Socket

[Pagina intenzionalmente vuota]

Concetto di socket

- Disponibili su tutti i sistemi odierni
- Endpoint di canale (logico) di comunicazione tra processi, anche **remoti**
- vanno create e collegate dinamicamente
- dopo, le socket sono accessibili come file con modalità dipendenti dalle loro caratteristiche

Caratteristiche delle socket:

dominio insieme di socket che possono comunicare tra loro; p.es.:

1. `PF_UNIX` (locale)
2. `PF_INET` (internet)
3. `PF_IPX` (IPX Novell)

tipo semantica della comunicazione, p.es.:

- `SOCK_STREAM` endpoint di byte stream: sequenced, reliable, two-way, connection-oriented
- `SOCK_DGRAM` endpoint per messaggi "datagram" (connectionless, unreliable)

protocollo usato dai livelli di rete per supportare la comunicazione verso la socket;

p.es. `ip`, `tcp`, `udp`, `iso-tp4` (vedi file `/etc/protocols`)

ogni dominio ha un set ammesso di protocolli (in genere uno per tipo)

L'uso delle socket segue due modalità distinte secondo che la semantica della comunicazione supportata sia: *connectionless* o *connection-oriented*

Uso delle socket

[Pagina intenzionalmente vuota]

Comunicazione connection-oriented

Connessione tra processi detti Server e Client (si noti l'asimmetria):

```
Server                                   Client
s = socket(...);                         t = socket(...);
bind(s,&myAddr,addrLen);
listen(s,pendingQSize);
s1 = accept(s,...);<----------connect(t,&servrAddr,addrLen);
```

Server:

- crea socket con `s=socket(...)`
- assegna un indirizzo alla socket con `bind(s...)`
- dichiara quante connessioni accetterà sulla socket, con `listen(s...)`
- accetta connessioni con `s1=accept(s...)`
- riceve/invia dati con `read(s1,...)`/`write(s1,...)`

Client:

- crea socket con `t=socket(...)`
- lo collega all'indirizzo del socket del server con `connect(t...)`
- invia/riceve dati sulla socket con `write(t...)`/`read(t...)`

Entrambi chiudono con `close()` su `s` e `t` (eventualmente preceduto da `shutdown()`).

NB

- per stabilire la *connessione* (associazione) tra socket (`s1` e `t` sopra) è essenziale che le due parti usino lo stesso *indirizzo* (`myAddr=servrAddr` sopra)
- la connessione stabilita è bidirezionale

Apertura di socket

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

dove l'`int` restituito è la socket (-1 in caso di errore)

`domain` individua un insieme di socket collegabili (non si possono collegare socket in domini diversi);

i domini più usati sono `PF_UNIX` e `PF_INET` (costanti)

`type` definisce la semantica della comunicazione; per lo più vale

- `SOCK_STREAM`: *stream* (flusso) di dati "in sequenza", affidabile, a due vie e con messaggi "out-of-band"

  prima di avviare lo stream occorre stabilire una *connessione*.
- `SOCK_DGRAM`: per scambio di "datagram", connectionless e inaffidabile

`protocol`: se `0`, seleziona il protocollo di default (spesso l'unico) per il `domain`/`type`;

Le sole coppie tipo-protocollo ammissibili nel dominio `PF_INET`:

- `SOCK_STREAM` e `6` o `IPPROTO_TCP` (TCP)
- `SOCK_DGRAM` e `17` o `IPPROTO_UDP` (UDP)
- `SOCK_RAW` (dà accesso diretto al layer IP) e qualunque protocollo valido su IP secondo RFC1700 (ma occorre essere root)

(vedi `/etc/protocols` per la mappa numeri-nomi dei protocolli, `<netinet/in.h>` per le costanti, `man 7 ip` in generale)

Socket non bloccanti

Per default la socket `s` è *bloccante* rispetto a I/O.

Diviene non-bloccante con `fcntl(s,F_SETFL,O_NONBLOCK)` (`open()` non si può usare):

```
/* nblksock.c */
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{   int s, flags;
    s = socket(PF_UNIX,SOCK_DGRAM,0);
    flags = fcntl(s,F_GETFL);
    printf("Current\nflags=%0.4x\tO_NONBLOCK=%0.4x\t\
            flags|O_NONBLOCK=%0.4x\n\n",
           flags, O_NONBLOCK, flags|O_NONBLOCK);
    fcntl(s,F_SETFL,O_NONBLOCK);
    printf("I have set\nflags=%0.4x\n", fcntl(s,F_GETFL));
    close(s);
    exit(0);
}
```

Vedi anche esempio a p. 47.

Binding di socket a un indirizzo

Il (descrittore di) socket ha un significato solo locale.
Legarla a un indirizzo le dà un "nome" usabile all'esterno (la *pubblica*)

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

Restituisce 0 o -1 se si ha errore, p.es.:

EBADF sockfd is not a valid descriptor

ENOTSOCK argument is descriptor for file, not socket

EINVAL socket already bound to an address (per la precisione: il legame socket-indirizzo deve essere 1-1)

EACCES address is protected, and the user is not the super-user

Strutture degli indirizzi di socket

Dagli `/usr/include/sys/socket.h`:

```
struct sockaddr {
    sa_family_t  sa_family;     // address family: AF_xxx
    char         sa_data[14];   // protocol address (14 bytes)
};
```

Ma la `struct sockaddr` si specializza secondo il dominio della socket (v. `man 4 unix` e `man 4 ip`):

- in `PF_UNIX` diventa

```
#define UNIX_PATH_MAX    108

struct sockaddr_un {
    sa_family_t sun_family;          // sempre AF_UNIX
    char sun_path[UNIX_PATH_MAX];    // pathname
};
```

- in `PF_INET` diventa

```
struct sockaddr_in {
    sa_family_t    sin_family;   // address family: AF_INET
    u_int16_t      sin_port;     // port in network byte order
    struct in_addr sin_addr;     // internet address
    // byte di padding fino a sizeof(struct sockaddr)
};

struct in_addr {
    u_int32_t s_addr;     // IPv4 address in network byte order
};
```

Type casting per `struct sockaddr`

Se una system call per le socket ha un argomento indirizzo di socket:

- nel prototipo figura il tipo generico `struct sockaddr *`; p.es.:

```
int bind(int sockfd, struct sockaddr *my_addr,
         int addrlen);
```

- nella chiamata, l'argomento ha un tipo specifico `struct sockaddr_xxx *`, reso legale via type casting; p.es.:

```
struct sockaddr_in thisAddr;  // da inizializzare
...
bind(sd, (struct sockaddr * ) &thisAddr, alen);
```

Connessione di una socket a un indirizzo remoto

Un processo (tipicamente client) crea una connessione tra:

- una propria socket `sockfd`
- e una remota (tipicamente di un server) menzionando l'indirizzo `sockaddr` di questa in `connect()`:

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *serv_addr,
            int addrlen);
```

In generale, una socket `sockfd` di tipo `SOCK_STREAM` può essere oggetto di `connect` con successo **solo una volta**.

Restituisce 0 o -1 se si ha errore, p.es.:

`EBADF` Bad descriptor

`ENOTSOCK` The descriptor is not associated with a socket.

`EISCONN` The socket is already connected.

`ECONNREFUSED` Connection refused at server.

`ETIMEDOUT` Timeout while attempting connection.

`ENETUNREACH` Network is unreachable.

`EADDRINUSE` Address is already in use.

Attesa della connessione lato server: `listen()`

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

Indica volontà di attendere connessioni verso la socket `s`:

- si applica solo a socket di tipo `SOCK_STREAM` e `SOCK_SEQPACKET`
- non comporta ancora accettazione
- `backlog` è la max lunghezza ammessa per la coda delle richieste `connect` pervenute

Se la richiesta perviene a coda piena:

- il cliente può tornare da `connect()` con `ECONNREFUSED`
- oppure, se il protocollo sottostante supporta la ritrasmissione, viene ignorata (in vista dei retry lato client).

  Si veda anche pag 55.

Restituisce 0 o -1 se si ha errore:

EBADF The argument s is not a valid descriptor.

ENOTSOCK The argument s is not a socket.

EOPNOTSUPP The socket is not of a type that supports the listen operation.

Accettazione della connessione lato server: `accept()`

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr *addr, int *addrlen);
```

Usata con socket di tipo connection-oriented (in pratica `SOCK_STREAM`).

Semantica:

- estrae la prima richiesta `connect` dalla coda di quelle pendenti (definita con `listen()`)
- crea e restituisce una nuova socket, con le proprietà di `s`, collegata a quella remota

  N.B.:
  - la nuova socket **non** si può usare come 1o arg. di altre `accept()`
  - la prima socket `s` rimane aperta
- se non vi sono richieste pendenti, `accept()` blocca se `s` non è marcata non-blocking, altrimenti dà errore (`EAGAIN`)

Indirizzo (del client collegantesi):

- `addr` è un parametro risultato, di formato dipendente dal dominio
- `addrlen` è un parametro value-result (inizialmente deve valere il n. di byte allocati per `addr`)
- nel dominio `AF_INET`, `addr` contiene l'ip del client e un port (alto) scelto automaticamente dal socket layer

Prototipi di `read` e `write`

```
#include <unistd.h>
ssize_t read(int fd, void *buf,  size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

Osservazioni sui tipi:

- `const` indica che `buf` non è modificato da `write()`
- `size_t` (definito, p.es., in `/usr/include/sys/types.h`) è un intero *unsigned*, adatto a contare i byte "misura" di un oggetto;
- `ssize_t` (definito, p.es., in `/usr/include/sys/types.h`) è un intero *signed*, adatto come tipo del valore restituito da una system call. In particolare:
  - `read()` restituisce il n. di byte letti o -1 in caso di errore
  - `write()` restituisce il n. di byte scritti o -1 in caso di errore

Il I argomento `fd` è il descrittore di file (o pipe o socket) da cui `read(fd,buf,count)` legge o su cui `write(fd,buf,count)` scrive.

Il II argomento `buf` punta alla memoria in cui risiedono i byte che `read(fd,buf,count)` ha letto da `fd` o `write(fd,buf,count)` scriverà su `fd`.

Il III argomento `count` è il numero di byte che `read(fd,buf,count)` cerca di leggere da `fd` su `buf` o `write(fd,buf,count)` cercherà di copiare da `buf` su `fd`

- `count` non deve superare la dim. di `buf` (tipicamente sono $=$)
- il n. di byte veramente letti/scritti, restituito da `read()`/`write()`, può risultare inferiore al n. richiesto `count`; per le socket, ciò accade tipicamente se:
  - `read()` non trova `count` byte disponibili (perché non sono stati inviati o, comunque, non sono ancora arrivati
  - `write()` viene interrotta da un segnale prima di avere scritto `count` byte)

Lettura e scrittura. Chiusura

Tipica sequenza di operazioni, in modalità connection-oriented:

Server

```
s = socket(PF_INET,SOCK_STREAM,0);
bind(s,&myAddr,sizeof(myAddr));
listen(s,maxQSize);
s1 = accept(s,&clntaddr,&addrL);
...
read(s1,inMsg,sizeof(inMsg));
```

Client

```
t = socket(PF_INET,SOCK_STREAM,0);
connect(t,&servrAddr,sizeof(servrAddr));
...
write(t,outMsg,sizeof(outMsg))
```

La chiusura si può fare con `close()`, come per i file.

Se la socket era bound, non sarà disponibile per un po', a meno che avesse il flag `SO_REUSEADDR` a 1.
Il ritardo è inteso per i pacchetti ancora in giro al `close`.

Chiusura selettiva con:

```
#include <sys/socket.h>
int shutdown(int s, int how);
```

Chiude tutte o parte delle operazioni associate con una connessione full-duplex terminante sulla socket `s`.

In particolare:

`how=0` disabilita ulteriori ricezioni

`how=1` disabilita ulteriori invii

`how=2` disabilita ulteriori invii e ricezioni

Semantica di lettura e scrittura

Semantica di `read` e `write` simile a quella di pipe e file:

- `read(s,buf,cnt)` legge fino a `cnt` byte
- il valore restituito è il n. di byte letti o `-1` in caso di errore
- leggere da socket `s` non bloccante, in mancanza di dati, causa `errno=EAGAIN`
- leggere da `s` bloccante (default):
  - se la connessione è aperta, in mancanza di dati, blocca (ovvio),
  - ma, se la connessione è stata chiusa (p. 14) dall'altro lato:
    * di norma `read()` legge 0 bytes
    * ma in certi casi può causare `EPIPE` (almeno sotto Linux)

      `read` non prevede `EPIPE` (`man read`), ma le system call per `AF_INET` sì (`man ip`)
- scrivere su `s` è bloccante se il buffer locale di supporto alla socket è pieno
- scrivere su `s` non (più) connessa all'altra estremità causa invio di `SIGPIPE` (e terminazione se `SIGPIPE` non è gestito o ignorato)
- `read` non è atomica/sincrona rispetto a `write` all'altro capo;

  cioè se si immagina che `write` e `read` costruiscono rispettivamente dei flussi di byte in entrata e in uscita:
  - l'unica garanzia è che il flusso in uscita è un prefisso di quello in ingresso
  - ma tra le singole porzioni inserite dai `write` e quelle estratte dai `read` non c'è una relazione prevedibile
  - la relazione che si stabilisce di volta in volta dipende dai tempi di trasmissione, dalle dimensioni dei buffer, dai protocolli di rete, etc.

Read singoli e multipli

La semantica di `read`/`write` (p. 15) rende un singolo `read` di dubbia utilità.

P. es. se dopo un singolo

```
retcode = read(s, buffer, NBYTES);
```

`retcode==R`, ciò non assicura comunque:

- né che all'altro capo siano stati scritti `R` byte e non di più
- né che verranno letti `NBYTES` byte, cioè che sia `NBYTES==R`

Serve invece qualcosa come (vedi anche esempio a p. 51):

```
while ((retcode = read(s, buffer, MAXBUF)) > 0) {
    // usa i byte in buffer
}
```

che cicla finché `read` trova byte in `s`, quindi si hanno i casi

```
// o retcode== 0 (è stata chiusa la connessione all'altro capo)
// o retcode==-1 && errno==EPIPE (chiusa connessione all'altro capo)
// o retcode==-1 && errno!=EPIPE (altro errore su read)
```

Oppure, per leggere un messaggio di lunghezza prefissata `MLEN`:

```
for (n = 0; n < MLEN; n += retcode) {
    if ( (retcode = read(s, buffer+n, MAXBUF-n)) <= 0 )
        break;   // connessione chiusa o errore
}
```

che cicla finché sono disponibili `MLEN` bytes o più in `buffer`; quindi di nuovo:

```
// o retcode== 0 (è stata chiusa la connessione all'altro capo)
// o retcode==-1 && errno==EPIPE (chiusa connessione all'altro capo)
// o retcode==-1 && errno!=EPIPE (altro errore su read)
```

Vedi anche esempio a p. 39.

Considerazioni analoghe per `recv` (p. 19).

Comunicazione connectionless

Le system call per scambiare dati tra socket non connesse sono:

- `sendto(loc_s,..., &rem_addr,...)` o `send(loc_s,...)`
- `recvfrom(loc_s,..., &rem_addr,...)` o `recv(loc_s,...)`

Le alternative

- `send(loc_s,...)`
- `recv(loc_s,...)`

hanno questa motivazione:

- `sendto` e `recvfrom` specificano la socket locale e l'indirizzo remoto con cui vengono scambiati i dati
- `send` e `recv` presuppongono che la socket locale sia connessa

  NB: la comunicazione non è in questo caso "connectionless"

Comunicazione connectionless: `send`

Per inviare datagram dall'indirizzo `fromAddr` a `toAddr`:

```
from_s = socket(dom,SOCK_DGRAM,0);//crea socket (endpoint locale)
bind(from_s, &fromAddr);          //associa socket a indirizzo locale
sendto(from_s,...,&toAddr...);
```

Funzioni send:

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int sockfd, const void *buf, size_t len, int
ssize_t sendto(int sockfd, const void *buf, size_t len,
               int flags,
               const struct sockaddr *dest_addr, socklen_t
```

dove:

- `send` valida solo per socket `s` collegata a una remota
- `flags` è l'or di alcuni tra:

```
#define MSG_OOB        0x1     //send out-of-band data (over connection)
#define MSG_DONTROUTE  0x4     //bypass routing (for testing)
#define MSG_DONTWAIT   0x40    //non-blocking send
#define MSG_NOSIGNAL   0x2000  //don't raise SIGPIPE if other endpoint breaks
```

Semantica:

- non sono permessi messaggi troppo lunghi per l'invio *atomico* via protocollo sottostante (errore `EMSGSIZE`)
- con un messaggio che non entra nel *send buffer* del socket, `send` blocca o, se la socket è non-blocking, causa `EAGAIN`
- `send` restituisce il n. di byte trasmessi o `-1` (errore)
- gli errori hanno significato locale (per lo più standard per socket layer) e non implicano mancata ricezione del messaggio

Comunicazione connectionless: `receive`

Per ricevere datagram a `toAddr` e sapere il mittente `whichAddr`:

```
to_s = socket(dom,SOCK_DGRAM,0);   //crea socket (endpoint locale)
bind(to_s, &toAddr);               //associa socket a indirizzo locale
recvfrom(to_s,...,&whichAddr...);
```

Synopsis:

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

dove:

- `recv` = `recvfrom` con `NULL` `from`; si usa per `s` connessa
- altrimenti, a `*from` viene assegnato l'indirizzo del mittente dei dati
- `fromlen` è un argomento *value-result*, che punta:
  - inizialmente alla dimensione del buffer per `from`
  - al ritorno alla lunghezza dell'indirizzo in `from`
- `flags` è l'or di `MSG_OOB`, `MSG_PEEK`, `MSG_WAITALL`, etc.

Semantica:

- in assenza di errori, la receive restituisce il n. di byte letti;
- se il buffer `buf` è troppo piccolo, si possono perdere dati
- di solito, la receive torna con i dati disponibili

  ma con il flag `MSG_WAITALL` si blocca in attesa di tutti i dati richiesti (3o arg)
- con il flag `MSG_PEEK` la receive restituisce i dati, ma non li elimina dalla coda di ricezione
- receive è bloccante sse lo è la socket (p. 6) (cf. `EAGAIN`, p. 20)

Receive: errori

These are some standard errors generated by the socket layer.

Additional errors may be generated and returned from the underlying protocol modules; see their manual pages.

EBADF The argument `s` is an invalid descriptor.

ENOTSOCK The argument s does not refer to a socket.

EFAULT The receive buffer pointer(s) point outside the process's address space.

EINVAL Invalid argument passed.

ENOTCONN The socket is associated with a connection-oriented protocol and has not been connected (see `connect`(2) and `accept`(2)).

Ma sotto Linux sembra che receive causi `ENOTCONN` anche con protocollo `udp` e socket `SOCK_DGRAM`, se dal lato di chi invia non si è fatto `bind()`.

EAGAIN Two cases:

- The socket is marked non-blocking and the receive operation does not find the data requested, or
- a receive timeout had been set and the timeout expired before data was received.

EINTR The receive was interrupted by delivery of a signal before any data were available.

Datagram e `connect`

```
connect(int sockfd, struct sockaddr *serv_addr, ...);
```

- più propriamente da utilizzare per connettere socket di tipo stream
- ma si può usare anche per socket `sockfd` datagram
- comunque, per socket stream o datagram, dà accesso alle varianti `send` di `sendfrom` e `recv` di `recvfrom` (p. 17)

Da completare.

Select

Vedi esempio `examples/socket/getmsgTout.c`, parte di `examples/socket/getmsg.c` (a pag. 47)

```
/* getmsgTout.c */
// to be included in getmsg.c

extern char buf[MAXBUF];

void getmsg_tout(int sock) // attende messaggio entro timeout
{
    struct timeval tout;    //definisci e inizializza valori di timeout
    tout.tv_sec = TOUTS;
    tout.tv_usec = TOUTUS;

    // select permette di sapere se una socket (in gen. file descriptor)
    // sara' disponibile in lettura entro un determinato timeout

    fd_set FD;           // set (contenitore) di descrittori di socket
    FD_ZERO(&FD);        // FD diviene insieme di descrittori vuoto;
    FD_SET(sock, &FD); // aggiunge al set FD la socket sock

    int ret = select(sock+1, &FD, NULL, NULL, &tout);
    // non attende eventi write/exception (NULL/NULL), in FD c'e' solo sock
    printf("select() ha visto eventi ");
    printf("su n.%d file descriptor\n", ret);

    switch (ret) {
    case -1:
        perror("Error occurred");
        break;
    case 0:  // select() restituisce n. di fd su cui c'e' stato un evento
        printf("Timeout occurred.\n");
        break;
    default: // select() lascia nel set FD solo i descrittori su cui
             // c'e' stato un evento, qui in FD puo' esserci solo sock
        if (FD_ISSET(sock,&FD)) // sempre vero: FD contiene solo sock
            getmsg_blk(sock);    // qui get_msg_blk() non blocca
    }                            // perche' si e' passato select()
}
```

Indirizzi IP

Strutture degli indirizzi di socket IP

Indirizzi di socket nel dominio `PF_INET`:

```
struct sockaddr_in {
    sa_family_t    sin_family;  // address family: AF_INET
    u_int16_t      sin_port;    // port in network byte order
    struct in_addr sin_addr;    // internet address<
    // byte di padding fino a sizeof(struct sockaddr)
};

struct in_addr {
    u_int32_t s_addr;     // IPv4 address in network byte order
};
```

Notare l'uso degli indirizzi di socket in

```
bind(int sockfd, struct sockaddr *my_addr, int addrlen)
```

Il membro `.sin_addr` di `my_addr` può assumere:

- `INADDR_ANY` (`0x00000000`): ascolta su tutti gli indirizzi IP locali (dell'host)
- indirizzi multicast (di gruppo) e broadcast (vedi `man 7 ip`)

IP address: formati puntato, di rete, host

Un indirizzo IP, nel formato in cui viaggia in *rete*, è un dato a 32 bit
p.es. (trascritto come 4 byte, in formato hex): `0x 97 4a fd 25`,
ovvero: `10010111 01001010 11111101 00100101`

NB: i bit sono elencati nell'ordine in cui passerebbero su una linea seriale.

Di solito l'indirizzo IP si trascrive in formato *puntato* (*dot notation*):

- individuando i 4 byte del formato di rete, dal primo in poi
- trascrivendoli in formato decimale senza segno
- separandoli con punti
- p.es. `0x 97 4a fd 25 -> 151.74.253.37`

In C non è standard che `int` o `long int` sia a 32 bit.
Quindi un indirizzo IP `addr` si definisce:

```
    #include <sys/types.h>     // tipi ANSI-C, tra cui u_int32_t
    u_int32_t addr;            // unsigned 32 bit integer
// oppure
    #include <sys/stdint.h>    // tipi standard, tra cui uint32_t
    uint32_t addr1;            // unsigned 32 bit integer
```

NB: i 4 byte dell'indirizzo IP sono trattati:

- dal sw di rete alla *big endian*:
- dai programmi C secondo convenzioni locali (*little-endian* per x86)

Esempio (host little-endian):

1. client: `connect` verso `0x 97 4a fd 25 = 151.74.253.37`
2. server con `uint32_t addr`:

```
accept(s,&from,alen);   lascia in from.in_addr.s_addr i byte
                        0x 97 4a fd 25 in indirizzi crescenti
addr = from.in_addr.s_addr;    copia indirizzo IP in addr
```

3. ma, su un host che segue la convenzione little endian, si ha:

```
printf("%u", addr);      // scrive 0x25fd4a97
lo = addr % 0x100;       // rende lo == 0x97 == 151
```

Viceversa, sempre su host little endian, per porre in `addr` l'IP address
`0x974afd25=151.74.253.37`:

```
addr = 0x25fd4a97;     // memorizza i byte 0x97 4a fd 25
```

Ma tale codice non è una buona idea . . . : non è leggibile, né portabile

Conversione tra formati di IP address

Conversioni tra formati di rete e locali (host):

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

Conversioni tra formati dotted e a 32 bit:

```
#include <arpa/inet.h>    //include a sua volta netinet/in.h

int inet_aton(const char *cp,
              struct in_addr *inp);   ascii to net (in *inp)
char *inet_ntoa(struct in_addr in);   net to ascii (pointer returned)
```

dove (da `netinet/in.h`):

```
struct in_addr {
   uint32_t s_addr;    //address in network byte order
}
```

Conversione tra formati: esempio

```
/* addrcnv.c */

#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <string.h>

int main(int argc, char * argv[])
{
    struct in_addr sa;
    char *p, dotted[16];
    int res;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s X.Y.Z.W\n", argv[0]);
        fprintf(stderr, "\tda X.Y.Z.W (dotted quad) a ");
        fprintf(stderr, "binary IP addr e indietro\n");
        exit(1);
    }
    inet_aton(argv[1], &sa);    // converte la stringa argomento in sa
    p = inet_ntoa(sa);          // converte sa nella stringa p
    strcpy(dotted, p);          // p punta a memoria statica
    res = strcmp(dotted, argv[1]);      // le stringhe devono essere uguali
    printf("strcmp(%s, %s) yields %d\n\n",
           dotted, argv[1], res);

    // Test: *p viene sovrascritto da un'altra chiamata inet_ntoa
    printf("string p is %s\n", p);
    inet_aton("255.255.255.255", &sa);
    inet_ntoa(sa);
    printf("string p is %s (overwritten by inet_ntoa())\n", p);
    exit(0);
}
```

Ricerca di indirizzi nel dominio Inet

```
#include <netdb.h>
extern int h_errno;
struct hostent *gethostbyname(const char *name);

#include <sys/socket.h>  /* for AF_INET */
struct hostent *gethostbyaddr(
                  const char *addr, int len, int type);
```

Resolver: sw che implementa le query `gethostbyname/addr`, indirizzandole

- alle sorgenti specificate, nell'ordine, in `/etc/nsswitch.conf` (`/etc/host.conf` con la vecchia libreria C)
- o, per default, nell'ordine a:
  1. DNS, Domain Name Service (distribuito)
  2. file `/etc/hosts` (tabella IP address - name - aliases)

Se una query è diretta a DNS, ma DNS non è attivo sull'host, essa va, nell'ordine, ai `nameserver` indicati in `/etc/resolv.conf`

Nomenclatura Unix relativa a DNS:

BIND (Berkeley Internet Name Domain): implementazione di DNS in Unix BSD
named (**name d**aemon): processo server DNS.

La query restituisce un puntatore:

- a una `struct hostent` (descrive nomi e indirizzi trovati),
- o, in caso di errore, `NULL`

Codici di errore in `h_errno`:

HOST_NOT_FOUND Host sconosciuto.

NO_ADDRESS or **NO_DATA** Nome noto a DNS, ma non corrisponde a IP address.

NO_RECOVERY errore *non-recoverable* del name server.

TRY_AGAIN Errore temporaneo su name server *authoritative* (riprovare).

Resolver: formato della host entry

```
#include <netdb.h>
extern int h_errno;
struct hostent *gethostbyname(const char *name);

#include <sys/socket.h>  /* for AF_INET */
struct hostent *gethostbyaddr(
                  const char *addr, int len, int type);
```

La struct `hostent` è definita in `<netdb.h>`:

```
struct hostent {
   char *h_name;         // official name of host
   char **h_aliases;     // alias list
   int  h_addrtype;      // always AF_INET
   int  h_length;        // length of address
   char **h_addr_list;   // list of addresses
}
#define h_addr h_addr_list[0]   // backward compatibility
```

NB (si veda anche l'esempio a p. 42):

- `h_length` è 4 (byte) per gli standard correnti
- l'array `h_addr_list[]` termina con una entry `NULL`

  ogni precedente `h_addr_list[0]`, `h_addr_list[1]`, ... punta a un indirizzo di `h_length` byte;
- il tipo di `h_addr_list[0]`, `h_addr_list[1]`, ... è `char *`;

  in realtà quindi essi puntano al 1° byte dell'IP address (a 4 byte):

  p.es. `*h_addr_list[0]` è il 1° byte del 1° indirizzo della host entry;
- con il cast `(uint32_t *)` ognuno di questi puntatori a `char` diviene un puntatore a IP address;

  p.es. `((uint32_t *) h_addr_list[0])` punta al 1° IP address, cioè `*((uint32_t *) h_addr_list[0])` è il 1° IP address.

Resolver: visualizzare la host entry

```
struct hostent {
   char *h_name;         // official name of host
   char **h_aliases;     // alias list
   int  h_addrtype;      // always AF_INET
   int  h_length;        // length of address
   char **h_addr_list;   // list of addresses
}
```

Per visualizzare questa `struct`:

```
/* printaddr.c */

/* Displays an IP struct hostent (see man gethostbyname) */

#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>

void printaddr(struct hostent * hp)
{
    char * *p, *ap;
    uint32_t addr;
    int al, n;

    printf(    "  Official name:   %s\n", hp->h_name);
    for (p = hp->h_aliases; *p != NULL; p++)
        printf("  Alias:           %s\n", *p);

    al = hp->h_length;

    for (p = hp->h_addr_list; (ap = *p) != NULL; p++) {
        printf("  Displaying IP addresses\n");
        printf("    byte by byte:  ");
        for (n = al; n > 0; n--)
            printf("%u.", (unsigned char) *ap++);

        addr = * ( (uint32_t *) * p);
        printf( "\n    32bit:         (net) %x  (host) %x\n",
               addr, ntohl(addr) );

        printf(  "    32bit div 256: ");
        for (n = al; n > 0; n--) {
            printf("%u%c", addr % 256, n == 1 ? '\n' : '.');
            addr /= 256;
        }
    }
}
```

Resolver: gethostbyname

```
struct hostent *gethostbyname(const char *name);
```

se la stringa `name` è:

- un nome, p.es. `www.unict.it`:
  - avviene la query
  - il puntatore restituito punta alla host entry risposta
- un indirizzo dotted, p.es. `151.74.253.37`
  - non si ha query
  - la routine costruisce una host entry con (unico) nome `name` e (unico) indirizzo di rete a 32 bit equivalente a `name`
  - a questa host entry punta il puntatore restituito

Esempio (vedi anche p. 42):

```
/* byname.c */
/* test gethostbyname(), gethostbyaddr() */

#include <stdio.h>
#include <netdb.h>
#include <stdlib.h>

void printaddr(struct hostent *);

int main(int argc, char * argv[])
{
    struct hostent *hp;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s name\n", argv[0]);
        exit(1);
    }
    // argv[1] is IP name: DNS lookup
    printf("\n>>>>DNS lookup for name %s\n\n", argv[1]);
    if ((hp = gethostbyname(argv[1])) == NULL)
        {herror("Resolver query"); exit(2);}
    printaddr(hp);
    exit(0);
}
```

Resolver: gethostbyaddr

```
struct hostent *gethostbyaddr(
                    const char *addr, int len, int type);
```

Effettua una *reverse* query:

- `addr` (un `char *`) punta a un indirizzo IP a 32 bit in formato di rete
- `len` (di fatto `4`) è la dimensione dell'indirizzo (serve perché `addr` è un `char *`
- `type` è sempre `AF_INET` (definito in `<sys/socket.h>`)
- il DNS fornisce l'unica host entry in cui figura l'indirizzo
- a questa entry punta il puntatore restituto

Esempio (vedi anche p. 42):

```
/* byaddr.c */
/* test gethostbyname(), gethostbyaddr() */

#include <stdio.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <stdlib.h>

void printaddr(struct hostent *);    // displays hostent

int main(int argc, char * argv[])
{
    struct hostent *hp;
    struct in_addr sa;  // ha il solo member sa.s_addr (u_int32_t)

    if (argc != 2) {
        fprintf(stderr, "Usage: %s X.Y.Z.W\n", argv[0]);
        exit(1);
    }
    // argv[1] is IP addr: reverse DNS lookup
    printf("\n>>>>DNS reverse lookup for IP addr %s\n\n",
           argv[1]);
    inet_aton(argv[1], &sa);
    hp = gethostbyaddr((char *) &(sa.s_addr),
                       sizeof(sa) , AF_INET  );
    if (hp == NULL) {herror("Resolver query");  exit(2);}
    printaddr(hp);
    exit(0);
}
```

Errori di DNS query

```
#include <netdb.h>
void herror(const char *s);
const char * hstrerror(int err);
```

The (obsolete) `herror()` function prints the error message associated with the current value of `h_errno` on stderr.

The (obsolete) `hstrerror()` function takes an error number (typically `h_errno`) and returns the corresponding message string.

Connessione di rete al DNS server

```
#include <netdb.h>
void sethostent(int stayopen);
void endhostent(void);
```

The `sethostent()` function specifies, if `stayopen` is true (`==1`), that:

- a connected TCP socket should be used for the name server queries
- the connection should remain open during successive queries.

Otherwise, name server queries will use UDP datagrams.

The `endhostent()` function ends the use of a TCP connection for name server queries.

Ricerca/scelta di port/servizi nel dominio Inet

- `/etc/services`, interrogato via `getservbyname` e `getservbyport`
- port libero, statico (fissato a compile time)
- ciclo di `bind` su valori crescenti del port: causa `EADDRINUSE` finché non si trova un port libero.

Esempi

Finger client

```
/* fingerc.c */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MAXHN 128
#define MAXBUF 8*1024    // provare a variare


int main(int argc, char * argv[])
{
    char buffer[MAXBUF];      // message buffer
    struct sockaddr_in addr; // server socket's address
    u_int16_t fport;
    int s, retcode;
    int mkaddr(struct sockaddr_in *, char *, u_int16_t);

    if(argc != 3 && argc != 4) {
        printf("usage: %s server_host message [port]\n", argv[0]);
        exit(1);
    }

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        {perror("opening socket"); exit(-1);}

    // collegamento al server host
    fport = argc==4 ? htons((uint16_t) atoi(argv[3]))
                    : htons(IPPORT_FINGER); // in netinet/in.h
    mkaddr(&addr, argv[1], fport);  // costruisce l'indirizzo del server
    if (connect(s, (struct sockaddr *) &addr, sizeof(addr))
        == -1 ) {perror("connecting"); exit(-2);}

    sprintf(buffer,"%s\n", argv[2]);  // buffer conterra' di certo una stringa
    if (write(s, buffer, strlen(buffer)) == -1)    // invia messaggio al serve
        {perror("writing to socket"); exit(-3);}

    // Sbagliato sostituire il prossimo while con if (anche per grandi MAXBUF)
    // read non e' atomica rispetto a write all'altro endpoint!

    while ((retcode = read(s, buffer,   // lascia un byte alla fine di
            MAXBUF-1)) != 0) {          // buffer, per eventuale ASCII 0
        if (retcode == -1)
            {perror("reading from socket"); exit(-4); }
        buffer[retcode] = '\0';  // ASCII 0 oltre l'ultimo byte ricevuto
```

```
        printf("\n----Read %d bytes----\n", retcode);
        printf("%s", buffer);
    }
    printf("\n>>>Exiting: server closed connection\n");
    close(s);  exit(0);
}
```

Semantica di `read`: esempi

Come segnalato nel commento al cliente di finger (p. 37), una sola `read` non è quasi mai appropriata.

Problema: `read` non è atomica rispetto a `write`.

Implicazioni della semantica di `read` (v. p 15):

- restituisce `0` se la connessione è stata chiusa all'altro capo
- bloccante finché sulla connessione non giungono nuovi dati
- si sblocca se arrivano dati
- se si immagina che `write` e `read` costruiscono rispettivamente dei flussi di byte in entrata e in uscita:
  - l'unica garanzia è che il flusso in uscita è un prefisso di quello in ingresso
  - ma le singole porzioni di flusso estratte dai `read` non hanno una relazione prevedibile con le porzioni inserite dai `write`

Alcuni esempi:

- `fingerc mit.edu brown`

  Sono attesi molti dati: anche se il buffer di `read` è grande abbastanza per tutti, la prima `read` può tornare prima che siano arrivati.

  Se in `fingerc.c` restituisce il ciclo di `read` con uno solo, si perderanno dati.

- `fingerc localhost xxxx 2000`, con il server fingers (p. 40)

  `fingers` invia messaggi di varia dimensione e temporizzazioni;

  si osservino le frontiere tra i messaggi ricevuti da `fingerc` con ogni `read`.

Uno pseudo finger server

```
/* fingers.c */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>        // per inet_aton(), ...

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LOCIP "127.0.0.1"
#define MAXQ 512
#define MYFINGERPORT 2000
#define MAXBUF 1024
#define NITER 10
#define MANYBYTES 16*1024
#define DELAY 7


int main()
{
    int s, s1;
    struct sockaddr_in locAddr, farAddr;
    socklen_t farAddrL, ipAddrL;
    int iter, retcode;
    char buf[MAXBUF], msg[MAXBUF], outbuf[MANYBYTES], *p, *q;

    if ( (s = socket(PF_INET, SOCK_STREAM, 0)) == -1 )
        {perror("creating socket"); exit(-10);}

    // Costruzione indirizzo socket nella struct locAddr
    locAddr.sin_family = AF_INET;
    locAddr.sin_port = htons((unsigned) MYFINGERPORT);
    // NB: si presuppone che LOCIP sia un IP address, se fosse
    //   un nome, servirebbe mkaddr()/gethostbyname(), cf. fingerc.c
    inet_aton(LOCIP, &locAddr.sin_addr);
    printf("My address/port: %s", inet_ntoa(locAddr.sin_addr));
    printf(":%d\n", ntohs(locAddr.sin_port));

    ipAddrL = farAddrL = sizeof(struct sockaddr_in);
    if ( bind(s, (struct sockaddr *) &locAddr, ipAddrL) == -1 )
        {perror("trying to bind"); exit(-1);}
    listen(s, MAXQ);

/* fingers.c ... */
```

```
/* fingers.c (cont.) */

    while ((s1 =
            accept(s, (struct sockaddr *) &farAddr, &farAddrL)
           ) != -1) {
        printf("Client from %s/%d connected\n",
               inet_ntoa(farAddr.sin_addr),
               ntohs(farAddr.sin_port));
        // we wrongly read only one message, but this may not suffice
        if ( (retcode = read(s1,buf,MAXBUF)) > 0) {
            buf[retcode-1] = '\0';      // overwrite NL
            printf("Client asks: <%s>\n", buf);
            buf[retcode-1] = '\n';
        }

        // reply to client, send various data
        sprintf(msg, "Hi from server: sending back %d bytes\n",
                retcode*NITER);
        write(s1, msg, strlen(msg));    // msg e' di certo una stringa
        sleep(DELAY);
        for (iter = 0; iter < NITER; iter++)
            write(s1,buf,retcode);
        sleep(DELAY);
        sprintf(msg, "Hi from server: sending back %d bytes\n",
                MANYBYTES);
        write(s1, msg, strlen(msg));    // msg e' di certo una stringa
        for (p = outbuf, q = p+MANYBYTES; p < q; p++)
            *p = '*';
        sleep(DELAY);
        write(s1, outbuf, MANYBYTES);
        close(s1);
        printf("Connection with client closed\n\n");
    }

    // Server non termina; servirebbe gestore SIGINT
    exit(0);
}
```

Semplice interfaccia al resolver

V. p. 30 per `printaddr()`.

```
/* gethost.c */
/* test gethostbyname(), gethostbyaddr() */

#include <stdio.h>
#include <ctype.h>
#include <netdb.h>
#include <stdlib.h>

void printaddr(struct hostent *);

int main(int argc, char * argv[])
{
    struct hostent * hp, *hq;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s [address|name]\n", argv[0]);
        exit(-1);
    }
    if (isalpha(argv[1][0])) // argv[1] is name to lookup
        printf("\n>>>>Querying DNS for name %s\n\n", argv[1]);
    else { // argv[1] is IP quad dotted address
        printf("\n>>>>Making hostent for IP addr %s ",argv[1]);
        printf("(no query for it)\n\n");
    }
    if ((hp = gethostbyname(argv[1])) == NULL) {
        fprintf(stderr, "Errore %d\n", h_errno);
        herror("Resolver");      // herror() obsolete
        exit(h_errno);
    }
    printaddr(hp);  putchar('\n');

    if (!isdigit(argv[1][0]))    // arg is not IP addr
        exit(0);                 // probably was name

    // arg is IP addr: reverse DNS lookup
    printf("\n>>>>DNS reverse lookup for hostent\n\n");
    hq = gethostbyaddr(hp->h_addr_list[0],
                       hp->h_length, AF_INET);
    if (hq == NULL) {
        fprintf(stderr, "Errore %d / ", h_errno);
        herror("Resolver");
        exit(h_errno);
    }
    printaddr(hq);  putchar('\n');
    exit(0);
}
```

Client-server in AF_UNIX: server

Ciclo server:

1. accetta una connessione su socket STREAM di nome noto
2. riceve un messaggio (anche su più righe, se tra apici)
3. riproduce il messaggio su stdout

```
/* unsrvr.c */
/* semplice server, Unix domain; notare signal handling */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <signal.h>

#define MAXBUF 16   // small, so that multiple reads will happen
#define NOMESOCK "servsock"

void terminazione(int), cleanup();
int terminated = 0;
int server_socket, connect_socket;   // global so both main() and
                                     // cleanup() may refer to them
char buffer[MAXBUF];

int main(int argc, char * argv[])
{
    socklen_t client_addr_len;       // necessario tipo previsto da accept
    int retcode;
    struct sockaddr_un client_addr, server_addr;

    // installa handler per terminazione server da tastiera
    signal(SIGINT,terminazione);

    // apertura del socket del server
    if ( (server_socket = socket(AF_UNIX,SOCK_STREAM,0)) == -1)
        {perror("opening server socket"); exit(-1);}

    // preparazione dell'indirizzo per il bind sulla socket
    if (unlink(NOMESOCK) == -1) {   // unlink() serve prima di bind()
        perror("unix socket");      // sulla socket NOMESOCK
        fprintf(stderr, "no old socket \"%s\" to delete\n",
                NOMESOCK);
    } else
        fprintf(stderr, "old socket %s deleted\n", NOMESOCK);
    server_addr.sun_family = AF_UNIX;
    strcpy(server_addr.sun_path, NOMESOCK);

/* unsrvr.c ... */
```

/* **unsrvr.c** (cont.) */

```
    // pubblicazione socket / listen
    retcode =
    bind(server_socket,
         (struct sockaddr *) &server_addr, //NB: type cast
         sizeof(server_addr) );
    if(retcode == -1)
        {perror("error binding socket"); exit(-1);}
    retcode = listen(server_socket, 1);
    if(retcode == -1)
        {perror("error listening"); exit(-1);}
    printf("Server ready (CTRL-C quits)\n");

    // ciclo che accetta le connessioni
    client_addr_len = sizeof(client_addr);
    while (!terminated &&
           (connect_socket =        //diversa da server_socket!
            accept(server_socket,
                   (struct sockaddr *) & client_addr, // cast
                   &client_addr_len)) != -1 ) {

        // cycle: process data from accepted connection
        printf("Server: new connection from client\n");
        while ((retcode = read(connect_socket, buffer, MAXBUF))
               > 0)      // multiple read()s likely if MAXBUF small
            write(fileno(stdout), buffer, retcode);
        if (retcode == 0)        // fine messaggio
            printf("\nClient connection closed\n");
        close(connect_socket);
    }                    // qui si arriva per errore in
    cleanup();           // accept() o terminated==TRUE
    return(0);
}

void cleanup()
{   close(connect_socket); close(server_socket);
    if (unlink(NOMESOCK) < 0)
        {perror("removing socket");}
    printf("Terminated\n");
    exit(0);
}

#define MOREMSG "need another message to terminate\n"
void terminazione(int signo)
{
    printf("Signal handler started. Terminating ...\n");
// Next 2 lines are alternative; comment out one!
/*  terminated=1; printf(MOREMSG); return; */ // to main's blocking accept()
    cleanup();                    // terminate at once
}
```

Esercizio

Il server di p. 43, dopo il `CTRL-C`, ha ancora bisogno di un messaggio per terminare, se nel codice di `terminazione()` è attivata la prima delle due alternartive.

- Perché?
- Come si può ovviare?
  (Cf. alternativa con `cleanup()` in `terminazione()`)

Client-server in AF_UNIX: client

```
/* unclnt.c */

/* spedisce 1o arg al server, non attende risposta */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define MAXBUF 1024
#define NOMESOCK "servsock"


int main(int argc, char * argv[])
{
    int client_socket;
    int retcode; size_t msglen;
    struct sockaddr_un server_addr;
    char msg[MAXBUF];

    // apertura del socket del client
    client_socket = socket(AF_UNIX, SOCK_STREAM, 0);
    if (client_socket == -1)
        {perror("creating client socket"); exit(1);}

    // preparazione indirizzo su cui connettere il socket
    server_addr.sun_family = AF_UNIX;
    strcpy(server_addr.sun_path, NOMESOCK);

    // connessione al socket del server
    retcode =
    connect(client_socket, (struct sockaddr *) &server_addr,
            sizeof(server_addr) );
    if (retcode == -1)
        {perror("connecting socket"); exit(2);}

    // scrittura del messaggio sul socket
    strcpy(msg, argc > 1 ? argv[1] : "<>");
    msglen = strlen(msg)+1;      // msg e' una stringa
    retcode = write(client_socket, msg, msglen);
    printf("sent %d (%u requested) bytes on socket %d\n",
           retcode, (unsigned) msglen, client_socket);

    close(client_socket);      // chiusura connessione
    exit(0);
}
```

Attesa di un messaggio: tre stili

- bloccante
- non bloccante, con max n. di tentate ricezioni
- bloccante con timeout di uscita, cf. pag. 22

```
/* getmsg.c */

/* await/display message: three waiting styles: blocking,
 * non-blocking with max retry count, blocking with timeout
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>

#include "getmsg.h"

#define MAXBUF 1024

static char buf[MAXBUF];


void getmsg_blk(int sock) // ricezione bloccante di un messaggio
{
    int retcode;

    retcode = read(sock,buf,MAXBUF);
    // NB: un singolo read() in generale non e' corretto
    if (retcode == -1) {
        printf("proc %d on read: %s\n", getpid(),
               strerror(errno));
        exit(-1);
    }
    printf(">>>got %d bytes\n", retcode);
    write(fileno(stdout), buf, retcode);
    printf("\n>>>end\n");
}


/* getmsg.c ... */
```

/* **getmsg.c** (cont.) */

```
void getmsg_max(int sock)
// ricezione non bloccante ripetuta un n. max di volte
{
    int retry = 0;
    int retcode;

    fcntl(sock, F_SETFL, O_NONBLOCK);   //la read non deve essere bloccante.
    do {
        printf("N. retry = %d ", retry);
        retcode = read(sock,buf,MAXBUF);
        if (retcode == -1) {
            if (errno != EAGAIN) {
                perror("Error occurred");
                exit(1);
            } // errno was EAGAIN: retry!
            retry++;
            printf("delaying...\n");
            sleep(1); //tra un tentativo ed un altro inserisco un intervallo
        }
    } while((retcode < 0) && (retry < MAXRETRIES));
    if (retcode <= 0) {
        printf("No data read after %d retries \n", MAXRETRIES);
        exit(-1);
    }
    printf(">>>got %d bytes\n", retcode);
    write(fileno(stdout), buf, retcode);
    printf("\n>>>end\n");
}


extern void getmsg_tout(int sock); // attende messaggio entro timeout

#include "getmsgTout.c"    // codice di getmsg_tout() (usa select())
```

Per `getmsg_tout()`, si veda pag. 22 o il file `examples/socket/getmsgTout.c`

Client in `AF_INET`: client

Uso: `inclnt` *request-string* `[w|r|t]`.

il secondo argomento chiede al cliente di attendere una risposta dal server come segue:

`w` finché una `read` bloccante ha successo

`t` finché una `read` bloccante ha successo o scade un timeout

`r` entro un numero max di fallite `read` non-bloccanti

Per un esempio di server, vedi p. 51.

```
/* inclnt.c */

/* client request/reply: spedisce 1o arg al server
 * e, se ha il 2o arg, aspetta/stampa la risposta */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>      // serve per chiamare strerror()
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h> // attenti a man ip!

#include "getmsg.h"      // per getmsg_xxx()

#define MAXBUF 1024
#define SERVERPORT 3002
#define SERVERNAME "localhost"

void prnerrmsg(char * progname)
{
    fprintf(stderr, "Usage: %s msg [r|t|w]\n", progname);
    fprintf(stderr, "\tsends msg to server ");
    fprintf(stderr, "(msg[0]-'A'==server delay)\n");
    fprintf(stderr, "\tif 3rd arg is present, ");
    fprintf(stderr, "tries to receive reply:\n");
    fprintf(stderr, "\t\tr => retry for given times\n");
    fprintf(stderr, "\t\tt => retry until timeout expires\n");
    fprintf(stderr, "\t\tw => wait reply indefinitely\n");
}

int main(int argc, char * argv[])

/* inclnt.c … */
```

/* **inclnt.c** (cont.) */

```
{
    pid_t my_pid;
    int client_socket;
    int retcode; size_t msglen;
    struct sockaddr_in server_addr;
    char msg[MAXBUF];
    void prnerrmsg(char *);

    if (argc != 2 && argc != 3)
        {prnerrmsg(argv[0]); exit(-1);}

    printf("proc %u ready\n", my_pid = getpid());

    // apertura del socket del client
    client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket == -1)
        {perror("creating client socket"); exit(1);}

    // preparazione indirizzo su cui connettere il socket
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVERPORT);
    struct hostent * hp = gethostbyname(SERVERNAME);
    memcpy(&server_addr.sin_addr, hp->h_addr, hp->h_length);

    // connessione al socket del server
    if (connect( client_socket, (struct sockaddr *)&server_addr,
                 sizeof(server_addr) ) == -1) {
        printf("proc %u on connect: %s\n", my_pid, strerror(errno))
        exit(2);
    }
    printf("proc %u connected, ", my_pid);

    // scrittura del messaggio sul socket
    strcpy(msg, argv[1]);
    msglen = strlen(msg)+1;      // msg e' una stringa
    retcode = write(client_socket, msg, msglen);
    printf("sent %d (%u request) bytes on socket %d\n",
           retcode, (unsigned) msglen, client_socket);

    if (argc == 3) {
        switch (argv[2][0]) {
        case 'r': getmsg_max(client_socket); break;
        case 't': getmsg_tout(client_socket); break;
        case 'w':
        default:  getmsg_blk(client_socket);
    }   }

    close(client_socket);  // chiusura connessione
    exit(0);
}
```

Server in AF_INET

Il client può essere quello di p. 49.

```
/* insrvr.c */


#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>    // vedi ”man 7” ip, non ”man ip”!
#include <string.h>

#include <signal.h>

void catchSig(int), cleanup();

#define MAXBUF 16
#define SERVERPORT 3002
#define SERVERNAME "localhost"

char buffer[MAXBUF];    // global, so doserv() can access it
void doserv(void);      // esegue servizio elaborando dati in buffer[]
int mkaddr(struct sockaddr_in *, const char *, u_int16_t);

int main(int argc, char * argv[])
{
    int server_socket, connect_socket;
    socklen_t client_addr_len;
    int retcode, nw;
    struct sockaddr_in client_addr, server_addr;

    signal(SIGPIPE,catchSig);

    // apertura del socket del server
    if ( (server_socket = socket(AF_INET,SOCK_STREAM,0)) == -1)
        {perror("opening server socket"); exit(-1);}

    // preparazione indirizzo locale (server) per il socket
    mkaddr(&server_addr, SERVERNAME, htons(SERVERPORT));

/* insrvr.c ... */
```

/* **insrvr.c** (cont.) */

```
    // pubblicazione socket / listen
    retcode =
    bind(server_socket,
         (struct sockaddr *) &server_addr, //NB: type cast
         sizeof(server_addr) );
    if(retcode == -1)
        {perror("error binding socket"); exit(-1);}
    retcode = listen(server_socket, 1);
    if(retcode == -1)
        {perror("error listening"); exit(-1);}
    printf("Server ready (CTRL-C quits)\n");

    // ciclo che accetta le connessioni
    client_addr_len = sizeof(client_addr);
    while ( 1 ) {
        if ((connect_socket =       // diversa da server_socket!
             accept(server_socket,
                    (struct sockaddr *) & client_addr, // cast
                    &client_addr_len)) == -1) {
            perror("accepting");
            close(server_socket);
            exit(-1);
        }
        printf("\nClient@%s connects on socket %d; sends:\n",
               inet_ntoa(client_addr.sin_addr),connect_socket);

        // cycle: process data from accepted connection
        while ((retcode = read(connect_socket, buffer, MAXBUF-1))
               > 0) {               // leave a byte for NULL in buffer[]
            buffer[retcode] = '\0'; // buffer must be a string
            printf("%s\n", buffer);
            doserv();       // esegue servizio operando su buffer[], che
            printf("here\n");
            if ((nw = write(connect_socket,       // invia al cliente collegato
                            buffer,strlen(buffer)+1)) < 0) {
                perror("replying to client");  // error check incompleto
            }
            printf("written %d\n", nw);
        }
        if (retcode == 0 || errno == EPIPE) // read returned 0 or -1
            printf("Client closed connection on socket %d\n",
                   connect_socket);
        else // retcode == -1 && (errno != EPIPE)
            perror(">>reading from connection");
        close(connect_socket);// prossima connect_socket riutilizza fd
                              //
    } // end while(1) (never gets here)
}
```

```
void catchSig(int signo)
{
    printf("Signal handler %d\n", signo);
}
```

Servizio conversione in maiuscolo

Il servizio invocato da `insrvr` è definito nella funzione `doserv()`, nel file seguente:

```
/* doserv.c */

/* servizio in AF_INET: converte in maiuscolo;
 * ritardo = ASCII 1o char ricevuto - 'A';
 */
#include <ctype.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>

extern char buffer[];    // defined in server's code

void doserv(void)
{
    int i, delay;

    // Esegue servizio maiuscole
    for (i = 0; i < strlen(buffer) ; i++)
        buffer[i] = toupper(buffer[i]);
    // simula un ritardo
    if ( (delay = buffer[0]-'A') < 0)     // delay 0 per A, 1 per B,...
        delay = 1000;  // very big delay
    printf("\nServer delay %d\n",delay);
    sleep((unsigned) delay);
}
```

Il tempo di servizio dipende dal 1° carattere ricevuto (p.es. attraverso il messaggio inviato dal client di pag. 49):

- se il 1° carattere è una lettera, il tempo sarà: 0 per `A` o `a`, 1 per `B` o `b`, . . . , 25 per `Z` o `z`;
- se no, sarà 1000 secondi.

Ciò permette di simulare i tempi di computazione di un “vero” server.

Server seriale

Il server di pag. 51, serve le richieste dei clienti una alla volta: finita quella in corso, prende in considerazione la prossima:

- se è in corso il servizio di una richiesta lunga di un cliente (p.es. `inclnt 1000`), ogni altra richiesta dovrà aspettare.
- più precisamente, mentre il server `insrvr` è dentro `doserv()`, la `connect()` del (nuovo) client deve attendere
- come detto a pag. 11 il n.max di `connect()` che può stare in attesa dipende dal 2° arg. `backlog` di `listen()` nel server

Per `backlog==1`, con il server impegnato attraverso:

```
$ ./inclnt 1000   // ritardo server == 100 sec
```

e nuove richieste multiple parallele

```
$ rm for.out
$ for i in 1..8; do (./inclnt avanti &>>for.out)& done
```

si hanno problemi:

```
$ cat for.out
proc 13781 on connect: Connection timed out
proc 13778 on connect: Connection timed out
proc 13783 on connect: Connection timed out
proc 13777 connected, sent 7 (7 request) bytes on socket 3
proc 13777 on read: Connection reset by peer
proc 13784 connected, sent 7 (7 request) bytes on socket 3
proc 13784 on read: Connection reset by peer
proc 13785 connected, sent 7 (7 request) bytes on socket 3
proc 13785 on read: Connection reset by peer
proc 13779 connected, sent 7 (7 request) bytes on socket 3
proc 13779 on read: Connection reset by peer
```

$\Rightarrow$ per alcuni processi `connect()` procede anche se `insrvr` non esegue la corrispondente `accept()`, che avverrà dopo

la successiva `read()` fallirà con `Connection reset by peer`; perché?

in effetti il server non termina, forse dopo un timeout l'`accept()` pendente evolve in una chiusura della connessione (mai stabilita dal lato server, ma apparentemente stabilita per il client)

- di quante `accept()` può restare indietro `insrvr`?

  Dipende dal parametro `backlog` di `listen()` in `insrvr`

- per altri processi, che trovano la coda per `accept()` piena, la `connect()` del client fallisce con `ETIMEDOUT`.

  e `ECONNREFUSED`??? c'entra qualcosa? Dal man:

  The backlog argument defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds.

  The behavior of the backlog argument on TCP sockets changed with Linux 2.2. Now it specifies the queue length for completely established sockets waiting to be accepted, instead of the number of incomplete connection requests.

  controllare le socket in gioco — le `accept()` "tacite" (se esistono?) generano altrettanti socket cloni?

Con `backlog==2`, sul mio sistema, la coda di `accept()` regge tutte le 8 richieste in attesa.

spiegare gli zombie

mi pare il ciclo di read nel server sia discutibile! capirlo meglio

Server parallelo

```
/* inparsrvr.c */

/* versione parallela di insrvr.c
 * le differenze sono marcate da commenti "///"
 * Provare a chiedere un servizio con ritardo 25 sec
 * e poi uno con ritardo 0: questo verra' fornito subito
 */

#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>    // vedi "man 7" ip, non "man ip"!
#include <string.h>

#define MAXBUF 1024
#define SERVERPORT 3002
#define SERVERNAME "localhost"

char buffer[MAXBUF];    // global, so doserv() can access it
void doserv(void);      // esegue servizio elaborando dati in buffer[]
int mkaddr(struct sockaddr_in *, const char *, u_int16_t);

int main(int argc, char * argv[])
{
    int server_socket, connect_socket;
    socklen_t client_addr_len;
    int retcode;
    struct sockaddr_in client_addr, server_addr;

    // apertura del socket del server
    if ( (server_socket = socket(AF_INET,SOCK_STREAM,0)) == -1)
        {perror("opening server socket"); exit(-1);}

    // preparazione indirizzo locale (server) per il socket
    mkaddr(&server_addr, SERVERNAME, htons(SERVERPORT));

/* inparsrvr.c ... */
```

/* **inparsrvr.c** (cont.) */

```
    // pubblicazione socket / listen
    retcode =
    bind(server_socket,
         (struct sockaddr *) &server_addr, //NB: type cast
         sizeof(server_addr) );
    if(retcode == -1)
        {perror("error binding socket"); exit(-1);}
    retcode = listen(server_socket, 1);
    if(retcode == -1)
        {perror("error listening"); exit(-1);}
    printf("Server ready (CTRL-C quits)\n");

    // ciclo che accetta le connessioni
    client_addr_len = sizeof(client_addr);
    while ( 1 ) {
        if ((connect_socket =        // diversa da server_socket!
             accept(server_socket,
                    (struct sockaddr *) & client_addr, // cast
                    &client_addr_len)) == -1) {
            perror("accepting");
            close(server_socket);
            exit(-1);
        }
    if (fork() == 0) {  //figlio: esegue servizio
        printf("\nClient@%s connects on socket %d; sends:\n",
               inet_ntoa(client_addr.sin_addr),connect_socket);

        // cycle: process data from accepted connection
        while ((retcode = read(connect_socket, buffer, MAXBUF-1))
               > 0) {                // leave a byte for NULL in buffer[]
            buffer[retcode] = '\0'; // buffer must be a string
            printf("%s\n", buffer);
            doserv();        // esegue servizio operando su buffer[], che
            if (write(connect_socket,       // invia al cliente collegato
                      buffer,strlen(buffer)+1) < 0)
                perror("replying to client");  // error check incompleto
        }
        if (retcode == 0 || errno == EPIPE) // read returned 0 or -1
            printf("Client closed connection on socket %d\n",
                   connect_socket);
        else // retcode == -1 && (errno != EPIPE)
            perror(">>reading from connection");
        exit(0);// figlio: termina
    }
    else         // padre: riprende ad ascoltare su accept() dopo aver chiuso la socket
        close(connect_socket);// prossima connect_socket riutilizza fd
                              // NB: inutile in figlio (terminando chiude cmq)
    } // end while(1) (never gets here)
}
```

Server-client parallelo

Server *stateless*:

- i processi figli “subserver” servono una richiesta ed escono
- la risposta non dipende dalla storia (richieste) passata
- altrimenti, il server sarebbe *stateful* (p.es. servizio di prelievo/query di cc bancario)
  - in questo caso la computazione dei subserver dovrebbe dipendere dalla comunicazione con il server “padre”, che detiene lo stato (p.es. saldo cc)
  - problema: sincronizzazione padre-subserver (multipli)

Esercizio:

- osservare come sia per il server seriale di pag. 51 che per quello parallelo di pag. 57, per via di `doserv()`, il tempo di servizio dipende dal 1° carattere ricevuto (p.es. attraverso il messaggio inviato dal client di pag. 49)
- osservare come, con il server parallelo, richieste veloci di un cliente possano ricevere subito risposta, anche se è pendente il servizio di una richiesta lunga di un altro cliente;

Client-server connectionless: server

```
/* sndrcvClessSrvr.c */
/* Datagram socket based connectionless server */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <signal.h>

#define MAXBUF 1024
#define SERVERPORT 3002
#define SERVERNAME "localhost"

void terminate(int);
int terminated = 0;
int server_socket;       // global so all functions see it

int main(int argc, char * argv[])
{
    int retcode, i;
    struct sockaddr_in client_addr, server_addr;
    int c_addr_l = sizeof(client_addr);
    char buffer[MAXBUF];

    // inizializzazione
    signal(SIGINT, terminate);

    // apertura del socket del server
    if ( (server_socket = socket(AF_INET, SOCK_DGRAM, 0))
         == -1)
        {perror("opening server socket"); exit(-1);}

    // preparazione indirizzo locale (server)
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVERPORT);
    memcpy(&server_addr.sin_addr,
           gethostbyname(SERVERNAME)->h_addr,
           sizeof(server_addr.sin_addr)      );

/* sndrcvClessSrvr.c ... */
```

```
/* sndrcvClessSrvr.c (cont.) */

    retcode =   // NB: senza bind() i recvfrom del client falliscono
    bind(server_socket,
         (struct sockaddr *) &server_addr, //NB: type cast
         sizeof(server_addr)
    );
    if(retcode == -1)
        {perror("binding socket"); exit(-1);}
    printf("Server ready (uscire con CTRL-C)\n");

    while (1) {
        // riceve richiesta da client
        retcode =
        recvfrom( server_socket, buffer, MAXBUF, 0,
                  (struct sockaddr *) &client_addr,
                  &c_addr_l );
        if (retcode == -1) {     // recvfrom può fallire perché
            if (terminated)      // terminate() ha chiuso la socket
                break;           // quindi si esce normalmente
            perror("receiving");// o per altre cause
            exit(-1);
        }
        printf("\n\n%d bytes from client %s:\nReplying...\n",
               retcode, inet_ntoa(client_addr.sin_addr));
        buffer[retcode] = 0;     // buffer deve essere una stringa

        // Esegue servizio maiuscole e risponde al client
        for (i = 0; i < strlen(buffer); i++)
            buffer[i] = toupper(buffer[i]);
        retcode =
        sendto(server_socket, buffer, strlen(buffer)+1, 0,
               (struct sockaddr *) &client_addr, c_addr_l);
        if (retcode == -1)
            { perror("sending"); exit(-1); }
        printf("Sent %d bytes back\n", retcode);
    }

    printf("\nServer terminated (by CTRL-C) \n");
    exit(0);
}


void terminate(int signo)
{
    // termina facendo fallire recvfrom e terminare il main loop
    if (close(server_socket) == -1)
        {perror("closing socket"); exit(-1); }
    terminated = 1;
}
```

Client-server connectionless: client

```
/* sndrcvClessClnt.c */
/* Datagram socket based connectionless client */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MAXBUF 1024
#define SERVERPORT 3002
#define SERVERNAME "localhost"

int main(int argc, char * argv[])
{
    int client_socket, retcode, s_addr_len;
    struct sockaddr_in server_addr;
    char msg[MAXBUF];

    // apertura del socket del client
    client_socket = socket(AF_INET, SOCK_DGRAM, 0);
    if (client_socket == -1)
        {perror("opening client socket"); exit(-1);}

    // preparazione indirizzo del server remoto
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVERPORT);
    memcpy(&server_addr.sin_addr,
           gethostbyname(SERVERNAME)->h_addr,
           sizeof(server_addr.sin_addr)      );
    s_addr_len = sizeof(server_addr);

/* sndrcvClessClnt.c ... */
```

/* **sndrcvClessClnt.c** (cont.) */

```
    // invio del messaggio al server
    strcpy(msg, argc > 1 ? argv[1] : "<>");      // msg e' di certo una stringa
    retcode = sendto( client_socket, msg, strlen(msg)+1, 0,
                      (struct sockaddr *) &server_addr,
                      s_addr_len );
    if (retcode == -1)
        {perror("sending"); exit(-1); }

    // overwrite server_addr (check recvfrom's effect)
    memset(&server_addr, 0, s_addr_len);

    // riceve risposta dal server
    retcode = recvfrom( client_socket, msg, strlen(msg)+1, 0,
                        (struct sockaddr *) &server_addr,
                        & s_addr_len );
    if (retcode == -1)
        {perror("receiving"); exit(-1); }
    msg[retcode] = '\0';          // msg must be a string
    printf( "\nServer %s replies %d bytes:\n%s\n\n",
            inet_ntoa(server_addr.sin_addr),
            retcode, msg );

    close(client_socket);
    exit(0);
}
```

Client alternativo con `snd()`/`rcv()`

```
/* sndrcvConnClnt.c */

/* client like the connectionless clessclnt.c;
 * but ... uses connect() with send() and recv()
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MAXBUF 1024
#define SERVERPORT 3002
#define SERVERNAME "localhost"

int main(int argc, char * argv[])
{
    int client_socket, retcode, s_addr_len;
    struct sockaddr_in server_addr;
    char msg[MAXBUF];

    // apertura del socket del client
    client_socket = socket(AF_INET, SOCK_DGRAM, 0);
    if (client_socket == -1)
        {perror("opening client socket"); exit(-1);}

    // preparazione indirizzo del server remoto
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVERPORT);
    memcpy(&server_addr.sin_addr,
           gethostbyname(SERVERNAME)->h_addr,
           sizeof(server_addr.sin_addr)      );
    s_addr_len = sizeof(server_addr);

    retcode = connect(client_socket,            // not in clessclnt.c
            (struct sockaddr *) &server_addr, s_addr_len );
    if (retcode == -1)
        {perror("connecting socket"); exit(-1);}


/* sndrcvConnClnt.c ... */
```

```
/* sndrcvConnClnt.c (cont.) */

    // invio del messaggio al server
    strcpy(msg, argc > 1 ? argv[1] : "<>");      // msg must be a string
    retcode = send( client_socket, msg, strlen(msg)+1, 0);
    if (retcode == -1)
        {perror("sending"); exit(-1); }

    // should not overwrite server_addr here (clessclnt.c does)
//  memset(&server_addr, 0, s_addr_len);

    // riceve risposta dal server
    retcode = recv( client_socket, msg, strlen(msg)+1, 0);
    if (retcode == -1)
        {perror("receiving"); exit(-1); }
    msg[retcode] = '\0';          // msg must be a string
    printf( "\nServer %s replies %d bytes:\n%s\n\n",
            inet_ntoa(server_addr.sin_addr),
            retcode, msg );

    close(client_socket);
    exit(0);
}
```