inviarli al processo in ascolto sulla porta cui essi sono destinati, oppure a scartarli inviando un messaggio ICMP «port unreachable» se non c'è alcun processo in ascolto.

La ricezione dei dati avviene attraverso la funzione recvfrom(), l'invio con la funzione sendto() che sono comunque utilizzabili anche con altri tipi di socket:

```
ssize_t sendto(int sd, const void *buf, size_t len, int flags,\
  \const struct sockaddr *to, socklen_t tolen)

//trasmette un messaggio al socket sd;
//buf e len hanno lo stesso significato visto nella write();
//flags ha un ruolo che non viene qui approfondito e viene sempre posta a 0;
//to è l'indirizzo della destinazione;
//tolenz è la lunghezza dell'indirizzo di destinazione.
```

Ritorna il numero di byte inviati in caso di successo e -1 se c'è errore, nel qual caso *errno* può valere (tra l'altro):

- EAGAIN: socket non bloccante, ma l'operazione richiede il blocco della funzione;
- ECONNRESET: l'altro nodo della comunicazione ha resettato la connessione;
- EMSGSIZE: il socket richiede l'invio dei dati in un blocco unico, ma la dimensione del messaggio è eccessiva;
- ENOTCONN: socket non connesso e non si è specificata una destinazione;
- EPIPE: estremo locale della connessione chiuso.

A differenza di quanto accade con la *write()* il numero di byte inviati deve sempre corrispondere a quanto specificato in *len* perché i dati non possono essere spezzati in invii successivi; se non c'è spazio nel buffer di uscita la funzione si blocca (se il socket è bloccante); se invece non è possibile inviare i dati dentro un unico pacchetto (perché eccede le dimensioni massime del protocollo IP sottostante) essa fallisce con l'errore di EMSGSIZE.

Il prototipo di recvfrom() è:

```
ssize_t recvfrom(int sd, const void *buf, size_t len, int flags,\
  \const struct sockaddr *from, socklen_t *fromlen)

//riceve un messaggio dal socket sd;
//buf e len hanno lo stesso significato visto nella read();
//flags ha un ruolo che non viene qui approfondito e viene sempre posta a 0;
//from è l'indirizzo di origine;
//fromlen è la lunghezza dell'indirizzo di origine.
```

Ritorna il numero di byte ricevuti in caso di successo e -1 se c'è errore, nel qual caso *errno* può valere (tra l'altro):

- EAGAIN: socket non bloccante, ma l'operazione richiede il blocco della funzione, oppure si è impostato un timeout in ricezione che è scaduto;
- ENOTCONN: il socket è connesso, ma non si è eseguita la connessione.

Se non ci sono dati disponibili la funzione si blocca (se il socket è bloccante); se *len* eccede la dimensione del pacchetto la funzione legge comunque i dati disponibili, e il suo valore di ritorno è il numero di byte letti.

A seconda del tipo di socket, *datagram* o *stream*, gli eventuali byte in eccesso non letti possono rispettivamente andare persi o restare disponibili per una lettura successiva.

Se il processo ricevente non è interessato a conoscere i dati relativi all'indirizzo di origine gli argomenti *from* e *fromlen* devono essere inizializzati a NULL.

Con le funzioni *sendto()* e *recvfrom()* è possibile inviare o ricevere nessun byte; nel caso dell'invio è semplicemente un pacchetto vuoto che contiene solo le intestazioni IP e UDP (e *len* deve essere 0); nel caso della ricezione, il valore di ritorno di 0 byte, non deve essere interpretato come chiusura della connessione o fine della comunicazione.

3.6 Socket UDP connessi

La comunicazione basata su socket UDP, è più semplice da gestire ma è soggetta da alcuni problemi abbastanza fastidiosi.

Supponiamo ad esempio di eseguire un processo che invia dati con *sento()* e riceve risposte con *recvfrom()* da un servente UDP; se qualche pacchetto di dati inviati o di risposta si perde o se il servente non è in ascolto, il nostro processo si blocca inesorabilmente eseguendo la funzione *recvfrom()*.

Infatti non essendo prevista alcuna connessione non è possibile neanche avere riscontri circa il buon esito dell'invio di un pacchetto.

In verità la condizione di servente non in ascolto viene rilevata con messaggi ICMP del tipo «destination unreachable» che però sono asincroni rispetto all'esecuzione della funzione *sendto()* che quindi non può rilevarli.

Il problema può essere almeno in parte risolto con l'uso della funzione *connect()*, tipica del TCP, anche da parte di un cliente UDP.

Quando si invoca una *connect()* su un socket UDP l'indirizzo passato come parametro viene registrato come indirizzo di destinazione del socket e, a differenza che in TCP, non viene inviato alcun pacchetto.

Dopo la *connect()* ogni invio di dati su quel socket viene diretto automaticamente a quell'indirizzo e gli argomenti *to* e *tolen* non devono più essere valorizzati.

Anche il comportamento in ricezione cambia; vengono recapitati ad un socket connesso solo i pacchetti con un indirizzo sorgente corrispondente a quello indicato nella connessione.

Il vantaggio è però nel fatto che, per le funzioni usate su un socket UDP connesso, gli errori dovuti a «destinazione non in ascolto» non bloccano il processo.

Infatti tale condizione viene ora rilevata, anche se non al momento della connect(), come avviene in TCP, (visto che in UDP essa non comporta alcun trasferimento di pacchetti), bensì al momento in cui la stazione tenta di scambiare dei dati con la destinazione.

L'altro problema evidenziato, cioè il blocco del cliente sulla *recvfrom()* causato dalla perdita di pacchetti inviati o di pacchetti di risposta, non viene invece risolto neanche con i socket UDP connessi; il processo cliente deve quindi gestire un *timeout* o usare un socket non bloccante.

3.7 Chiusura di un socket

Un socket vine chiuso con la funzione close() che è molto semplice:

```
int close (int sd)

// chiude il socket sd.
```

Ritorna 0 in caso di successo o -1 se c'è un errore.

Il suo scopo è quello di rendere inutilizzabile un socket presso uno dei due estremi della comunicazione; la funzione deve quindi essere eseguita da entrambi i processi che stanno comunicando.

La chiusura avviene mediante l'invio di un segmento FIN come illustrato nel paragrafo 1.2.3.

I dati eventualmente in coda per essere spediti vengono comunque inviati prima che la chiusura sia effettuata; inoltre ogni socket ha un contatore di riferimenti perché potrebbe essere usato da altri processi (ad esempio dei processi figli) e quindi la chiusura viene innescata solo quando tale contatore si annulla.

Se solo uno dei due estremi della comunicazione esegue la chiusura l'altro nodo può continuare a inviare i dati che però non possono essere letti dal primo nodo che ha il socket chiuso.

Per gestire in modo effciente anche queste situazioni di *half-close* si può utilizzare la funzione shutdown() che ha il seguente

prototipo:

```
int shutdown(int sd, int val)

//chiude un lato della connessione del socket sd;
//val indica la modalità di chiusura.
```

Ritorna zero in caso di successo e -1 se c'è un errore.

Il secondo argomento può valere:

- SHUT_RD: chiude il lato in lettura del socket, i dati inviati dall'altro estremo vengono scartati, ma il processo può continuare a usare il socket per inviare dati;
- SHUT_WR: chiude il lato in scrittura del socket, i dati in attesa di invio sono spediti prima della chiusura; il processo può continuare a usare il socket per ricevere dati;
- SHUT_RDWR: chiude entrambi i lati del socket.

La modalità SHUT_RDWR può sembrare inutile perché pare rendere la *shutdown()* del tutto equivalente alla *close()*; invece c'è un'importante differenza: con essa infatti si chiude il socket immediatamente, anche se ci sono altri riferimenti attivi su di esso.

3.8 Altre funzioni per i socket

Esistono diverse altre funzioni di varia utilità per la gestione dei socket; in questo paragrafo vengono illustrate le più importanti.

3.8.1 Impostazione e lettura delle opzioni di un socket

Le opzioni di un socket possono essere impostate con la funzione *setsockopt()* che ha il seguente prototipo:

```
int setsockopt(int sd, int livello, int nomeopz, const void *valopz,\
  \socklen_t lunopz)

//imposta le opzioni del socket sd;
//livello è il protocollo su cui si vuole intervenire;
//nomeopz è l'opzione da impostare;
//valopz è il puntatore ai valori da impostare;
//lunopz è la lunghezza di valopz.
```

Ritorna 0 in caso di successo e -1 se c'è errore, nel qual caso *errno* può valere:

- EBADF o ENOTSOCK: socket non valido;
- EFAULT: indirizzo *valopz* non valido;
- EINVAL: valore di *lunopz* non valido;
- ENOPROTOOPT: opzione scelta non esiste per il livello indicato.

I possibili livelli sono:

- SOL_SOCKET: opzioni generiche dei socket;
- SOL_IP: opzioni per socket che usano IPv4;
- SOL_TCP: opzioni per socket che usano TCP;
- SOL_IPV6: opzioni per socket che usano IPv6;
- SOL_ICMPV6: opzioni per socket che usano ICMPv.

Il parametro *valopz* è solitamente un intero oppure NULL se non si vuole impostare una certa opzione.

Le opzioni di un socket possono essere lette con la funzione *getsockopt()* che ha il seguente prototipo:

```
int getsockopt(int sd, int livello, int nomeopz, void *valopz, socklen_t *lunopz)

//legge le opzioni del socket sd;
//il significato dei parametri è lo stesso visto in setsockopt().
```

Ritorna 0 in caso di successo e -1 se c'è errore con i codici EBADF, ENOTSOCK, EFAULT, ENOPROTOOPT visti in precedenza.

Ovviamente in questa funzione il parametro *valopz* serve a ricevere il valore letto per l'opzione impostata in *nomeopz*.

Le opzioni da impostare o leggere sono molto numerose e qui l'argomento non viene approfondito; per le opzioni generiche si può consultare il manuale in linea di *socket*.

Ecco alcune opzioni generiche interessanti:

- SO_BINDTODEVICE: utilizzabile da entrambe le funzioni, il suo valore è una stringa (ad esempio *eth0*) e permette di associare il socket a una particolare interfaccia di rete; se la stringa è nulla e *lunopz* è zero si rimuove un precedente collegamento;
- SO_TYPE: utilizzabile solo in lettura, permette di leggere il tipo di socket su cui si opera; *valopz* è un numero in cui viene restituito il valore che identifica lo stile di comunicazione (ad esempio SOCK_DGRAM);
- SO_ACCEPTCONN: utilizzabile solo in lettura, serve a verificare se il socket su cui opera è in ascolto di connessioni (*listen()*

eseguita); *valopz* vale 1 in caso positivo, 0 altrimenti;

- SO_DONTROUTE: utilizzabile da entrambe le funzioni; se *valopz* è 1 significa che il socket opera solo con nodi raggiungibili direttamente ignorando la tabella di *routing*; se è 0 il socket può operare usando la tabella di *routing*;
- SO_BROADCAST: utilizzabile da entrambe le funzioni; se *valopz* è 1 il socket riceve datagrammi indirizzati all'indirizzo *broadcast* e può anche inviarne a tale indirizzo; questa opzione ha effetto solo su comunicazioni di tipo SOCK_DGRAM;
- SO_REUSEADDR: utilizzabile da entrambe le funzioni; se *valopz* è 1 è possibile effettuare la *bind()* su indirizzi locali che sono già in uso; questo è molto utile in almeno due casi:
  1. se un servente è terminato e deve essere fatto ripartire ma ancora qualche suo processo figlio è attivo su una connessione che utilizza l'indirizzo locale, alla ripartenza del servente, quando si effettua la *bind()* sul socket, si ottiene l'errore EADDRINUSE; l'opzione SO_REUSEADDR a 1 permette di evitare l'errore;
  2. se si vuole avere la possibilità di più programmi o più istanze dello stesso programma in ascolto sulla stessa porta ma con indirizzi IP diversi;

  queste situazioni sono abbastanza comuni e quindi i serventi TCP hanno spesso l'opzione SO_REUSEADDR impostata a 1.

A titolo di esempio vediamo come impostare l'uso degli indirizzi *broadcast* per un socket UDP precedentemente aperto e identificato con *sd*:

```
val=1;
ritorno=setsockopt(sd,SOL_SOCKET,SO_BROADCAST,&val,sizeof (val)));
```

3.8.2 Recupero indirizzo locale di un socket

In certi casi può essere utile sapere quale è l'indirizzo locale asoociato a un socket; ad esempio in un processo cliente per conoscere IP e porta assegnati automaticamente dal *kernel* dopo la *bind()* oppure in un servente che ha eseguito la *bind()* con numero di porta locale 0 e vuole conoscere la porta assegnata dal *kernel*.

A questo scopo si usa la funzione *getsockname()* che ha il seguente prototipo:

```
int getsockname(int sd, struct sockaddr *nome, socklen_t *lunnome)

//legge indirizzo locale del socket sd;
//nome serve a ricevere l'indirizzo letto;
//lunnome è la lunghezza dell'indirizzo.
```

Ritorna 0 in caso di successo e -1 se c'è errore, nel qual caso *errno* può valere:

- EBADF o ENOTSOCK: socket non valido;
- ENOBUFS: risorse non sufficienti nel sistema per eseguire l'operazione;
- EFAULT: indirizzo *nome* non valido.

La funzione è anche utile nel caso di un servente che ha eseguito una *bind()* su un indirizzo generico e che dopo il completamento della connessione a seguito della *accept()* vuole conoscere l'indirizzo locale asegnato dal *kernel* a quella connessione.

3.8.3 Recupero indirizzo remoto di un socket

Per conoscere l'indirizzo remoto di un socket si usa la funzione *getpeername()*, il cui prototipo è:

```
int getpeername(int sd, struct sockaddr * nome, socklen_t * lunnome)

//legge indirizzo remoto del socket sd;
//nome serve a ricevere l'indirizzo letto;
//lunnome è la lunghezza dell'indirizzo.
```

Ritorna 0 in caso di successo e -1 se c'è errore con valori per *errno* uguali a quelli della funzione *getsockname()*.

La funzione è del tutto simile a *getsockname()*, ma ma restituisce l'indirizzo remoto del socket.

Apparentemente sembra inutile visto che:

- il cliente conosce per forza l'indirizzo remoto con cui fare la connessione;
- il servente può usare i valori di ritorno della funzione *accept()*.

Esiste però una situazione in cui la funzione è utile e cioè quando un servente lancia un programma per ogni connessione ricevuta attraverso una delle funzioni della famiglia *exec* (questo ad esempio è il comportamneto del demone di GNU/Linux *inetd*).

In tal caso infatti il processo generato perde ogni riferimento ai valori dei file e dei socket utilizzati dal processo padre (a differenza di quello che accade quando si genera un processo con la funzione *fork()*) e quindi anche la struttura ritornata dalla *accept()*; il descrittore del socket però è ancora aperto e, se il padre segue una opportuna convenzione per rendere noto al programma generato qual'è il socket connesso, (ad esempio usando sempre gli stessi valori interi come descrittori) quest'ultimo può usare *getpeername()* per conoscere l'indirizzo remoto del cliente.

Dovrebbe essere possibile fare riferimento a questa pagina anche con il nome le_funzioni_dei_socket_di_rete.html

[successivo] [precedente] [inizio] [fine] [indice generale]


ISO/IEC
15445


W3C
CSS