

---

# Programmazione dei socket di rete in GNU/Linux

---

Fulvio Ferroni *fulvioferroni@teletu.it*

2006.09.21

Copyright © Fulvio Ferroni *fulvioferroni@teletu.it*

Via Longarone, 6 - 31030 - Casier (TV)

Le informazioni contenute in questa opera possono essere diffuse e riutilizzate in base alle condizioni poste dalla licenza GNU General Public License, come pubblicato dalla Free Software Foundation.

In caso di modifica dell'opera e/o di riutilizzo parziale della stessa, secondo i termini della licenza, le annotazioni riferite a queste modifiche e i riferimenti all'origine di questa opera, devono risultare evidenti e apportate secondo modalità appropriate alle caratteristiche dell'opera stessa. In nessun caso è consentita la modifica di quanto, in modo evidente, esprime il pensiero, l'opinione o i sentimenti del suo autore.

L'opera è priva di garanzie di qualunque tipo, come spiegato nella stessa licenza GNU General Public License.

Queste condizioni e questo copyright si applicano all'opera nel suo complesso, salvo ove indicato espressamente in modo diverso.

# Indice generale

Premessa .....	V
L'autore .....	V
1 Il software di rete .....	1
1.1 Tipi di applicazioni di rete .....	1
1.2 Richiami sulle connessioni TCP .....	1
1.2.1 Il segmento TCP .....	1
1.2.2 Attivazione della connessione TCP .....	3
1.2.3 Rilascio della connessione TCP .....	3
1.3 Tipologie di server .....	5
1.4 Comunicazione tra processi in rete .....	5
2 I socket di rete .....	6
2.1 Funzioni da usare per la programmazione in rete .....	6
2.2 Creazione di un socket .....	7
2.3 Socket bloccanti e non bloccanti .....	9
2.4 Indirizzi dei socket .....	9
2.4.1 Struttura indirizzi generica .....	9
2.4.2 Struttura indirizzi IPv4 .....	10
2.4.3 Struttura indirizzi IPv6 .....	11
2.4.4 Struttura indirizzi locali .....	11
2.5 Conversione dei valori numerici per la rete .....	12
2.5.1 Ordinamento <i>big endian</i> e <i>little endian</i> .....	12
2.5.2 Funzioni di conversione di indirizzi e porte .....	13
2.5.3 Conversione del formato degli indirizzi .....	13
2.6 Uso della funzione <i>ioctl()</i> con i socket .....	15
3 Le funzioni dei socket di rete .....	18
3.1 Assegnazione dell'indirizzo a un socket .....	18
3.2 Connessione .....	19
3.3 Attesa di connessione .....	20
3.4 Invio e ricezione dati .....	21
3.5 Invio e ricezione dati con socket UDP .....	22
3.6 Socket UDP connessi .....	25
3.7 Chiusura di un socket .....	25
3.8 Altre funzioni per i socket .....	26
3.8.1 Impostazione e lettura delle opzioni di un socket .....	26
3.8.2 Recupero indirizzo locale di un socket .....	28
3.8.3 Recupero indirizzo remoto di un socket .....	29
4 <i>Multiplexing</i> dell'input/output .....	30

5	Risoluzione dei nomi .....	33
5.1	Nome della macchina ospite .....	33
5.2	Ricavare l'indirizzo IP dal nome .....	33
5.3	Risoluzione inversa .....	34
5.4	Risoluzione con connessioni TCP .....	35
5.5	Nomi dei servizi noti .....	35
6	Esempi vari .....	36
6.1	Servente concorrente che riceve un file da un cliente connesso .....	37
6.2	Servente iterativo che scambia messaggi con un cliente .....	42
6.3	Servente di <i>chat</i> con <i>multiplexing</i> .....	45
6.4	Acquisizione dei dati delle interfacce di rete con <i>ioctl()</i> .....	52

# Premessa

Questo documento spiega come programmare a basso livello, in linguaggio C, i socket di rete in ambiente GNU/Linux per arrivare a costruire piccole applicazioni serventi e clienti da utilizzare a scopo didattico.

Vengono considerate già note le nozioni di base circa la configurazione di una rete locale in ambiente GNU/Linux e in particolare quelle relative agli indirizzi di rete IPv4 e alla natura e ruolo delle *porte* di rete.

Il contenuto di queste dispense prende per molti argomenti spunto dalla seconda parte dell'opera «Guida alla Programmazione in Linux» di Simone Piccardi <http://gapil.firenze.linux.it> sicuramente più completa e dettagliata. Qui lo scopo è proporre uno strumento non esaustivo, ma agile e semplificato, adatto ad un eventuale utilizzo in ambito didattico a livello di scuola superiore; non vengono quindi elencati tutti i dettagli, le opzioni, i valori dei parametri delle funzioni. Per una trattazione completa si rimanda all'opera appena citata o ai manuali in linea delle funzioni utilizzate.

## L'autore

Fulvio Ferroni  
via Longarone,6 31030 Casier (TV)  
fulvioferroni@teletu.it

Docente di sistemi informatici presso l'ITIS "Max Planck" di Lancenigo di Villorba (TV).



# Il software di rete

La programmazione di rete si occupa della realizzazione di programmi che vengono eseguiti su macchine diverse e devono comunicare attraverso la rete.

Il problema può quindi essere visto come la generalizzazione di quello della comunicazione tra processi (IPC *Inter Process Communication*) in cui i processi sono eseguiti su una stessa macchina.

## 1.1 Tipi di applicazioni di rete

Esistono vari modelli di architettura per le applicazioni di rete; i più importanti sono:

- cliente/ servente (*client/server*);
- paritetico (P2P, *peer to peer*);
- a tre livelli (*three-tier*).

Il modello a tre livelli è molto interessante ma in un contesto diverso da quello preso in esame in queste dispense e cioè nel campo della realizzazione di applicazioni basate sul Web; quindi non viene approfondito in questa sede.

Il modello paritetico ha acquisito importanza e notorietà grazie alla diffusione dei servizi di condivisione di file tra nodi paritetici in Internet.

Il modello più importante in ambito GNU/Linux e, storicamente in Unix, è però quello cliente/ servente sul quale si sofferma in modo particolare il presente documento.

Lo scenario considerato è quello di comunicazioni in rete basate su protocolli orientati alla connessione (TCP) dove un cliente invia richieste di connessione a un servente che è in attesa di tali richieste.

Comunque anche le comunicazioni basate su protocolli non orientati alla connessione (UDP) vengono esaminati anche se più brevemente.

## 1.2 Richiami sulle connessioni TCP

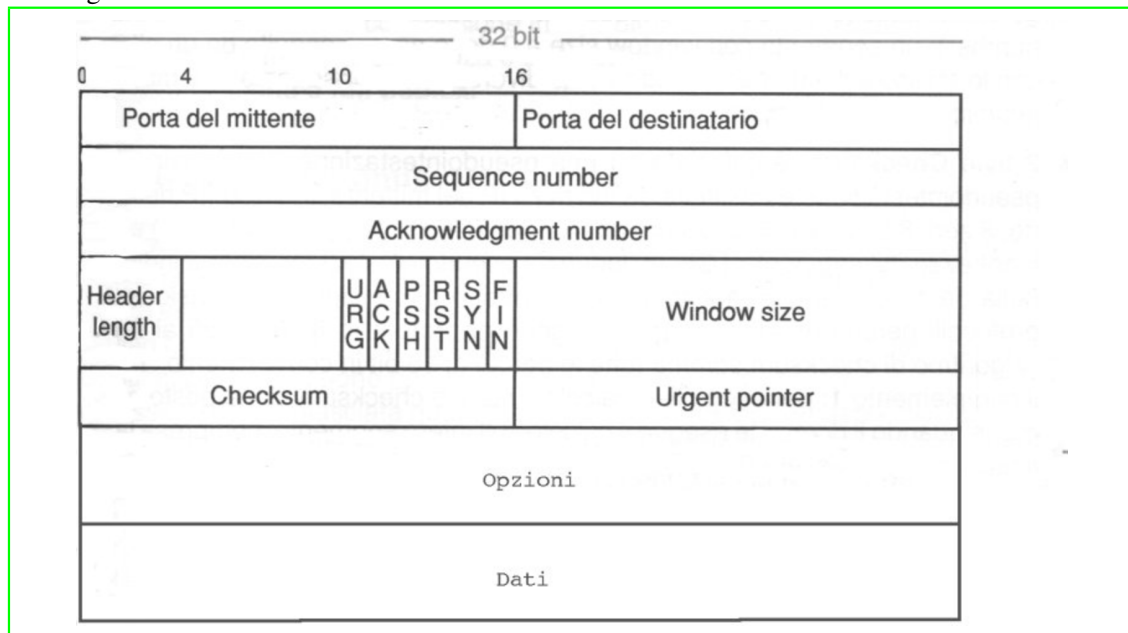
Questo paragrafo contiene una breve illustrazione del funzionamento del protocollo TCP, in particolare riguardo i meccanismi di attivazione e rilascio di una connessione.

La trattazione non ha la minima pretesa di essere esaustiva; per conoscere più in profondità questo argomento, e in generale tutto ciò che riguarda il software di rete, si consiglia la consultazione di testi specializzati come «Reti di calcolatori» di A.S. Tanenbaum.

### 1.2.1 Il segmento TCP

Nella figura 1.1 è riportato il formato del segmento TCP.

Figura 1.1



Segue una breve descrizione delle funzioni dei vari campi:

- *porta del mittente* e *porta del destinatario*: sono le porte che identificano gli estremi della connessione a livello di trasporto; la connessione è comunque completamente identificata indicando anche l'indirizzo IP dei due nodi, quindi da due coppie di valori IP:porta;
- *sequence number*: numero d'ordine del primo byte contenuto nel campo dati;
- *acknowledgement number*: numero d'ordine del prossimo byte atteso;
- *header length*: numero di parole di 32 bit presenti nell'intestazione; tale numero può variare perché il campo *opzioni* è di dimensione variabile;
- *URG*: è un bit e vale 1 se è usato il campo *urgent pointer*, 0 altrimenti;
- *ACK*: è un bit e vale 1 se *acknowledgement number* è valido, 0 altrimenti;
- *PSH*: è un bit e vale 1 se dati urgenti (*pushed data*), da consegnare senza aspettare il riempimento del buffer;
- *RST*: è un bit e vale 1 se viene richiesto il reset della connessione a seguito di qualche problema;
- *SYN*: è un bit e vale 1 in fase di attivazione della connessione;
- *FIN*: è un bit e vale 1 in fase di rilascio della connessione;
- *window size*: dimensione in byte della finestra di spedizione usata per il controllo di flusso di tipo *sliding window go-back-n*; in pratica questo campo indica quanti byte possono essere spediti a partire da quello confermato con *acknowledgement number*; un valore zero indica al nodo mittente di interrompere la trasmissione e riprenderla quando riceve un segmento con stesso *acknowledgement number* e *window size* diversa da zero;
- *checksum*: serve per un controllo di correttezza sul segmento inviato includendo però anche una pseudointestazione contenente gli indirizzi IP; questo permette al ricevente di individuare i pacchetti consegnati in modo errato ma è una violazione della gerarchia dei protocolli in quanto gli indirizzi IP appartengono al livello rete e non a quello di trasporto dove «risiede» il TCP;



- *urgent pointer*: puntatore ai dati urgenti;
- *opzioni*: vengono negoziate all'attivazione della connessione; le più importanti sono: dimensione massima dei segmenti da spedire, uso di *selective repeat* invece che *go-back-n*, uso di segmenti NAK;

### 1.2.2 Attivazione della connessione TCP

La connessione TCP tra due nodi di rete viene attivata con il meccanismo TWH (*Three Way Handshake*).

Uno dei due nodi, il server, esegue le funzioni *listen()* e *accept()* e rimane in attesa di una richiesta di connessione su una certa porta; l'altro nodo, il cliente, esegue la funzione *connect()* indicando l'indirizzo IP e la porta del server.

Questo causa l'invio di un segmento TCP col bit SYN a uno, il bit ACK a zero e un numero di sequenza creato al momento con un meccanismo basato sul clock di sistema.

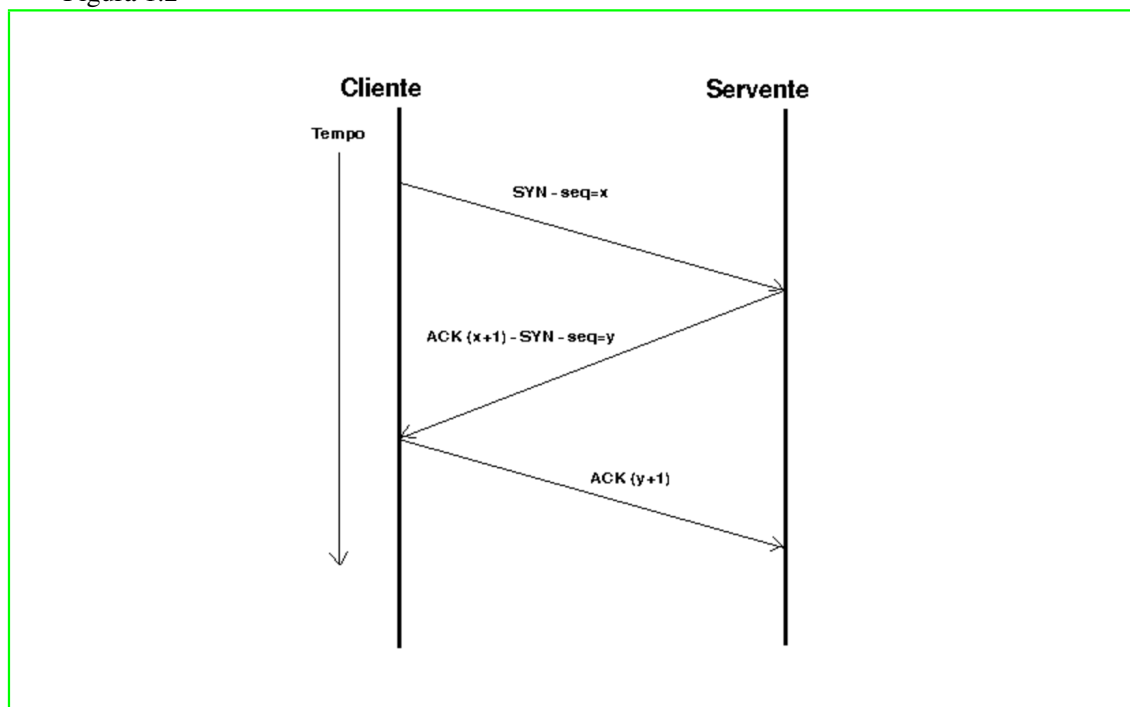
All'arrivo di tale segmento l'entità a livello di trasporto del server controlla se c'è un processo in ascolto sulla porta specificata:

- in caso negativo invia una risposta col bit RST a uno, in modo da rifiutare la connessione;
- in caso positivo e se il processo accetta la connessione, viene inviato un segmento di conferma, con bit ACK a uno; in tale segmento però anche il bit SYN viene posto a uno e viene generato un altro numero di sequenza, affinché anche il server chieda conferma della connessione al cliente.

Se anche quest'ultimo accetta la connessione invia una risposta con bit ACK a uno al server e l'attivazione si conclude con successo; da questo momento i due nodi possono iniziare a scambiarsi segmenti dati.

Nella figura 1.2 è schematizzato il meccanismo di attivazione di una connessione TCP.

Figura 1.2



### 1.2.3 Rilascio della connessione TCP

Per il rilascio, la connessione, che è *full-duplex*, viene considerata come una coppia di connessioni *simplex* indipendenti.

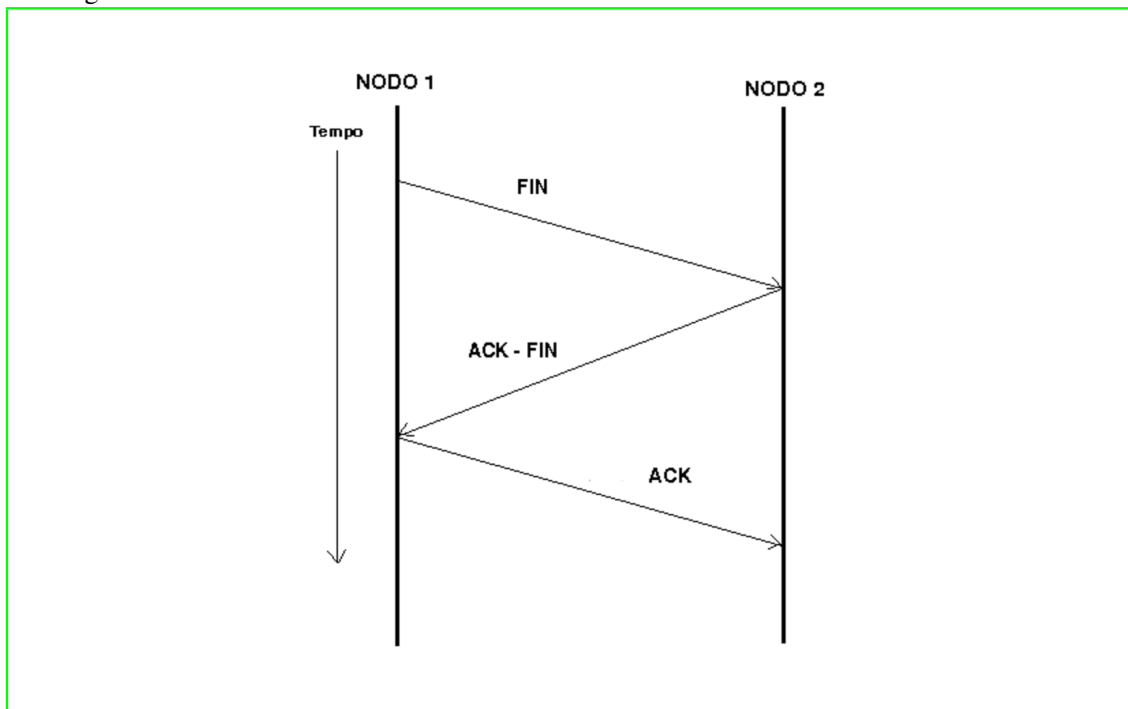
Il meccanismo (stavolta a «quattro vie») è il seguente:

- uno dei due nodi invia un segmento con bit FIN a uno;
- se l'altro nodo conferma con un segmento con ACK a uno, la connessione in uscita dal primo nodo viene rilasciata;
- anche l'altro nodo esegue lo stesso procedimento e rilascia la connessione nell'altra direzione.

Anche in questo caso i passi del procedimento possono diventare tre se il secondo nodo invia in un solo segmento l'ACK e il suo FIN al primo nodo.

Nella figura 1.3 è schematizzato il meccanismo di rilascio di una connessione TCP.

Figura 1.3



Per evitare il problema noto in letteratura tecnica come «problema dei 2 eserciti» vengono usati dei contatori di tempo.

Il problema dei due eserciti consiste in due eserciti alleati che devono attaccare insieme, pena la sconfitta, un esercito nemico e quindi devono sincronizzare le proprie azioni con l'invio di messaggeri che però possono essere catturati o uccisi; è una situazione molto simile a quella dei due nodi di rete che devono sincronizzarsi per rilasciare la connessione correttamente e si inviano dei messaggi che possono andare persi.

Il problema, che non ha una soluzione teoricamente perfetta, viene affrontato, in modo comunque efficace nella pratica con l'uso di temporizzatori: se una risposta a un FIN non arriva entro un certo tempo, il mittente del FIN chiude la connessione; l'altro nodo va in *timeout* perché si accorge che nessuno sembra ascoltarlo più.

## 1.3 Tipologie di server

I processi server in rete possono essere di due tipi diversi:

- *iterativi*: rispondono alle richieste di servizio e restano occupati finì al loro completamento, al termine ritornano disponibili per altre richieste;
- *concorrenti*: gestiscono le richieste creando processi figli, incaricati di rispondere, e ponendosi immediatamente ancora in ascolto di altre richieste; i processi figli terminano una volta esaurito il loro compito.

## 1.4 Comunicazione tra processi in rete

Le interfacce software o API (*Application Program Interface*) di comunicazione più utilizzate fra processi in rete, sono:

- socket di Berkeley, di Unix BSD;
- TLI (*Transport Layer Interface*) di Unix System V.

Entrambe sono definite per il linguaggio C e fra le due la API basata sui socket è senz'altro la più usabile e flessibile e quindi la più diffusa; essa è stata introdotta nel 1982 in Unix BSD 4.1c ed è rimasta sostanzialmente invariata.

I socket vengono usati per la comunicazione di rete in molti sistemi operativi, compresi Unix e GNU/Linux e possono essere utilizzati anche per far comunicare processi in esecuzione sulla stessa macchina.

La flessibilità dei socket permette di usarli a fronte di stili di comunicazione diversi: ad esempio nel caso di invio dei dati come flusso di byte o suddivisi in pacchetti, di comunicazioni affidabili o non affidabili, di comunicazioni punto a punto o *broadcast* e altro ancora.

Nel caso di comunicazione in rete i socket possono essere usati con protocolli diversi; come già detto però in queste dispense si fa riferimento unicamente all'uso dei socket con l'architettura TCP/IP (protocolli TCP e UDP).

# I socket di rete

L'interfaccia software basata su socket prevede l'utilizzo di un *handle* (maniglia) per gestire un flusso di dati, in modo del tutto analogo a quello che avviene con i file; tale *handle* è un intero che prende il nome di socket.

Quindi un socket di rete è un canale di comunicazione tra due processi in rete, identificato da un intero, sul quale ognuno dei processi può leggere o scrivere usando funzioni simili a quelle usate per leggere o scrivere sui file (a basso livello).

## 2.1 Funzioni da usare per la programmazione in rete

Per il linguaggio C esistono delle funzioni (molte delle quali riguardano i socket) e strutture dati da utilizzare per realizzare processi che comunicano in rete; esse sono definite in alcuni *header* memorizzati solitamente in `/usr/local`.

Nella tabella 2.1 sono elencate alcune di queste funzioni con il relativo *header*.

Tabella 2.1

uso	funzione	header
creazione socket	int socket()	<sys/socket.h>
creazione socket	int bind()	<sys/socket.h>
connessione socket	int connect()	<sys/socket.h>
ascolto socket	int listen()	<sys/socket.h>
attesa connessione socket	int accept()	<sys/socket.h>
ascolto socket	int select()	<sys/select.h>
lettura socket	ssize_t recvfrom()	<sys/socket.h>
lettura socket	ssize_t recv()	<sys/socket.h>
lettura socket	ssize_t read()	<unistd.h>
scrittura socket	ssize_t sendto()	<sys/socket.h>
scrittura socket	ssize_t send()	<sys/socket.h>
scrittura socket	ssize_t write()	<unistd.h>
chiusura socket	int close()	<unistd.h>
chiusura socket	int shutdown()	<sys/socket.h>
lettura ind. locale socket	int getsockname()	<sys/socket.h>
lettura ind. remoto socket	int getpeername()	<sys/socket.h>
impostazione opzioni socket	int setsockopt()	<sys/socket.h>
lettura opzioni socket	int getsockopt()	<sys/socket.h>
conversione indirizzi	short int htons()	<netinet/in.h>
conversione indirizzi	short int ntohs()	<netinet/in.h>
conversione indirizzi	long int htonl()	<netinet/in.h>
conversione indirizzi	long int ntohl()	<netinet/in.h>
conversione indirizzi	int inet_aton()	<arpa/inet.h>
conversione indirizzi	char *inet_ntoa()	<arpa/inet.h>
conversione indirizzi	int inet_pton()	<arpa/inet.h>
conversione indirizzi	char *inet_ntop()	<arpa/inet.h>
nome <i>host</i>	int gethostname()	<unistd.h>
risoluzione nomi	struct hostent *gethostbyname()	<netdb.h>
risoluzione nomi	struct hostent *gethostbyname2()	<netdb.h>
risoluzione nomi	void sethostent ()	<netdb.h>
risoluzione nomi	void endhostent ()	<netdb.h>
risoluzione nomi	struct hostent *gethostbyaddr()	<netdb.h>
nomi di servizi	struct servent *getservbyname()	<netdb.h>
nomi di servizi	struct servent *getservbyport()	<netdb.h>

Il tipo `ssize_t` è un *tipo primitivo* definito in `<sys/types.h>` (come `size_t` usato nel paragrafo 2.4.2).

Le strutture `hostent` e `servent` sono definite in `<netdb.h>` e vengono descritte, insieme alle funzioni che le usano, nel capitolo 5.

Quando si programma in C in Unix o GNU/Linux si deve porre attenzione ai problemi di portabilità tra piattaforme diverse; storicamente infatti alcuni tipi di dati dello standard ANSI C sono stati associati a variabili dei sistemi ospiti dandone per scontata la dimensione. Ad esempio il puntatore all'interno di un file è un intero a 32 bit, cioè un *int*, il numero di dispositivo è un intero a 16 bit cioè uno *short*; cambiando la piattaforma hardware però questi tipi possono non corrispondere dando origine a notevoli problemi di portabilità. Per questo motivo le funzioni di libreria non fanno riferimento ai tipi elementari del linguaggio C, ma ai tipi primitivi del sistema, definiti in `<sys/types.h>`. In questo modo i tipi di dati utilizzati da tali funzioni rimangono indipendenti dai tipi elementari supportati dal particolare compilatore C utilizzato.

Anche le funzioni da usare per i socket fanno largo uso dei tipi di dati primitivi e quindi per il loro utilizzo è sempre necessario includere l'*header* `<sys/types>` oltre a quello in cui la funzione è definita.

Per la funzione `select()` è inoltre necessaria l'inclusione di `<sys/time.h>`.

## 2.2 Creazione di un socket

La funzione per la creazione di un socket è `socket()` il cui prototipo è il seguente:

```
int socket(int domain, int type, int protocol)
```

restituisce l'identificatore del socket oppure -1 se c'è errore; in tal caso viene valorizzata la variabile globale `errno`, definita nell'*header* `<errno.h>`, con opportuni valori i più significativi dei quali sono:

- `EPROTONOSUPPORT`: socket o protocollo non supportato nel dominio;
- `EACCES`: mancanza di privilegi per creare il socket;
- `EINVAL`: protocollo sconosciuto o dominio non disponibile;
- `ENOBUFS` o `ENOMEM`: memoria non sufficiente per la creazione del socket.

I valori elencati in maiuscolo sono alcuni *nomi simbolici* associati ai valori numerici che identificano i vari errori. Tutti i nomi simbolici di errori iniziano per «E», sono nomi riservati e sono definiti in `<errno.h>`. Il loro utilizzo è consigliato ogniqualvolta si usa una delle funzioni delle librerie del C e questa ritorna «errore» (-1); in tal caso infatti può essere utile conoscere il tipo di errore occorso testando la variabile *errno*. I valori che essa può assumere in seguito all'invocazione di una funzione che va in errore sono elencati nel manuale in linea della funzione utilizzata.

Per visualizzare il messaggio di errore associato a un certo valore della variabile *errno* si può utilizzare la funzione *perror()* che ha il seguente prototipo:

```
void perror(const char *msg)

//stampa sullo standard error il messaggio di errore relativo;
//al valore di errno preceduto da msg, da un «:» e
//      da uno spazio.
```

Il valore di *errno* che viene preso in considerazione è quello relativo all'ultimo errore avvenuto.

I parametri della funzione *socket()* sono:

- domain: famiglia dei protocolli da usare, con i seguenti valori possibili (solo i più utili per gli scopi di queste dispense):
  - PF\_UNIX o PF\_LOCAL: comunicazioni locali;
  - PF\_INET: comunicazioni con protocollo IPv4;
  - PF\_INET6: comunicazioni con protocollo IPv6.
- type: stile di comunicazione identificato dai seguenti valori:
  - SOCK\_STREAM: comunicazione bidirezionale, sequenziale, affidabile, in connessione con un altro socket; dati inviati e ricevuti come flusso di byte;
  - SOCK\_SEQPACKET: come SOCK\_STREAM ma con dati suddivisi in pacchetti di dimensione massima prefissata;
  - SOCK\_DGRAM: pacchetti (*datagram*) di lunghezza massima prefissata, indirizzati singolarmente, senza connessione e in modo non affidabile;
  - SOCK\_RDM: comunicazione affidabile ma senza garanzia sull'ordine di arrivo dei pacchetti.
- protocol: deve valere sempre zero.

I nomi indicati in maiuscolo sono *nomi simbolici* definiti in `<sys/socket.h>`.

I valori più interessanti dello stile di comunicazione sono SOCK\_STREAM e SOCK\_DGRAM perché sono quelli utilizzabili (sia con PF\_INET che con PF\_INET6) rispettivamente con i protocolli TCP e UDP (e sono utilizzabili anche con PF\_UNIX).

## 2.3 Socket bloccanti e non bloccanti

Un socket viene normalmente creato come «bloccante»; ciò significa che, a seguito di una chiamata alla funzione di attesa connessione, blocca il processo chiamante, fino all'arrivo di una richiesta di connessione.

Un socket «non bloccante», invece, non provoca il blocco del processo chiamante in una attesa indefinita; se al momento dell'attesa di connessione non è presente alcuna richiesta, il processo continua la sua esecuzione e la funzione di attesa connessione fornisce un appropriato codice di errore.

Per rendere un socket non bloccante si deve usare la funzione *fcntl()*, di controllo dei file (un socket può essere in effetti assimilato ad un file gestito a basso livello) definita in *<fcntl.h>*, nel modo seguente:

```
fcntl(sd, F_SETFL, O_NONBLOCK);

//sd è il socket precedentemente aperto;
//F_SETFL significa che si vuole impostare il file status flag
//      al valore dall'ultimo parametro;
//O_NONBLOCK costante corrispondente al valore che significa non bloccante.
```

Ritorna -1 se c'è qualche errore (comunemente socket non aperto) oppure un valore che dipende dal tipo di operazione richiesto.

Anche il numero e la natura dei parametri varia secondo il tipo di utilizzo che può essere molteplice; per i dettagli si rimanda alla consultazione del manuale in linea della funzione.

## 2.4 Indirizzi dei socket

La creazione di un socket serve solo ad allocare nel *kernel* le strutture necessarie (in particolare nella *file table*) e a indicare il tipo di protocollo da usare; non viene specificato niente circa gli indirizzi dei processi coinvolti nella comunicazione.

Tali indirizzi si indicano con altre funzioni, utili alla effettiva realizzazione del collegamento, mediante l'uso di apposite strutture dati.

### 2.4.1 Struttura indirizzi generica

Siccome le funzioni di gestione dei socket sono concepite per funzionare con tutti i tipi di indirizzi presenti nelle varie famiglie di protocolli, è stata definita in *<sys/socket.h>* una struttura generica per gli indirizzi dei socket:

```
struct sockaddr {
    sa_family_t    sa_family;    // famiglia di indirizzi
    char           sa_data[14];  // indirizzo
};
```

Le funzioni che usano gli indirizzi hanno nel prototipo un puntatore a questa struttura; quando si richiamano occorre fare il *casting* del puntatore effettivo per il protocollo specifico utilizzato.

I nomi delle strutture dati associate ai vari protocolli hanno la forma *sockaddr\_* seguito da un

suffisso che dipende dal protocollo.

Tutte queste strutture, e anche quella generica, usano dei tipi dati standard (*POSIX*) che sono descritti in tabella 2.2 dove è indicato anche l'*header* di definizione:

Tabella 2.2

tipo dato	descrizione	header
int8_t	intero a 8 bit con segno	<sys/types.h>
uint8_t	intero a 8 bit senza segno	<sys/types.h>
int16_t	intero a 16 bit con segno	<sys/types.h>
uint16_t	intero a 16 bit senza segno	<sys/types.h>
int32_t	intero a 32 bit con segno	<sys/types.h>
uint32_t	intero a 32 bit senza segno	<sys/types.h>
sa_family_t	famiglia degli indirizzi	<sys/socket.h>
socklen_t	lunghezza ind. (uint32_t)	<sys/types.h>
in_addr_t	indirizzo IPv4 (uint32_t)	<netinet/in.h>
in_port_t	porta TCP o UDP (uint16_t)	<netinet/in.h>

In queste dispense vediamo solo le strutture relative agli indirizzi IPv4, IPv6 e locali.

## 2.4.2 Struttura indirizzi IPv4

La struttura indirizzi IPv4 è definita in <netinet/in.h> nel modo seguente:

```
struct sockaddr_in {
    sa_family_t    sin_family;    // famiglia, deve essere = AF_INET
    in_port_t      sin_port;      // porta
    struct in_addr  sin_addr;      // indirizzo IP
    unsigned char  sin_zero[8]    // per avere stessa dim. di sockaddr
};
```

con la struttura *in\_addr* così definita:

```
struct in_addr {
    u_int32_t      s_addr;        // indirizzo IPv4 di 32 bit
};
```

Il campo *sin\_zero* è inserito affinché *sockaddr\_in* abbia la stessa dimensione di *sockaddr* e deve essere valorizzato con zeri con la funzione *memset()*, definita in <string.h> e il cui prototipo è il seguente:

```
void *memset(void *s, int c, size_t n);
```

La funzione riempie i primi *n* byte dell'area di memoria puntata da *s* con il byte *c*.



### 2.4.3 Struttura indirizzi IPv6

La struttura indirizzi IPv6 è definita in `<netinet/in.h>` nel modo seguente:

```
struct  sockaddr_in6 {
    sa_family_t      sin6_family;    // famiglia, deve essere = AF_INET6
    in_port_t        sin6_port;      // porta
    uint32_t          sin6_flowinfo; // informazioni sul flusso IPv6
    struct in6_addr    sin6_addr;     // indirizzo IP
    uint32_t          sin6_scope_id;  // id di scope
};
```

con la struttura *in6\_addr* così definita:

```
struct  in6_addr {
    uint8_t           s6_addr[16];   // indirizzo IPv6 di 128 bit
};
```

Il campo *sin6\_flowinfo* è relativo a certi campi specifici dell'intestazione dei pacchetti IPv6 e il suo uso è sperimentale.

Il campo *sin6\_scope\_id* è stato introdotto in Linux con il *kernel* 2.4, e serve per gestire il multicasting.

La struttura *sockaddr\_in6* è più grande di *sockaddr* quindi non serve il riempimento con zeri necessario invece in *sockaddr\_in*.

### 2.4.4 Struttura indirizzi locali

La struttura indirizzi locali è definita in `<sys/un.h>` nel modo seguente:

```
#define UNIX_PATH_MAX    108
struct  sockaddr_un {
    sa_family_t  sun_family;           // famiglia, deve essere = AF_UNIX
    char         sun_path[UNIX_PATH_MAX]; // nome percorso
};
```

Il campo *sun\_path* contiene l'indirizzo che può essere:

- un file nel filesystem il cui nome, completo di percorso, è specificato come una stringa terminata da uno zero;
- una stringa univoca che inizia con zero e con i restanti byte, senza terminazione, che rappresentano il nome.

## 2.5 Conversione dei valori numerici per la rete

Gli indirizzi IP e i numeri di porta devono essere specificati nel formato chiamato *network order*, che corrisponde al *big endian* e che si contrappone al *little endian*.

### 2.5.1 Ordinamento *big endian* e *little endian*

Nel caso di posizioni di memoria costituite da più byte (ad esempio di 16 bit) l'ordine con cui i diversi byte di una stessa posizione sono memorizzati dipende dall'architettura del computer.

I due ordinamenti più diffusi sono:

- *big endian* o *big end first*: in questo caso le posizioni di memoria sono occupate a partire dal byte più a sinistra del dato, quindi dal più significativo;
- *little endian* o *little end first*: in questo caso le posizioni di memoria sono occupate a partire dal byte più a destra del dato, quindi dal meno significativo.

Da quanto detto emerge che nel caso di *big endian* il byte più significativo (MSB *Most Significant Byte*) ha l'indirizzo di memoria più piccolo, mentre nel caso di *little endian* è il byte meno significativo (LSB *Least Significant Byte*) ad avere l'indirizzo più piccolo.

Ad esempio se si deve memorizzare il dato 'AB' a partire dall'indirizzo 100, avremo, nel caso di *big endian*:

```
indirizzo 100: A
indirizzo 101: B
```

invece, nel caso di *little endian*:

```
indirizzo 100: B
indirizzo 101: A
```

I termini *big endian* e *little endian* derivano dai Lillipuziani dei "Viaggi di Gulliver", il cui problema principale era se le uova debbano essere aperte dal lato grande (*big endian*) o da quello piccolo (*little endian*); il significato di questa analogia è ovvio: nessuno dei due metodi è migliore dell'altro.

Esiste comunque un problema di compatibilità noto come '**NUXI problem**' dovuto al fatto che i processori Intel usano il metodo *little endian* e quelli Motorola il metodo *big endian*; si dice anche che hanno *endianess* deverse.

Il termine NUXI deriva dall'aspetto che avrebbe la parola UNIX se memorizzata in due posizioni consecutive di due byte in *little endian*.

Il seguente programma può essere utilizzato per verificare la *endianess* della propria macchina:<sup>1</sup>

```
#include <stdio.h>
#include <sys/types.h>
int main(void) {
    union {
        long lungo;
        char ch[sizeof(long)];
    };
}
```

```
    } unione;  
    unione.lungo = 1;  
    if (unione.ch[sizeof(long)-1] == 1)  
        printf("big endian\n");  
    else  
        printf("little endian\n");  
    return (0);  
}
```

## 2.5.2 Funzioni di conversione di indirizzi e porte

Da quanto detto emerge la necessità di convertire i valori numerici corrispondenti agli indirizzi e alle porte in *network order* o **formato rete** cioè *big endian* se stiamo lavorando su una piattaforma che usa come **formato macchina** il *little endian*.

Le funzioni da utilizzare sono le seguenti (definite anche sulle piattaforme che usano *big endian* ma, ovviamente, vuote):

```
unsigned long int htonl(unsigned long int hostlong)  
  
//converte l'intero a 32 bit hostlong dal formato macchina al formato rete;  
  
unsigned short int htons(unsigned short int hostshort)  
  
//converte l'intero a 16 bit hostshort dal formato macchina al formato rete;  
  
unsigned long int ntohl(unsigned long int netlong)  
  
//converte l'intero a 32 bit netlong dal formato rete al formato macchina;  
  
unsigned short int ntohs(unsigned short int netshort)  
  
//converte l'intero a 16 bit netshort dal formato rete al formato macchina.
```

Tutte le funzioni ritornano il valore convertito, e non sono previsti errori.

## 2.5.3 Conversione del formato degli indirizzi

Esistono anche delle funzioni per convertire gli indirizzi dal formato rete binario al *dotted decimal*, o **decimale puntato**, caratteristico degli indirizzi IPv4; tali funzioni sono definite in `<arpa/inet.h>` e i seguenti sono i loro prototipi:

```
in_addr_t inet_addr(const char *strptr)  
  
//converte la stringa strptr decimale puntata nel numero IP in formato rete;  
  
int inet_aton(const char *src, struct in_addr *dest)  
  
//converte la stringa src decimale puntata in un indirizzo IP;
```

```
char *inet_ntoa(struct in_addr ind)

//converte l'indirizzo IP ind in una stringa decimale puntata.
```

L'uso della prima funzione è deprecato a favore della seconda.

La funzione *inet\_aton()* converte la stringa *src* nell'indirizzo binario *dest* in formato rete e restituisce 0 in caso di successo e 1 in caso di errore; può anche essere usata con il parametro *dest* pari a *NULL* nel qual caso controlla se l'indirizzo è valido (valore di ritorno 0) oppure no (valore di ritorno 1).

La funzione *inet\_ntoa()* converte l'indirizzo *addr\_ptr*, espresso in formato rete, nella stringa in formato decimale puntata che è il valore di ritorno della funzione stessa.

Le funzioni appena illustrate operano solo con indirizzi IPv4 mentre ce ne sono altre che possono gestire anche indirizzi IPv6:

```
int inet_pton(int af, const char *src, void *addr_ptr)

//converte l'indirizzo stringa src nel valore numerico in formato
// rete restituito all'indirizzo puntato da addr_ptr;

char *inet_ntop(int af, const void *addr_ptr, char *dest, size_t len)

//converte l'indirizzo puntato da addr_ptr dal formato rete in una stringa
// puntata da dest che deve essere non nullo e di lunghezza coerente con il
// tipo di indirizzo (si possono usare le costanti INET_ADDRSTRLEN per IPv4
// e INET6_ADDRSTRLEN per IPv6); con tale lunghezza deve anche essere
// valorizzato il parametro len.
```

Le lettere «n» (*numeric*) e «p» (*presentation*) nei nomi delle funzioni servono a ricordare il tipo di conversione fatta.

In entrambe c'è il parametro *af* che indica il tipo di indirizzo, e quindi può essere *AF\_INET* o *AF\_INET6*.

La prima funzione ritorna un valore positivo in caso di successo, nullo se l'indirizzo da convertire non è valido e negativo se il tipo di indirizzo *af* non è valido (in tal caso *errno* vale *ENOAFSUPPORT*).

La seconda funzione ritorna un puntatore alla stringa convertita in caso di successo oppure *NULL* in caso di errore con *errno* che allora vale:

- *ENOSPC*: le dimensioni della stringa sono maggiori di *len*;
- *ENOAFSUPPORT*: il tipo di indirizzo *af* non è valido.

Per entrambe le funzioni gli indirizzi vengono convertiti usando le rispettive strutture (*in\_addr* per IPv4 e *in6\_addr* per IPv6)) da preallocare e passare con il puntatore *addr\_ptr*.

## 2.6 Uso della funzione *ioctl()* con i socket

La funzione *ioctl()* viene utilizzata per gestire i dispositivi quando non sono sufficienti le comuni funzioni di gestione dei file (si ricorda che in GNU/Linux e Unix i dispositivi sono visti come file dal sistema).

L'uso della *ioctl()* diviene necessario quando si devono gestire caratteristiche specifiche dell'hardware; la sua sintassi dipende dal tipo di dispositivo e varia anche in funzione del sistema ospite introducendo problemi di portabilità per i programmi fra versioni diverse di Unix.

In questa sede ci limitiamo a vederne l'utilizzo in GNU/Linux per accedere a informazioni associate ad un socket di rete: in particolare per reperire la lista delle interfacce di rete della macchina con rispettivi indirizzi IP, MAC, NETMASK e BROADCAST.

La funzione *ioctl()*, definita in `<ioctl.h>`, in questo contesto ha il seguente prototipo:

```
int ioctl(int sd, int ric, struct xxx *p);

//sd è il socket creato in precedenza;
//ric è il tipo di operazione richiesta;
//ifc p è il puntatore a una struttura apposita
//      che dipende dal tipo di richiesta.
```

Ritorna 0 in caso di successo oppure -1 se c'è qualche errore.

Per interrogare il socket circa le interfacce presenti sulla macchina si usa la funzione nel modo seguente:

```
int ioctl(int sd, SIOCGIFCONF, struct ifconf *ifc);
```

La struttura *ifconf* è definita in `<net/if.h>` nel modo seguente:

```
struct ifconf {
    int    ifc_len;           // grandezza del buffer per i dati
    union {
        __caddr_t ifcu_buf;   // è un puntatore a carattere
        struct ifreq *ifcu_req;
    } ifc_ifcu;
};
#define ifc_buf    ifc_ifcu.ifcu_buf // puntatore al buffer
#define ifc_req    ifc_ifcu.ifcu_req // struttura alternativa
                                     // per accedere al buffer
```

Anche la struttura *ifreq* è definita in `<net/if.h>` in questo modo:

```
struct ifreq {
#define IFHWADDRLEN    6
#define IFNAMSIZ      16
    union {
        char ifrn_name[IFNAMSIZ]; // nome dell'interfaccia di rete
    } ifr_ifrn;
    union {
        struct sockaddr ifru_addr;
```

```

    struct sockaddr ifru_dstaddr;
    struct sockaddr ifru_broadaddr;
    struct sockaddr ifru_netmask;
    struct sockaddr ifru_hwaddr;
    short int ifru_flags;
    int ifru_ivalue;
    int ifru_mtu;
    struct ifmap ifru_map;
    char ifru_slave[IFNAMSIZ];
    char ifru_newname[IFNAMSIZ];
    __caddr_t ifru_data;
} ifr_ifru;

};
// Ci sono quindi due campi: uno per il nome dell'interfaccia,
// l'altro per il parametro di ritorno.
// Per accedere ai campi si possono utilizzare le seguenti macro:
#define ifr_name          ifr_ifrn.ifrn_name      // nome interfaccia
#define ifr_hwaddr        ifr_ifru.ifru_hwaddr    // indirizzo MAC
#define ifr_addr          ifr_ifru.ifru_addr      // indirizzo IP
#define ifr_dstaddr       ifr_ifru.ifru_dstaddrll // other end of p-p link
#define ifr_broadaddr     ifr_ifru.ifru_broadaddr // ind. BROADCAST
#define ifr_netmask       ifr_ifru.ifru_netmask   // NETMASK
#define ifr_flags         ifr_ifru.ifru_flags     // flags
#define ifr_metric        ifr_ifru.ifru_ivalue    // metric
#define ifr_mtu           ifr_ifru.ifru_mtu      // mtu
#define ifr_map           ifr_ifru.ifru_map       // device map
#define ifr_slave         ifr_ifru.ifru_slave     // slave device
#define ifr_data          ifr_ifru.ifru_data      // for use by interface
#define ifr_ifindex       ifr_ifru.ifru_ivalue    // interface index
#define ifr_bandwidth     ifr_ifru.ifru_ivalue    // link bandwidth
#define ifr_qlen          ifr_ifru.ifru_ivalue    // queue length
#define ifr_newname       ifr_ifru.ifru_newname   // new name

```

Per usare la funzione si definisce un area dati abbastanza ampia (ad esempio 1024 byte), si inserisce la sua lunghezza in *ifc\_len* e il puntatore ad essa in *ifc\_buf*; una volta invocata la funzione, in assenza di errori, abbiamo in *ifc\_len* la dimensione dei dati e nell'area puntata da *ifc\_buf* i nomi delle varie interfacce.

Per acquisire le caratteristiche di una qualche interfaccia di rete si deve poi utilizzare la funzione *ioctl()* con la richiesta opportuna e passando come parametro la struttura *ifreq* ottenuta dall'invocazione con richiesta *SIOCGIFCONF*.

Ad esempio:

```

int ioctl(int sd, SIOCGIFADDR, struct ifreq *ifr);
int ioctl(int sd, SIOCGIFBRDADDR, struct ifreq *ifr);
int ioctl(int sd, SIOCGIFNETMASK, struct ifreq *ifr);
int ioctl(int sd, SIOCGIFHWADDR, struct ifreq *ifr);

```

servono ad ottenere in *ifr* rispettivamente:

- indirizzo IP;

- indirizzo BROADCAST;
- NETMASK;
- indirizzo MAC.

I nomi delle richieste usate iniziano tutti con SIOCGIF che è l'acronimo di *Socket Input Output Control Get InterFace*; esistono anche analoghe richieste per il settaggio delle proprietà delle interfacce di rete il cui nome inizia con per SIOCSIF ( *Socket Input Output Control Set InterFace*).

Un esempio completo di utilizzo di queste funzioni si trova nel paragrafo 6.4.

<sup>1</sup> una copia di questo file, dovrebbe essere disponibile anche qui: *<allegati/progr-socket/esempio-01.c>*.

## 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 rispettivamente con i valori `IN6ADDR_ANY_INIT` e `IN6ADDR_LOOPBACK_INIT`.

## 3.2 Connessione

La connessione di un cliente TCP ad un server 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 server ha ricevuto 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 transitoria 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 server 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 server, 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 server 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 fondamentale 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.

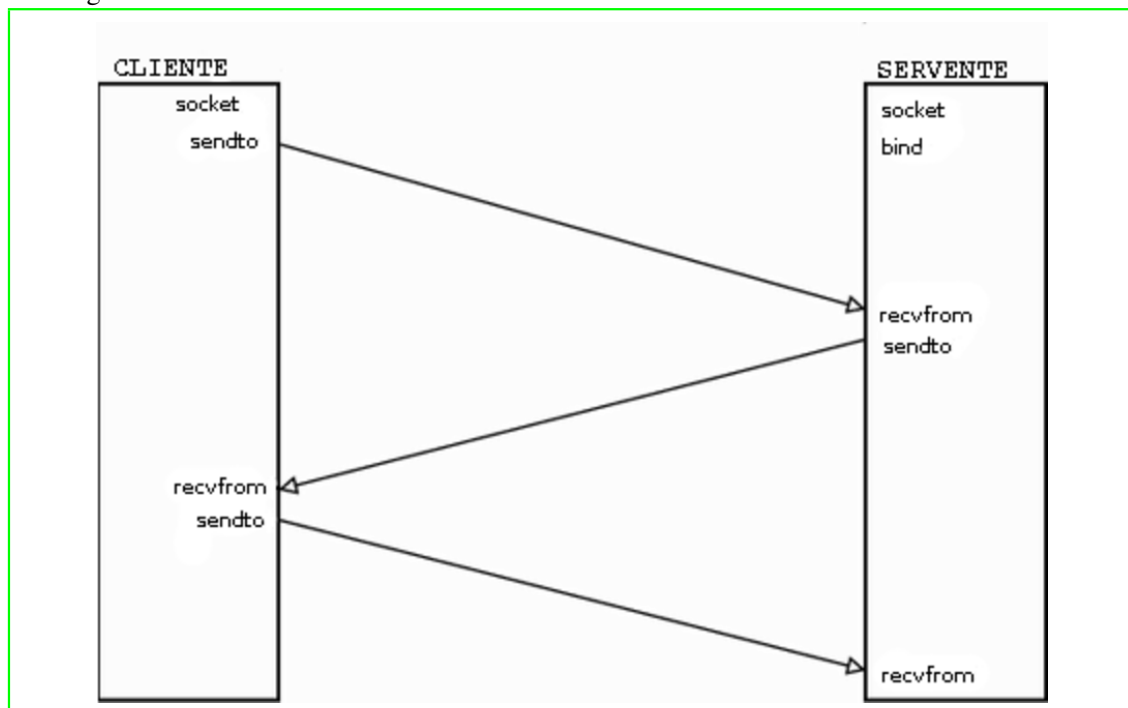
Quindi i socket UDP devono essere aperti con la funzione *socket* utilizzando per lo stile di comunicazione il valore *SOCK\_DGRAM*; inoltre, non esistendo il concetto di connessione non è ovviamente necessario alcun meccanismo di attivazione e non servono le funzioni *connect()*, *listen()* e *accept()*.

La funzione *bind()* invece serve ancora:

- ad entrambi i processi che comunicano, se si tratta di una comunicazione paritetica;
- solo al server se c'è ancora la presenza di un cliente e un server.

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

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 *sendto()* e riceve risposte con *recvfrom()* da un server UDP; se qualche pacchetto di dati inviati o di risposta si perde o se il server 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 server 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 viene 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 efficiente 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 server 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 server, 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 server 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 associato a un socket; ad esempio in un processo cliente per conoscere IP e porta assegnati automaticamente dal *kernel* dopo la *bind()* oppure in un server 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 server 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 assegnato 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 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.

## Multiplexing dell'input/output

In caso si abbiano più canali di comunicazione (in rete e/o locali) aperti c'è il problema di controllarli contemporaneamente; si pensi ad esempio a un server in attesa di dati in ingresso da vari clienti oppure a un processo che riceve dati sia da un utente, attraverso la tastiera (*standard input*), sia da un processo remoto.

Può succedere di rimanere bloccati in una operazione su un descrittore di file o di socket non pronto mentre ce ne potrebbe essere un altro disponibile, ritardando l'accesso a quest'ultimo e con il rischio di incorrere in un *deadlock* (se l'accesso al descrittore in attesa può essere sbloccato solo da qualcosa in arrivo su quello disponibile).

Una soluzione abbastanza semplice sarebbe quella di rendere l'accesso ai socket non bloccante in modo che le relative funzioni ritornino subito (con errore EAGAIN) e prevedere un meccanismo ciclico di interrogazione di tutti i flussi dati verso il nostro processo.

Questa soluzione, chiamata *polling*, usa in modo molto inefficiente le risorse del sistema in quanto la maggior parte del tempo macchina è perso ad interrogare flussi che non hanno dati.

Molto migliore è la soluzione chiamata *I/O multiplexing* con la quale si controllano vari flussi contemporaneamente, permettendo di bloccare un processo quando le operazioni volute non sono possibili, e di riprenderne l'esecuzione una volta che almeno una sia disponibile.

Il tutto viene realizzato con l'uso della funzione *select()* che ha il seguente prototipo:

```
int select(int n, fd_set *rfd, fd_set *wfd, fd_set *exfd, struct timeval *tim)

//attende che uno dei descrittori di file degli insiemi indicati diventi attivo;
//n è la dimensione degli insiemi di descrittori (sizeof(fd_set));
//rfd è l'insieme dei descrittori da controllare in lettura;
//wfd è l'insieme dei descrittori da controllare in scrittura;
//exfd è l'insieme dei descrittori da controllare per verifica di errori;
//tim è il tempo dopo il quale la funzione ritorna se nessun flusso è attivo.
```

Ritorna il numero di file descriptor, eventualmente anche 0, che sono attivi, oppure -1 se c'è errore, nel qual caso *errno* può valere:

- EBADF: uno degli insiemi di file descriptor è sbagliato;
- EINTR: la funzione è stata interrotta da un segnale;
- EINVAL: *n* è negativo o *tim* ha un valore non valido.

La funzione mette il processo in stato di *sleep*, finché almeno uno dei descrittori degli insiemi specificati non diventa attivo, per un tempo massimo specificato da *timeout*.

Per valorizzare il parametro *n* si può usare la costante *FD\_SETSIZE* definita in `<sys/select.h>`.

Gli insiemi di descrittori possono essere vuoti nel caso non si voglia controllare nulla in lettura, o in scrittura o per gli errori; anche il parametro *tim* può essere NULL con il significato di attesa indefinita da parte della funzione.

Il *timeout tim* viene indicato valorizzando i due campi della struttura *timeval*:

```
int tv_sec

//secondi di attesa
```

```
int tv_usec  
  
//microsecondi di attesa
```

Per la selezione dei descrittori la funzione usa il «file descriptor set» corrispondente al tipo dati *fd\_set*.

La gestione degli insiemi di descrittori è resa possibile dalle seguenti macro di preprocessore, definite in `<sys/select.h>`:

```
FD_ZERO(fd_set *set)  
  
//inizializza l'insieme set (vuoto);  
  
FD_SET(int fd, fd_set *set)  
  
//aggiunge fd all'insieme set;  
  
FD_CLR(int fd, fd_set *set)  
  
//elimina fd dall'insieme set;  
  
FD_ISSET(int fd, fd_set *set)  
  
//controlla se fd è presente nell'insieme set.
```

Si deve prestare attenzione al fatto che la *select()* altera i valori degli insiemi di descrittori che quindi devono essere sempre impostati se si invoca la funzione ciclicamente.

La funzione modifica anche il valore di *tim*, impostandolo al tempo restante in caso di interruzione prematura; questo è utile perché permette di non ricalcolare il tempo rimanente quando la funzione, interrotta da un segnale (errore EINTR), deve essere invocata di nuovo.

Segue un frammento di codice con un esempio di uso della funzione *select()* dove si suppone di dover controllare due flussi di input corrispondenti alla tastiera e a un socket di rete *sd*:

```
fd_set set, set2, set3;  
while (1)  
{  
    FD_ZERO (&set);  
    FD_ZERO (&set2);  
    FD_ZERO (&set3);  
    FD_SET(fileno(stdin), &set); // con macro fileno ho il descrittore di stdin  
    FD_SET(sd, &set);  
    if (select(FD_SETSIZE, &set, &set2, &set3, NULL)<0) {  
        // istruzioni di gestione dell'errore della select  
        exit(1);  
    }  
    if (FD_ISSET(fileno(stdin), &set)) {  
        // istruzione di gestione dell'input da tastiera  
    }  
    if (FD_ISSET (sd, &set)) {
```

```
        // istruzione di gestione dell'input dal socket
    }
} // fine while(1)
// le due ultime if non devono essere esclusive, possono
// arrivare dati contemporaneamente dai due flussi
```

## Risoluzione dei nomi

La comunicazione tra i processi in rete è possibile se si specificano gli indirizzi dei due nodi che comunicano; come detto in precedenza essi sono costituiti da coppie IP:porta.

Quindi è fondamentale conoscere gli indirizzi IP dei nodi in cui vengono eseguiti i processi e per questo possono essere molto utili le funzioni che permettono di risalire a tali informazioni partendo dal nome delle macchine, cioè quelle che eseguono la «risoluzione dei nomi».

Tali funzioni sono definite in `<netdb.h>` insieme alle strutture dati che utilizzano.

### 5.1 Nome della macchina ospite

La prima funzione esaminata in verità non riguarda la risoluzione dei nomi ma può essere comunque utile perché permette di conoscere il nome della macchina in cui il processo che la invoca è in esecuzione; si tratta della funzione `gethostname()` definita in `<unistd.h>` e il cui prototipo è:

```
int gethostname(char *nome, size_t lun)

//nome è il nome che si ottiene in risposta;
//lun dimensione della variabile nome.
```

Ritorna 0 se non ci sono errori, altrimenti -1 (ad esempio se *nome* è un puntatore non valido).

### 5.2 Ricavare l'indirizzo IP dal nome

Ci sono diverse funzioni utili a risolvere il nome in un indirizzo IP; iniziamo da `gethostbyname()` che ha questo prototipo:

```
struct hostent *gethostbyname(const char *nome)

//trova l'indirizzo associato al nome nome.
```

Ritorna il puntatore ad una struttura di tipo *hostent* con i dati associati al nome, oppure un puntatore nullo in caso di errore, nel qual caso la variabile esterna *h\_errno* può valere:

- `HOST_NOT_FOUND` o `NO_DATA`: dati non trovati;
- `TRY_AGAIN`: errore temporaneo, è possibile riprovare;

mentre *errno* può valere:

- `EINTR`: funzione interrotta da un segnale, è possibile riprovare;
- `ENETDOWN`: il sottosistema di rete non è attivo.

La struttura *hostent* è definita nel seguente modo:

```
struct hostent {
    char    *h_name;           // nome ufficiale dell'host
    char    **h_aliases;       // lista degli alias
```

```

int    h_addrtype;    // tipo di indirizzo
int    h_length;      // lunghezza dell'indirizzo
char   **h_addr_list; // lista degli indirizzi
}
#define h_addr  h_addr_list[0] // primo indirizzo della lista,
                               // serve per motivi di compatibilità

```

La funzione `gethostbyname()` interroga un server DNS oppure usa il contenuto del file `/etc/hosts` a seconda di quanto indicato in `/etc/resolv.conf` e valorizza i campi della struttura *hostent* in questa maniera:

- *h\_name*: nome canonico (nel caso del DNS è il nome associato ad un record «A»);
- *h\_aliases*: puntatore a un vettore di puntatori, terminato da un puntatore nullo, ciascuno dei quali punta ad una stringa contenente uno dei possibili alias del nome canonico (nel DNS sono i record «CNAME»);
- *h\_addrtype*: tipo di indirizzo restituito, `AF_INET`;
- *h\_length*: lunghezza dell'indirizzo;
- *h\_addr\_list*: puntatore a un vettore di puntatori, terminato da un puntatore nullo, ciascuno dei quali punta a un singolo indirizzo.

La funzione `gethostbyname()` permette di ottenere solo indirizzi IPv4; per ottenere anche indirizzi IPv6 si utilizza `gethostbyname2()` che ha il seguente prototipo:

```

struct hostent *gethostbyname2(const char *nome, int af)

//trova l'indirizzo di tipo af associato a nome.

```

Questa funzione è del tutto analoga alla precedente; l'unica differenza è il parametro *af* da impostare come `AF_INET` o `AF_INET6` per stabilire il tipo di indirizzi che si vuole in risposta.

Occorre ricordare infine che esistono anche le funzioni `gethostbyname_r()` e `gethostbyname2_r()` che sono le versioni rientranti delle due funzioni appena esaminate e delle quali non vengono forniti dettagli.

## 5.3 Risoluzione inversa

Per ottenere il nome di una stazione se è noto il suo indirizzo IPv4 o IPv6 si può usare la funzione `gethostbyaddr()` che ha il seguente prototipo:

```

struct hostent *gethostbyaddr(const char *ind, int lun, int tipo)

//trova il nome associato all'indirizzo ind;
//lun è la dimensione dell'indirizzo (4 o 16);
//tipo è il tipo di indirizzo (AF_INET o AF_INET6).

```

Ritorna il puntatore a una struttura *hostent* in caso di successo o `NULL` in caso di errore; in quest'ultimo caso valgono gli stessi codici per *h\_errno* e *errno* visti per la funzione `gethostbyname()`



con in più, per *errno*, il valore EFAULT (*ind* puntatore non valido).

L'unico campo della struttura *hostent* da considerare è *h\_name* contenente il nome cercato; la funziona comunque valorizza anche il primo campo di *h\_addr\_list* con l'indirizzo *ind*.

Si deve notare che, malgrado nel prototipo *ind* sia un puntatore a carattere, esso deve essere definito come struttura *in\_addr* (o *in6\_addr*) su cui fare il *casting* a puntatore a carattere.

## 5.4 Risoluzione con connessioni TCP

La risoluzione dei nomi tramite un servente DNS viene effettuata solitamente con il protocollo UDP (sulla porta 53); nel caso si voglia invece ricorrere al TCP si può utilizzare la funzione *sethostent()* il cui prototipo è:

```
void sethostent(int val)

//con val=1 attiva l'uso di connessioni TCP per interrogare il DNS.
```

Per disattivare l'uso del TCP si usa invece *endhostent()* così definita:

```
void endhostent(void)

//disattiva l'uso di connessioni TCP per interrogare il DNS.
```

## 5.5 Nomi dei servizi noti

Nel caso sia utile conoscere il numero di porta associato ad un servizio di rete o viceversa si possono usare le due funzioni *getservbyname()* e *getservbyport()* con i seguenti prototipi:

```
struct servent *getservbyname(const char *nome, const char *proto)

//restituisce la porta associata al servizio nome.

struct servent *getservbyport(int porta, const char *proto)

//restituisce il nome di servizio associato a porta.
```

Ritornano il puntatore ad una struttura *servent* in caso di successo, oppure NULL se c'è errore.

In entrambe c'è l'argomento *proto* che indica il protocollo su cui effettuare la ricerca e che può essere *udp*, *tcp* o NULL, nel qual caso la ricerca viene fatta su un protocollo qualsiasi.

Le due funzioni si servono di quanto contenuto nel file *'/etc/services'* per elaborare la risposta che viene restituita nella struttura *servent* così definita:

```
struct servent {
    char    *s_name;           // nome del servizio
    char    **s_aliases;       // lista di nomi ulteriori
    int     s_port;            // porta
    char    *s_proto;          // protocollo
}
```

## Esempi vari

Nei prossimi paragrafi vengono illustrati alcuni programmi che gestiscono dei socket di rete.

Questi esempi non hanno alcuna pretesa di completezza e non sono sicuramente ottimizzati per quanto riguarda l'interfaccia, i controlli sui possibili errori, l'utilizzo delle risorse; d'altra parte l'obiettivo non è la creazione di applicazioni di rete da utilizzare concretamente (di quelle ce ne sono già moltissime, libere, e spesso, anche ben fatte) ma solo di fornire degli spunti utili a fini didattici.

Nei programmi viene utilizzata la libreria «utilsock.h» contenente le inclusioni delle varie librerie necessarie al loro funzionamento e le definizioni delle funzioni personalizzate di lettura e scrittura sui socket di rete.

Il contenuto di «utilsock.h» è il seguente:<sup>1</sup>

```
/* File:          utilsock.h
 * Autore:       FF
 * Data:         05/02/2006
 * Descr.:       Inclusione librerie e definizione funzioni utili per i socket
 */
#include <unistd.h>          // interfaccia Unix standard
#include <errno.h>           // codici di errore
#include <sys/types.h>       // tipi predefiniti
#include <arpa/inet.h>       // per convertire ind. IP
#include <sys/socket.h>      // socket
#include <stdio.h>           // i/o
#include <stdlib.h>          // utilita' standard
#include <string.h>          // stringhe
#include <fcntl.h>           // file a basso livello
#include <time.h>            // tempo
// Funzioni
int leggisock(int fd, void *buf, size_t conta)
/*
 * legge "conta" byte da un descrittore
 * da usare al posto di read quando fd e' un socket di tipo stream
 */
{
    int mancano, letti;
    mancano=conta;
    while (mancano>0) {
        letti = read(fd, buf, mancano);
        if ( (letti<0) && (errno!=EINTR) )
            return(letti);          // errore
        else
            if (letti==0)
                break;              // EOF
            mancano = mancano - letti;
            buf = buf + letti;
    }
    return(conta - mancano);
}

int scrivisock(int fd, void *buf, size_t conta)
/*
 * scrive "conta" byte su un descrittore
 */
```

```

* da usare al posto di write quando fd e' un socket di tipo stream
*/
{
    int mancano, scritti;
    mancano=conta;
    while (mancano>0) {
        scritti = write(fd, buf, mancano);
        if ( (scritti<=0) && (errno!=EINTR) )
            return(scritti);          // errore
        mancano = mancano - scritti;
        buf = buf + scritti;
    }
    return(conta - mancano);
}

```

## 6.1 Servente concorrente che riceve un file da un cliente connesso

Questo esempio consiste in un processo servente che accetta connessioni da un cliente e riceve da quest'ultimo il contenuto di un file e lo memorizza in un file di output.

Nel primo listato abbiamo il codice del servente che deve essere lanciato fornendo come parametro la porta da ascoltare; dopo aver creato il file di log il programma crea il socket, prepara l'indirizzo con le necessarie conversioni di formato e esegue la *bind()* e la *listen()*.

Il cuore del programma inizia al momento della chiamata alla *accept()* che crea il socket connesso; qui il programma crea un figlio (con la funzione *fork()*) e si rimette in ascolto, mentre il figlio gestisce la connessione svolgendo le seguenti elaborazioni:

- definisce delle stringhe basate su data e ora corrente da usare nel file di log e per il nome del file di output;
- aggiorna il file di log;
- invia al cliente un messaggio di conferma;
- crea il file di output;
- legge dal socket i dati inviati dal cliente e li scrive nel file.<sup>2</sup>

```

/* Programma:      servente_c
* Autore:         FF
* Data:           05/02/2006
* Descr.:         Esempio di servente concorrente TCP che accetta connessione,
*                  la registra su un file di log e riceve un file dal cliente da
*                  memorizzare nella directory corrente con nome composto
*                  da ip_cliente:porta_data_ora;
*/
#include "utilsock.h"
/* utilsock.h contiene tutte le inclusioni delle altre librerie necessarie
* e le funzioni di lettura e scrittura dal socket
*/

```

```

#define MAX      80
#define BACKLOG  20

int main(int argc, char *argv[])
{
    int sd, accsd, fd, fdw, l, scritti, letti;
    struct sockaddr_in serv_ind, cliente;
    char buffer[MAX], datil[MAX], indcli[128];
    socklen_t lung;
    pid_t pid;
    in_port_t porta;
    char stringa[MAX], stringa2[MAX];
    time_t    now;
    struct    tm    *tnow;          // struttura definita in <time.h>
    char tempo[24], tempo2[24];
    if (argc != 2) {
        printf("\nErrore\n");
        printf("Attivare con ./servente_c porta(>1023)\n");
        exit(-1);
    }
    // apertura file di log con xor fra i flag O_RDWR, O_CREAT, O_APPEND
    // definiti in <fcntl.h>; apre in lettura-scrittura, append e lo
    // crea se serve
    fd=open("log_servente", O_RDWR | O_CREAT | O_APPEND, 0644);
    // crea socket
    if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Errore nella creazione del Socket");
        exit(-1);
    }
    // indirizzo ip
    memset((void *)&serv_ind, 0, sizeof(serv_ind)); // pulizia ind
    serv_ind.sin_family = AF_INET;                  // ind di tipo INET
    porta = atoi(argv[1]);
    serv_ind.sin_port = htons(porta);                // porta
    serv_ind.sin_addr.s_addr = htonl(INADDR_ANY);    // connessioni da ogni ip
    // bind
    if (bind(sd, (struct sockaddr *)&serv_ind, sizeof(serv_ind)) < 0) {
        perror("Errore nella bind");
        exit(-1);
    }
    // listen
    if (listen(sd, BACKLOG) < 0 ) {
        perror("Errore nella listen");
        exit(-1);
    }
    printf ("Servente in ascolto sulla porta %d\n", porta);
    printf ("'^Ctrl+c' per chiuderlo\n\n");
    lung = sizeof(struct sockaddr_in); //lung della giusta dimensione
    // accetta connessione
    while (1) {
        if ((accsd = accept(sd, (struct sockaddr *)&cliente, &lung)) < 0) {
            perror("Errore nella accept");
            exit(-1);
        }
    }
}

```

```

    if ((pid=fork()) <0) {
        perror("Errore nella fork");
        exit(-1);
    }
    if (pid == 0) {          // figlio
        close(sd);
        time (&now);
        tnow = localtime(&now);
        strftime(tempo,24,"%b %d %H:%M:%S %Y",tnow);
        strftime(tempo2,24,"%Y_%b_%d_%H:%M:%S",tnow);
        inet_ntop(AF_INET, &cliente.sin_addr, indcli, sizeof(indcli));
        l=strlen(indcli);
        snprintf(stringa,55+l,"%s Richiesta da nodo %s, porta %5d\n",
                    tempo,indcli,ntohs(cliente.sin_port));
        snprintf(stringa2,30+l,"%s:%5d_%s",indcli,ntohs(cliente.sin_port),
                    tempo2);
        printf("%s",stringa);
        write (fd,stringa,strlen(stringa));          // scrive nel file di log
        strcpy(buffer,"Connessione accettata\n");
        if ( (scrivisock(accsd, buffer, strlen(buffer))) < 0 ) {
            perror("Errore nella write");
            exit(-1);
        }
        // apertura file di output con xor fra i flag O_RDWR, O_CREAT
        // lettura-scrittura e lo crea se serve
        fdw=open(stringa2, O_RDWR | O_CREAT | O_APPEND,0644);
        // scrive nel file di output i dati letti dal socket
        // per leggere usa leggisock() definita in utilsock.h
        while(1) {
            if((letti=leggisock(accsd,datil,MAX))==0)
                break;
            scritti=write(fdw,datil,letti);
        }
        close(accsd);
        exit(0);
    }
    else {                  // padre
        close(accsd);
    }
}
close(fd);
close(fdw);
exit(0);
}

```

Nel secondo listato abbiamo il codice del cliente che deve essere lanciato fornendo nell'ordine:

- indirizzo IP del server;
- porta in cui il server è in ascolto;
- nome del file da trasferire.

Il programma verifica che il file da trasferire esista, quindi crea il socket e definisce l'indirizzo con le necessarie conversioni di formato; infine, con *connect()*, stabilisce la connessione dalla quale legge il messaggio di conferma inviato dal servente e nella quale scrive i dati letti dal file da trasferire.<sup>3</sup>

```

/* Programma:      cliente.c
 * Autore:         FF
 * Data:           05/02/2006
 * Descr.:         Esempio di cliente TCP che chiede una connessione
 *                 riceve risposta e termina
 */
#include "utilsock.h"
/* utilsock.h contiene tutte le inclusioni delle altre librerie necessarie
 * e le funzioni di lettura e scrittura dal socket
 */
#define MAX        80

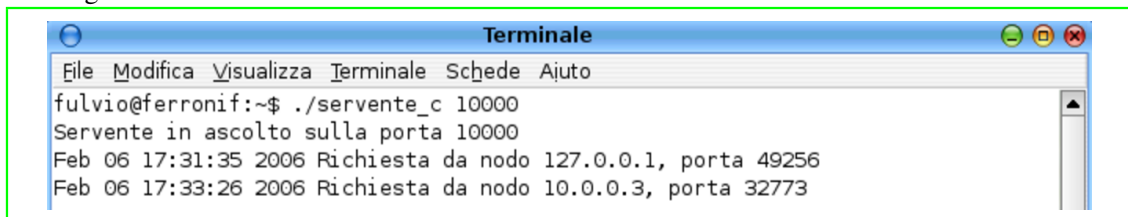
int main(int argc, char *argv[])
{
    int sd,n,i,fd, scritti,letti;
    struct sockaddr_in serv_ind;
    in_port_t porta;
    char buffer[MAX], datil[MAX];
    // controlla argomenti
    if (argc !=4) {
        printf("\nErrore\n");
        printf("Esempio di connessione per trasferire file: \n");
        printf("./client ip_servente porta (>1023) nome_file\n");
        exit(-1);
    }
    // apre file per verificarne esistenza
    if ( (fd=open(argv[3], O_RDONLY))<0) {    // O_RDONLY flag def. in <fcntl.h>
        perror("Errore in apertura file da trasferire");
        exit(-1);
    }
    // crea socket
    if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Errore in creazione socket");
        exit(-1);
    }
    // indirizzo IP
    memset((void *)&serv_ind, 0, sizeof(serv_ind)); // pulizia ind
    serv_ind.sin_family = AF_INET;                  // ind di tipo INET
    porta = atoi(argv[2]);
    serv_ind.sin_port = htons(porta);                // porta a cui collegarsi
    /* crea indirizzo usando inet_pton */
    if ( (inet_pton(AF_INET, argv[1], &serv_ind.sin_addr)) <= 0) {
        perror("Errore in creazione indirizzo");
        return -1;
    }
    // stabilisce la connessione
    if (connect(sd, (struct sockaddr *)&serv_ind, sizeof(serv_ind)) < 0) {
        perror("Errore nella connessione");
        exit(-1);
    }
}

```

```
// legge dal servernte
if ( (n=read(sd, buffer, MAX))<=0) {
    perror("Errore nella read");
    exit(-1);
}
buffer[n]='\0';
if (fputs(buffer, stdout) == EOF) {
    perror("Errore nella fputs");
    exit(-1);
}
// legge i dati dal file da trasferire e li scrive sul socket
// usa la funzione scrivisock() di utilsock.h
while(1) {
    letti=read(fd,datil,MAX);
    if(letti==0) {
        break;
    }
    scritti=scrivisock(sd,datil,letti);
    if (scritti<letti) {
        printf("Errore in scrittura sul socket\n");
    }
}
printf("File %s trasferito\n",argv[3]);
close (sd);
close (fd);
exit(0);
}
```

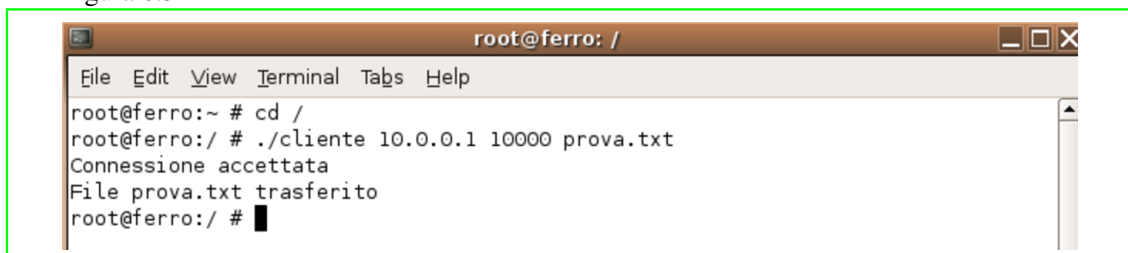
Nelle figure seguenti vediamo l'esecuzione dei due programmi: in figura 6.4 il servernte, dopo che ha ricevuto alcune richieste di connessione, in figura 6.5 il cliente.

Figura 6.4



```
Terminale
File Modifica Visualizza Terminale Schede Aiuto
fulvio@ferronif:~$ ./servente_c 10000
Servente in ascolto sulla porta 10000
Feb 06 17:31:35 2006 Richiesta da nodo 127.0.0.1, porta 49256
Feb 06 17:33:26 2006 Richiesta da nodo 10.0.0.3, porta 32773
```

Figura 6.5



```
root@ferro: /
File Edit View Terminal Tabs Help
root@ferro:~ # cd /
root@ferro:/ # ./cliente 10.0.0.1 10000 prova.txt
Connessione accettata
File prova.txt trasferito
root@ferro:/ #
```

## 6.2 Servente iterativo che scambia messaggi con un cliente

Questo esempio consiste in un processo servente che accetta una connessione da un cliente per scambiare messaggi con quest'ultimo.

La comunicazione è *half-duplex* nel senso che ognuno dei due nodi invia un messaggio e si pone in attesa della risposta, ricevuta la quale può inviare un nuovo messaggio; il nodo che deve iniziare la comunicazione è il servente.

Nel primo listato c'è il servente che deve essere lanciato indicando la porta in cui si mette in ascolto.

Dopo aver eseguito le funzioni di attivazione e ascolto del socket, al verificarsi della connessione, cioè dopo la *accept()*, esegue un ciclo nel quale scambia messaggi con il cliente connesso finché uno dei due nodi non invia la stringa «/ciao».

La differenza importante rispetto all'esempio precedente è nel fatto che qui non viene creato alcun processo figlio ed è il processo lanciato che si occupa direttamente di gestire la connessione (servente iterativo anziché concorrente).<sup>4</sup>

```
/* Programma:      servente_i_msg
 * Autore:         FF
 * Data:           06/02/2006
 * Descr.:         Esempio di servente iterativo TCP che accetta una connessione
 *                  da un cliente e scambia messaggi con esso
 */
#include "utilsock.h"
/* utilsock.h contiene tutte le inclusioni delle altre librerie necessarie
 * e le funzioni di lettura e scrittura dal socket
 */
#define MAX        256
#define BACKLOG    20

int main(int argc, char *argv[])
{
    int sd, accsd, l;
    struct sockaddr_in serv_ind, cliente;
    char msg[MAX], indcli[128];
    int msglun=MAX, val=1;
    socklen_t lung;
    in_port_t porta;
    printf("\033[2J\033[H");
    // controlla argomenti
    if (argc !=2) {
        printf("\nErrore\n");
        printf("Si deve digitare: \n");
        printf("./servente_i_msg porta (>1023)\n");
        printf("Premere Invio\n");
        getchar();
        exit(-1);
    }
    // crea socket
    if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Errore nella creazione del Socket");
        exit(-1);
    }
}
```



```

}
// rende il socket riusabile
setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
// indirizzo ip
memset((void *)&serv_ind, 0, sizeof(serv_ind)); // pulisce ind
serv_ind.sin_family = AF_INET;                  // ind di tipo INET
porta=atoi(argv[1]);
serv_ind.sin_port = htons(porta);                // scelgo porta non priv.
serv_ind.sin_addr.s_addr = htonl(INADDR_ANY);    // connessioni da ogni ip
// bind socket
if (bind(sd, (struct sockaddr *)&serv_ind, sizeof(serv_ind)) < 0) {
    perror("Errore nella bind");
    exit(-1);
}
// listen
if (listen(sd, BACKLOG) < 0 ) {
    perror("Errore nella listen");
    exit(-1);
}
printf("In ascolto sulla porta %d\n",porta);
// accetta connessione
if ( (accsd = accept(sd, (struct sockaddr *)&cliente, &lung)) <0 ) {
    perror("Errore nella accept");
    exit(-1);
}
inet_ntop(AF_INET, &cliente.sin_addr, indcli, sizeof(indcli));
printf("Cliente %s connesso\n",indcli);
printf("Per terminare digitare il messaggio: /ciao\n");
do {
    printf("Messaggio da inviare: ");
    for(l=0; ((msg[l]=getchar())!='\n' && l<MAX-1);l++);
    msg[l]='\0';
    scrivisock(accsd,msg,l+1);
    printf("OK messaggio inviato\n");
    printf("In attesa di messaggio\n\n");
    if ( (read(accsd,msg,msglung))<=0 ) {
        perror("Errore nella read");
        exit(-1);
    }
    printf("Messaggio ricevuto: %s\n",msg);
}
while(strcmp(msg,"/ciao")); // usare /ciao per finire
close(accsd);
close(sd);
exit(0);
}

```

Nel prossimo listato abbiamo il cliente che deve essere lanciato fornando il nome del servente e la porta a cui collegarsi.

L'indirizzo IP da utilizzare per la connessione viene determinato in base al nome del servente con la funzione *gethostbyname()*.

Dopo la connessione viene eseguito il ciclo di scambio dei messaggi con l'altro nodo fino alla digitazione di «/ciao».<sup>5</sup>

```

/* Programma:      cliente_msg
* Autore:          FF
* Data:            06/02/2006
* Descr.:          Esempio di cliente TCP che chiede una connessione
*                  e scambia messaggi con un server del quale
*                  determina l'IP conoscendo il nome
*/
#include "utilsock.h"
/* utilsock.h contiene tutte le inclusioni delle altre librerie necessarie
* e le funzioni di lettura e scrittura dal socket
*/
#include <netdb.h>
#define MAX        256

int main(int argc, char *argv[])
{
    int sd, l;
    struct sockaddr_in serv_ind;
    struct hostent *ip_serv;
    in_port_t porta;
    char msg[MAX];
    int msglen=MAX;
    printf("\033[2J\033[H");
    // controlla argomenti
    if (argc != 3) {
        printf("\nErrore\n");
        printf("Esempio di connessione: \n");
        printf("./client nome_server porta (>1023)\n");
        printf("Premere Invio\n");
        getchar();
        exit(-1);
    }
    ip_serv=gethostbyname(argv[1]);    // trova IP server in base al nome
    // crea socket
    if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Errore in creazione socket");
        exit(-1);
    }
    // indirizzo IP
    memset((void *)&serv_ind, 0, sizeof(serv_ind)); // pulizia ind
    serv_ind.sin_family = AF_INET;                  // ind di tipo INET
    porta = atoi(argv[2]);
    serv_ind.sin_port = htons(porta);                // porta a cui collegarsi
    memcpy(&serv_ind.sin_addr.s_addr, ip_serv->h_addr, ip_serv->h_length);
    // copiato ip server dalla struct ip_serv a serv_ind.sin_addr
    // stabilisce la connessione
    if (connect(sd, (struct sockaddr *)&serv_ind, sizeof(serv_ind)) < 0) {
        perror("Errore nella connessione");
        exit(-1);
    }
    printf("Connessione stabilita con %s\n", inet_ntoa(serv_ind.sin_addr));
    printf("Per terminare digitare il messaggio: /ciao\n");
}

```

```

printf("In attesa di messaggio\n\n");
do {
    if ( (read(sd,msg,msglen))<=0 ) {
        perror ("Errore nella read");
        exit(-1);
    }
    printf("Messaggio ricevuto: %s\n",msg);
    if (strcmp(msg,"/ciao")){
        printf("Messaggio da inviare: ");
        for(l=0; ((msg[l]=getchar())!='\n' && l<MAX-1);l++);
        msg[l]='\0';
        scrivisock(sd,msg,l+1);
        printf("OK messaggio inviato\n");
        printf("In attesa di messaggio\n\n");
    }
}
while (strcmp(msg,"/ciao"));          // msg /ciao per finire
close(sd);
exit(0);
}

```

Nelle figure 6.8 e 6.9 viene mostrato un esempio di dialogo tra il server e il cliente.

Figura 6.8

```

Terminale
File Modifica Visualizza Terminale Schede Aiuto
fulvio@ferronif:~/lavoro/prove-mie/c/socket$ ./servente_i_msg 10000

In ascolto sulla porta 10000
Cliente 10.0.0.3 connesso
Per terminare digitare il messaggio: /ciao
Messaggio da inviare: primo messaggio
OK messaggio inviato
In attesa di messaggio

Messaggio ricevuto: ok ricevuto
Messaggio da inviare: secondo e ultimo
OK messaggio inviato
In attesa di messaggio

Messaggio ricevuto: /ciao
fulvio@ferronif:~/lavoro/prove-mie/c/socket$

```

Figura 6.9

```

root@ferro: /
File Edit View Terminal Tabs Help
root@ferro:/ # ./cliente_msg ferronif.max.planck 10000

Connessione stabilita con 10.0.0.1
Per terminare digitare il messaggio: /ciao
In attesa di messaggio

Messaggio ricevuto: primo messaggio
Messaggio da inviare: ok ricevuto
OK messaggio inviato
In attesa di messaggio

Messaggio ricevuto: secondo e ultimo
Messaggio da inviare: /ciao
OK messaggio inviato
In attesa di messaggio

root@ferro:/ #

```

## 6.3 Servente di *chat* con *multiplexing*

In questo esempio abbiamo un servente di *chat* che accetta connessioni da vari clienti, riceve i dati da uno qualsiasi di essi e li invia a tutti gli altri.

Per gestire l'input proveniente da nodi diversi viene utilizzato il *multiplexing* sul socket di rete in modo da poter leggere i dati da uno qualsiasi dei clienti collegati.

Ovviamente c'è anche il programma cliente da utilizzare per collegarsi al servente e «partecipare» alla *chat*; anche qui viene usato il *multiplexing* ma per uno scopo diverso e cioè per controllare contemporaneamente l'input dalla tastiera e dal socket di rete.

Nel primo listato c'è il servente che deve essere lanciato indicando la porta in cui si mette in ascolto.

Il socket viene reso riutilizzabile con la funzione *setsockopt()*, quindi, dopo la *bind()* e la *listen()* viene preparata la lista dei descrittori da utilizzare con la *select()*.

A questo punto il programma entra in un ciclo infinito (che si interrompe con *Ctrl+c*) nel quale «ascolta» i socket attivi.

Inizialmente non c'è alcun cliente collegato; via via che si collegano viene eseguita la *accept()* e il descrittore del socket connesso entra a far parte della lista dei descrittori gestita dalla *select()*; quando invece un cliente si disconnette, il relativo descrittore viene tolto dalla lista.

Sempre grazie alla *select()* i dati inviati da un certo nodo vengono recepiti e inviati a tutti gli altri nodi connessi.<sup>6</sup>

```
/* Programma:      servente_chat
 * Autore:         FF
 * Data:           08/02/2006
 * Descr.:         Esempio di servente di chat che riceve connessioni
 *                 e le gestisce con la funzione select (multiplexing) invece
 *                 di creare un figlio per ogni connessione; tutto quello che
 *                 inviato da un cliente viene rispedito dal servente
 *                 a tutti gli altri clienti
 */
#include "utilsock.h"
/* utilsock.h contiene tutte le inclusioni delle altre librerie necessarie
 * e le funzioni di lettura e scrittura dal socket
 */
#include <sys/select.h>
#include <netdb.h>
#define MAX        256
#define BACKLOG 20

int main(int argc, char *argv[])
{
    fd_set pri;                // pri lista di descrittori per select()
    fd_set set_lettura;        // temp altra lista di descrittori
    struct sockaddr_in indserv; // indirizzo del servente
    struct sockaddr_in indcli;  // indirizzo di un cliente
    int sdgr;                  // numero descrittore piu' alto
    int sd;                    // socket
    int csd;                    // socket connesso con accept()
    char buf[MAX],buf2[MAX];    // buffer dei dati
    int bytel;
```

```

in_port_t porta;
int val=1;                                // per la setsockopt()
socklen_t lind;
struct hostent *cli;                       // per trovare nome cliente dall'IP
struct in_addr vet[MAX];                   // per mem. IP cliente (max 256 clienti)
int i, j;
printf("\033[2J\033[H");
//controlli sui parametri
if (argc!=2) {
    printf("Digitare ./servente_chat porta\n");
    exit(-1);
}
porta=atoi(argv[1]);
FD_ZERO(&pri);                             // pulizia liste descrittori
FD_ZERO(&set_lettura);
// socket
if ((sd = socket(PF_INET, SOCK_STREAM, 0))<0) {
    perror("Errore nella socket");
    exit(-1);
}
// rende l'indirizzo riutilizzabile
if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(int))<0) {
    perror("Errore nella setsockopt");
    exit(-1);
}
// bind
indserv.sin_family = AF_INET;
indserv.sin_addr.s_addr = INADDR_ANY;
indserv.sin_port = htons(porta);
memset(&(indserv.sin_zero), '\0', 8);
if (bind(sd, (struct sockaddr *)&indserv, sizeof(indserv)) == -1) {
    perror("Errore nella bind");
    exit(-1);
}
// listen
if (listen(sd, BACKLOG)<0) {
    perror("Errore nella listen");
    exit(-1);
}
printf("servente_chat pronto; 'Ctrl+c' per chiuderlo\n\n");
// aggiunge sd al set principale
FD_SET(sd, &pri);
// conserva il descrittore di file piu' grande
sdgr = sd;
/*
* cuore del programma:
* prima della select copia la lista pri (che contiene tutti i socket
* aperti e quello tuttora in ascolto) in set_lettura in modo che pri
* non venga "sporcato" dalla select; set_lettura è usato solo nella select
* e nel successivo test per rilevare socket attivi; eventuali nuovi
* socket connessi vengono aggiunti a pri cosi' come i socket chiusi
* vengono rimossi da pri
*/
while(1) {

```

```

set_lettura = pri;                // imposta set per la select()
if (select(sdgr+1, &set_lettura, NULL, NULL, NULL)<0) {
    perror("Errore nella select");
    exit(-1);
}
// ispeziona le connessioni per cercare dati in arrivo
for(i=0;i<=sdgr;i++) {
    if (FD_ISSET(i, &set_lettura)) {    // trovato un socket con dati
        if (i==sd) {
            lind = sizeof(indcli);
            if ((csd=accept(sd, (struct sockaddr *)&indcli,&lind))<0) {
                perror("Errore nella accept");
            }
        }
        else {
            FD_SET(csd, &pri);    // aggiunge socket conn al set pri
            if (csd > sdgr) {
                sdgr = csd;
            }
            printf("servente_chat: connessione da %s sul "
                "socket %d\n",inet_ntoa(indcli.sin_addr),csd);
            vet[csd]=indcli.sin_addr;    // conserva IP cliente
        }
    }
    else {
        // gestisce dati in arrivo da un cliente
        if ((bytel=leggisock(i,buf,sizeof(buf)))<= 0) {

            if (bytel==0) {
                // connessione chiusa dal cliente
                printf("servente_chat: socket %d chiuso\n", i);
            }
            else {
                perror("Errore in ricezione");
            }
            close(i);
            FD_CLR(i,&pri);    // toglie il socket da pri
        }
        else {
            // invia i dati ai vari clienti
            // prima recupera IP del mittente e ne trova il nome
            // grazie a IP salvato in vet[i]; mette il nome
            // in buf2 in testa ai dati da inviare
            strcpy(buf2,"Da: ");
            cli=gethostbyaddr((char *)&vet[i],4,AF_INET);
            if (cli!=NULL) {
                strcat(buf2,cli->h_name);
            }
            else {
                strcat(buf2,inet_ntoa(vet[i]));
            }
            strcat(buf2," ----> ");
            strcat(buf2,buf);
            for(j=0;j<=sdgr;j++) {
                if (FD_ISSET(j, &pri)) {

```

```

// non invia i dati al mittente
if (j!=sd && j!=i) {
    if (scrivisock(j,buf2,bytel)<0) {
        perror("Errore in spedizione");
    }
}
}
// fine if (FD_ISSET(j.....))
// fine for j<=sdgr
// fine else: invia i dati
// fine else: gestisce dati in arrivo
// fine if (FD_ISSET(i.....))
// fine for i<=sdgr
// fine while(1)
return 0;
}

```

Nel prossimo listato abbiamo il cliente di *chat* che deve essere lanciato fornando il nome del servente e la porta a cui collegarsi.

L'indirizzo IP da utilizzare per la connessione viene determinato in base al nome del servente con la funzione *gethostbyname()*.

Come detto, anche qui viene utilizzata la funzione *select()* allo scopo di controllare simultaneamente la tastiera e il socket di rete, dopo la connessione.<sup>7</sup>

```

/* Programma:      cliente_chat
* Autore:         FF
* Data:           08/02/2006
* Descr.:         Esempio di cliente TCP che chiede una connessione
*                 a un servente per scambiare messaggi con altri clienti;
*                 determina l'IP del servente dal nome; usa select
*                 per gestire il mutiplexing fra tastiera e socket di rete
*/
#include "utilsock.h"
/* utilsock.h contiene tutte le inclusioni delle altre librerie necessarie
* e le funzioni di lettura e scrittura dal socket
*/
#include <netdb.h>
#include <sys/select.h>
#define MAX        256

int main(int argc, char *argv[])
{
    int sd,l;
    struct sockaddr_in serv_ind;
    struct hostent *ipserv;
    in_port_t porta;
    char msg[MAX];
    int msglen=MAX;
    fd_set setl,setw,sete;
    printf("\033[2J\033[H");
    // controlla argomenti
    if (argc !=3) {
        printf("\nErrore\n");
    }
}

```

```

        printf("Esempio di connessione: \n");
        printf("./client nome_servente porta (>1023)\n");
        printf("Premere Invio\n");
        getchar();
        exit(-1);
    }
    ipserv=gethostbyname(argv[1]);    // trova IP servente in base al nome
    // crea socket
    if ( (sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Errore in creazione socket");
        exit(-1);
    }
    // indirizzo IP
    memset((void *)&serv_ind, 0, sizeof(serv_ind)); // pulizia ind
    serv_ind.sin_family = AF_INET;                  // ind di tipo INET
    porta = atoi(argv[2]);
    serv_ind.sin_port = htons(porta);                // porta a cui collegarsi
    memcpy(&serv_ind.sin_addr.s_addr, ipserv->h_addr, ipserv->h_length);
    // copiato ip servente dalla struct ipserv a serv_ind.sin_addr
    // stabilisce la connessione
    if (connect(sd, (struct sockaddr *)&serv_ind, sizeof(serv_ind)) < 0) {
        perror("Errore nella connessione");
        exit(-1);
    }
    printf("Connessione stabilita con %s\n", inet_ntoa(serv_ind.sin_addr));
    printf("Per terminare digitare il messaggio: /ciao\n");
    while (1)
    {
        FD_ZERO (&setl);        // azzera i set; solo setl viene davvero usato
        FD_ZERO (&setw);
        FD_ZERO (&sete);
        FD_SET(fileno(stdin), &setl); // aggiunge descrittore di stdin in setl
        FD_SET(sd, &setl);
        if (select(FD_SETSIZE, &setl, &setw, &sete, NULL) < 0) {
            perror("Errore nella select");
            exit(1);
        }
        if (FD_ISSET(fileno(stdin), &setl)) {    // tastiera
            for(l=0; ((msg[l]=getchar())!='\n' && l<MAX-1); l++);
            msg[l]='\0';
            if (strcmp("/ciao", msg)==0) {
                printf("Fine\n");
                break;
            }
            scrivisock(sd, msg, MAX);
            printf("<---- OK\n");
        }
        if (FD_ISSET (sd, &setl)) {
            if ( (read(sd, msg, msglen)) <= 0 ) {
                perror("Errore nella read");
                exit(-1);
            }
            printf("%s\n", msg);
        }
    }
}

```



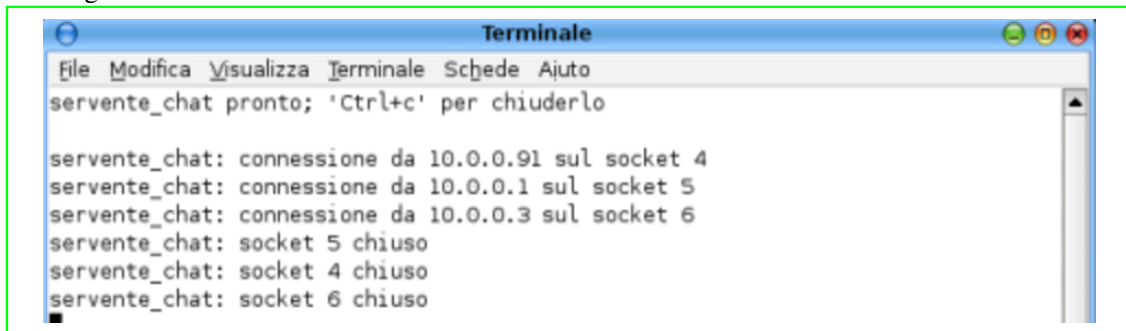
```

}
close(sd);
exit(0);
}

```

Nelle figure 6.12, 6.13, 6.14 e 6.15 viene mostrato un esempio di *chat* con il server e tre clienti coinvolti.

Figura 6.12



```

Terminale
File Modifica Visualizza Terminale Schede Ajuto
servente_chat pronto; 'Ctrl+c' per chiuderlo

servente_chat: connessione da 10.0.0.91 sul socket 4
servente_chat: connessione da 10.0.0.1 sul socket 5
servente_chat: connessione da 10.0.0.3 sul socket 6
servente_chat: socket 5 chiuso
servente_chat: socket 4 chiuso
servente_chat: socket 6 chiuso

```

Figura 6.13



```

Terminale
File Modifica Visualizza Terminale Schede Ajuto
fulvio@ferronif:~/lavoro/prove-mie/c/socket$ ./cliente_chat ferronif 10000
Connessione stabilita con 10.0.0.1
Per terminare digitare il messaggio: /ciao
Da: ferro.max.planck ----> messaggio da nodo 3
Da: fulviofe.max.planck ----> messaggio da nodo 91
msg da nodo 1
<---- OK
nodo 1 chiude
<---- OK
/ciao
Fine
fulvio@ferronif:~/lavoro/prove-mie/c/socket$

```

Figura 6.14



```

root@ferro: /
File Edit View Terminal Tabs Help
root@ferro:/ # ./cliente_chat ferronif.max.planck 10000
Connessione stabilita con 10.0.0.1
Per terminare digitare il messaggio: /ciao
messaggio da nodo 3
<---- OK
Da: fulviofe.max.planck ----> messaggio da nodo 91
Da: ferronif.max.planck ----> msg da nodo 1
Da: ferronif.max.planck ----> nodo 1 chiude
Da: fulviofe.max.planck ----> nodo 91 chiude
nodo 3 chiude
<---- OK
/ciao
Fine
root@ferro:/ #

```

Figura 6.15

```

root@fulviofe: /root
File Edit View Terminal Tabs Help
root@fulviofe:~ # ./cliente_chat ferronif.max.planck 10000
Connessione stabilita con 10.0.0.1
Per terminare digitare il messaggio: /ciao
Da: ferro.max.planck ----> messaggio da nodo 3
messaggio da nodo 91
<---- OK
Da: ferronif.max.planck ----> msg da nodo 1
Da: ferronif.max.planck ----> nodo 1 chiude
nodo 91 chiude
<---- OK
/ciao
Fine
root@fulviofe:~ #

```

## 6.4 Acquisizione dei dati delle interfacce di rete con *ioctl()*

In questo esempio abbiamo l'utilizzo della funzione *ioctl()* per acquisire informazioni sulle interfacce di rete presenti sul sistema.<sup>8</sup>

```

/* Programma:      sock_ioctl
 * Autore:         FF
 * Data:           10/02/2006
 * Descr.:         Esempio in cui si visualizzano le interfacce di rete
 *                 della macchina con relativi indirizzi IP, MAC, NETMASK
 *                 e BROADCAST usando la funzione ioctl()
 */
#include "utilsock.h"
/* utilsock.h contiene tutte le inclusioni delle altre librerie necessarie
 * e le funzioni di lettura e scrittura dal socket
 */
#include <sys/ioctl.h>
#include <net/if.h>
#define MAX        256

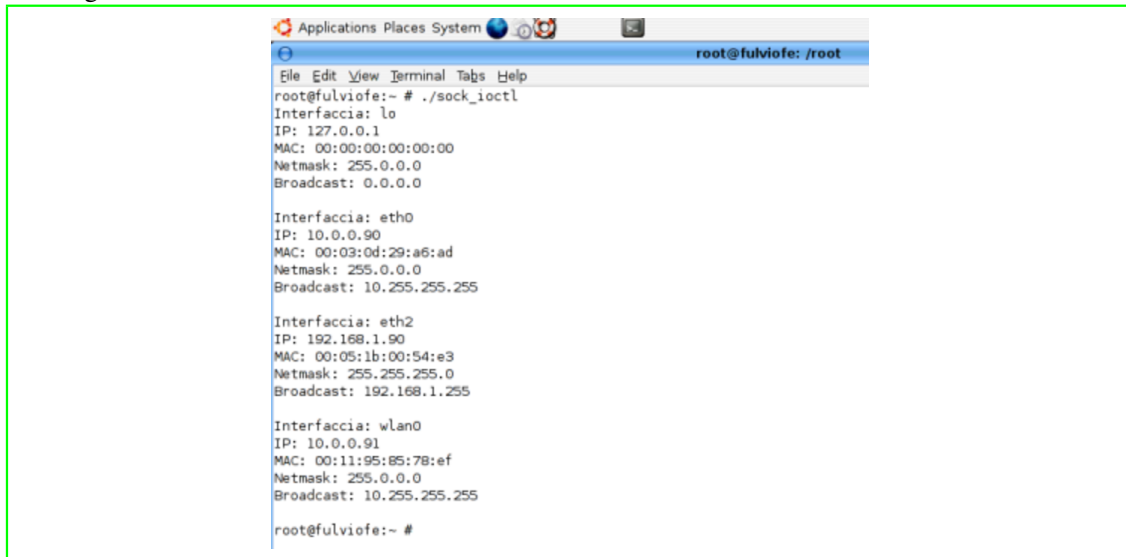
int main(int argc, char *argv[])
{
    int sd;                                // socket
    char buf[1024];                         // buffer dei dati
    struct ifreq *ifr = NULL;
    struct ifconf ifc;
    unsigned char *indmac;
    int i;
    // socket
    if ((sd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Errore nella socket");
        exit(-1);
    }
    // Recupero delle informazioni delle interfacce
    ifc.ifc_len = sizeof(buf);
    ifc.ifc_buf = buf;
    if (ioctl(sd, SIOCGIFCONF, &ifc) < 0) {
        perror("Errore nella ioctl SIOCGIFCONF");
        exit(-1);
    }
}

```

```
}
ifr=ifc.ifc_req;
for (i=0; i< ifc.ifc_len / sizeof(struct ifreq); i++, ifr++)
{
    printf("Interfaccia: %s\n", ifr->ifr_ifrn.ifrn_name);
    // IP
    if (ioctl(sd, SIOCGIFADDR, ifr)< 0) {
        perror("Errore nella ioctl SIOCGIFADDR");
        exit(-1);
    }
    printf("IP: %s\n",inet_ntoa
        (((struct sockaddr_in *)&(ifr->ifr_addr))->sin_addr));
    // MAC
    if (ioctl(sd, SIOCGIFHWADDR, ifr)< 0) {
        perror("Errore nella ioctl SIOCGIFHWADDR");
        exit(-1);
    }
    indmac=ifr->ifr_hwaddr.sa_data;
    // usato sa_data che è un campo della struttura sockaddr generica
    printf("MAC: %02x:%02x:%02x:%02x:%02x:%02x\n",
        indmac[0],indmac[1],indmac[2],indmac[3],indmac[4],indmac[5]);
    // Netmask
    if (ioctl(sd, SIOCGIFNETMASK, ifr)< 0) {
        perror("Errore nella ioctl SIOCGIFNETMASK");
        exit(-1);
    }
    printf("Netmask: %s\n",inet_ntoa
        (((struct sockaddr_in *)&(ifr->ifr_netmask))->sin_addr));
    // Broadcast
    if (ioctl(sd, SIOCGIFBRDADDR, ifr)< 0) {
        perror("Errore nella ioctl SIOCGIFBRDADDR");
        exit(-1);
    }
    printf("Broadcast: %s\n\n",inet_ntoa
        (((struct sockaddr_in *)&(ifr->ifr_broadaddr))->sin_addr));
}
close(sd);
exit(0);
}
```

Nella figura 6.17 vediamo il risultato di una esecuzione del programma su un sistema con diverse interfacce di rete.

Figura 6.17



```

root@fulviofe:~ # ./sock_ioctl
Interfaccia: lo
IP: 127.0.0.1
MAC: 00:00:00:00:00:00
Netmask: 255.0.0.0
Broadcast: 0.0.0.0

Interfaccia: eth0
IP: 10.0.0.90
MAC: 00:03:0d:29:a6:ad
Netmask: 255.0.0.0
Broadcast: 10.255.255.255

Interfaccia: eth2
IP: 192.168.1.90
MAC: 00:05:1b:00:54:e3
Netmask: 255.255.255.0
Broadcast: 192.168.1.255

Interfaccia: wlan0
IP: 10.0.0.91
MAC: 00:11:95:85:78:ef
Netmask: 255.0.0.0
Broadcast: 10.255.255.255
root@fulviofe:~ #

```

- <sup>1</sup> una copia di questo file, dovrebbe essere disponibile anche qui: *<allegati/progr-socket/utlsock.h>*.
- <sup>2</sup> una copia di questo file, dovrebbe essere disponibile anche qui: *<allegati/progr-socket/servente.c>*.
- <sup>3</sup> una copia di questo file, dovrebbe essere disponibile anche qui: *<allegati/progr-socket/cliente.c>*.
- <sup>4</sup> una copia di questo file, dovrebbe essere disponibile anche qui: *<allegati/progr-socket/servente\_i\_msg.c>*.
- <sup>5</sup> una copia di questo file, dovrebbe essere disponibile anche qui: *<allegati/progr-socket/cliente\_msg.c>*.
- <sup>6</sup> una copia di questo file, dovrebbe essere disponibile anche qui: *<allegati/progr-socket/servente\_chat.c>*.
- <sup>7</sup> una copia di questo file, dovrebbe essere disponibile anche qui: *<allegati/progr-socket/cliente\_chat.c>*.
- <sup>8</sup> una copia di questo file, dovrebbe essere disponibile anche qui: *<allegati/progr-socket/sock\_ioctl.c>*.

# Appendici