

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)
- SOCK_RDM endpoint per *reliable* datagram

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

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
<code>s = socket(...);</code>	<code>t = socket(...);</code>
<code>bind(s, &myAddr, addrLen);</code>	
<code>listen(s, pendingQSize);</code>	
<code>s1 = accept(s, ...);</code>	<code>connect(t, &servrAddr, addrLen);</code>

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` (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 determina la semantica della comunicazione; in genere vale una delle costanti:

- SOCK_STREAM: semantica tipo file (sequenced, reliable), a due vie e con messaggi “out-of-band”
- SOCK_DGRAM: semantica per messaggi “datagram” (connectionless, unreliable)

protocol ; se 0 seleziona il protocollo di default (spesso l'unico) per il domain.

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>

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 esempio a p. 69.

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).

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 (dell'altra parte 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`)

Letture 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. 36) 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` `s_i` (man `ip`)
- 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. 37) 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. 74):

```
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. 62.

Considerazioni analoghe per `recv` (p. 41).

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

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>
int send(int s, const void *msg, int len,
         unsigned int flags);
int sendto(int s, const void *msg, int len,
           unsigned int flags,
           const struct sockaddr *to, int tolen);
```

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>
int recv(int s, void *buf, int len, unsigned int flags);
int recvfrom(int s, void *buf, int len, unsigned int flags,
             struct sockaddr *from, int *fromlen);
```

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. 29) (cf. `EAGAIN`, p. 42)

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, ...);
```

si può usare anche per socket sockfd datagram.

Da completare.

Select

Vedi esempio `examples/socket/udpclient_select.c`.

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 4 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:

1. client: connect da 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, per la convenzione little endian:

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

Viceversa, per porre in `addr` l'address 0x974afd25=151.74.253.37:

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

Conversione tra formati di IP address

Conversioni tra formati di rete e locali (host):

```
#include <netinet/in.h>

unsigned long int htonl(unsigned long int hostlong);
unsigned short int htons(unsigned short int hostshort);
unsigned long int ntohl(unsigned long int netlong);
unsigned short int ntohs(unsigned short int netshort);
```

Come detto in `netinet/in.h`, i tipi dovrebbero essere `uint32_t` e `uint16_t`.

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 {
    unsigned long int s_addr; //network byte order
}
```

Conversione tra formati: esempio

```
/* addrconv.c */

#include <stdio.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("*p is %s\n", p);
    inet_aton("255.255.255.255", &sa);
    inet_ntoa(sa);
    printf("*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/host.conf`
- 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 server DNS elencati in `/etc/resolv.conf`.

Nomenclatura Unix relativa a DNS:

BIND (Berkeley Internet Name Domain): implementazione di DNS in Unix BSD
named (**name daemon**): 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. 59):

- `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 1o byte dell'IP address (a 4 byte):
p.es. `*h_addr_list[0]` è il 1o byte del 1o 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 1o IP address,
cioè `*((uint32_t *) h_addr_list[0])` è il 1o 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. 59):

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

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

Esempio (vedi anche p. 59):

```
/* byaddr.c */  
/* test gethostbyname(), gethostbyaddr() */  
  
#include <stdio.h>  
#include <netdb.h>  
#include <arpa/inet.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

Semplice interfaccia al resolver

V. p. 52 per `printaddr()`.

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

#include <stdio.h>
#include <ctype.h>
#include <netdb.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);
        perror("Resolver"); // perror() 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);
        perror("Resolver");
        exit(h_errno);
    }
    printaddr(hq); putchar('\n');
    exit(0);
}
```

Finger client

```
/* fingerc.c */

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

#include <netdb.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <netinet/in.h>

#define MAXHN 128
#define MAXBUF 64*1024 // provare a variare

int main(int argc, char * argv[])
{
    char hostname[MAXHN], buffer[MAXBUF];
    struct hostent * host;
    struct sockaddr_in addr;
    u_int16_t fport;
    int s, retcode;
    unsigned int nbytes;

    if(argc != 3 && argc != 4) {
        printf("usage: %s host message [port]\n", argv[0]);
        exit(1);
    }
    strcpy(hostname, argv[1]);
    sprintf(buffer, "%s\n", argv[2]);
    fport = argc == 4 ? htons(atoi(argv[3]))
        : htons(IPPORT_FINGER); // in netinet/in.h

    // construct remote address
    addr.sin_family = AF_INET;
    addr.sin_port = fport;

    host = gethostbyname(hostname);
    memcpy(&addr.sin_addr, host->h_addr_list[0],
        nbytes = sizeof(host->h_addr_list[0]));
    printf("\t(requested IP address is %u bytes long)\n", nbytes);

    if ((s = socket(AF_INET, SOCK_STREAM, 0)) == -1)
        {perror("opening socket"); exit(-1);}
    if (connect(s, (struct sockaddr *) &addr, sizeof(addr))
        == -1 ) {perror("connecting"); exit(-2);}

/* fingerc.c ... */
```

```

/* fingerc.c (cont.) */

if (write(s, buffer, strlen(buffer)) == -1)
    {perror("writing to socket"); exit(-3);}

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

while ((retcode = read(s, buffer, MAXBUF)) != 0) {
    if (retcode == -1)
        {perror("reading from socket"); exit(-4); }
    printf("\n----Read %d bytes----\n", retcode);
    write(fileno(stdout),buffer,retcode);
}
printf("\n>>>Exiting: server closed connection\n");
close(s); exit(0);
}

```

Semantica di `read`: esempi

Come segnalato nel commento al cliente di `finger` (p. 60), una sola `read` non è quasi mai appropriata.

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

Implicazioni della semantica di `read` (v. p 37):

- 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 `fi ngers` (p. 63)
`fi ngers` 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>

#define LOCIP "127.0.0.1"
#define MAXQ 512
#define MYFINGERPORT 2000
#define MAXBUF 1024
#define NITER 10
#define MANYBYTES 8*1024
#define DELAY 7

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

    s = socket(PF_INET, SOCK_STREAM, 0);

    // Costruzione indirizzo socket
    locAddr.sin_family = AF_INET;
    locAddr.sin_port = htons(MYFINGERPORT);
    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 connected\n",
           inet_ntoa(farAddr.sin_addr));
    // read one message but 1 read 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));
    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));
    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 gestire SIGINT
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 for multiple reads to 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

int main(int argc, char * argv[])
{
    int client_addr_len, retcode;
    struct sockaddr_un client_addr, server_addr;
    char line[MAXBUF];

    // 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 socket
    server_addr.sun_family = AF_UNIX;
    strcpy(server_addr.sun_path, NOMESOCK);
    unlink(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("publishing socket"); exit(-1);}
printf("Server ready (^C to terminate)\n");
listen(server_socket, 1);

// 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, line, MAXBUF))
           != 0) { // multiple read(s) likely if MAXBUF small
        write(fileno(stdout), line, retcode);
        if (retcode == 0) // fi ne messaggio
            break;
    }
    printf("\nClient connection closed\n");
    close(connect_socket);
} // qui si arriva per errore in
cleanup(); // accept() o terminated==TRUE
}

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");
    terminated = 1;
// Next 2 lines are alternative; comment out one!
    printf(MOREMSG); return; // return to blocking read() in main()
// cleanup(); // terminate at once
}

```

Esercizio

Il server di p. 65, dopo il CTRL-C ha ancora bisogno di un messaggio per terminare.

- Perché?
- Come si può ovviare?

Client-server in AF_UNIX: client

```
/* unclnt.c */
/* client che spedisce brevi messaggi al server, UNIX domain */

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

#define NOMEMSOCK "servsock"
#define MAXBUF 1024

int main(int argc, char * argv[])
{
    int client_socket;
    int retcode;
    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("aprendo il socket del client"); exit(-1);}

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

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

    // scrittura del messaggio sul socket
    strcpy(msg, argc > 1 ? argv[1] : "<>");
    retcode = write(client_socket, msg, strlen(msg)+1);
    printf("Client: scritti %d (%d) bytes sul socket\n",
           retcode, strlen(msg)+1);

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

Attesa di un messaggio: tre stili

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

```
/* getmsg.c */

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

#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>

#define MAXBUF 1024
#define RETRIES 50          // upper bound for non-blocking reads
#define TOUTS 5
#define TOUTMS 5

static char buf[MAXBUF];

void getmsg_blk(int sock)
{
    int retcode;

    retcode = read(sock,buf,MAXBUF);
    if (retcode == -1) {
        perror("reading");
        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)
```

```
{  
    printf("\nshould try to read reply up to RETRIES times\n");  
    printf("\tto be implemented");  
    printf(" (by nonblocking read/receive)\n");  
    printf("\tsee e.g. udpcnt_nowait.c\n");  
}
```

```
void getmsg_tout(int sock)
```

```
{  
    printf("\nshould wait for reply until");  
    printf(" timeout TOUTS+TOUTMS expires\n");  
    printf("\tto be implemented (by select())\n");  
    printf("\tsee e.g. udpcnt_select.c\n");  
}
```

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. 74.

```
/* inclnt.c */

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

#include <stdio.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\n\tif 3rd arg is");
    fprintf(stderr, " present also tries to receive reply:\n");
    fprintf(stderr, "\t\ttr => retry for given times\n");
    fprintf(stderr, "\t\ttt => retry until timeout expires\n");
    fprintf(stderr, "\t\ttw => wait reply indefinitely\n");
}

/* inclnt.c ... */
```

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

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

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

/* inclnt.c ... */
```

```

/* inclnt.c (cont.) */

// 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);
memcpy(&server_addr.sin_addr,
       gethostbyname(SERVERNAME)->h_addr,
       sizeof(server_addr.sin_addr) );

// 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;
retcode = write(client_socket, msg, msglen);
printf("sent %d (%d requested) bytes\n", retcode, msglen);

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);
    }
}

retcode = close(client_socket);
exit(0);
}

```

Server in AF_INET (converte in maiuscolo)

Il client può essere quello di p. 71 con opzione reply.

```
/* insrvr.c */

/* servizio in AF_INET: converte in maiuscolo;
 * ritardo = ASCII 1o char ricevuto - 'A';
 */

#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> // attenti a man ip!
#include <ctype.h>

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

char buffer[MAXBUF];
void doserv(int sock);

int main(int argc, char * argv[])
{
    int server_socket, connect_socket;
    int client_addr_len, retcode;
    struct sockaddr_in client_addr, // NB: type cast later
                    server_addr;
    char * addrp;

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

    // preparazione dell'indirizzo per il socket
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVERPORT);
    addrp = gethostbyname(SERVERNAME)->h_addr;
    memcpy(&server_addr.sin_addr, addrp, sizeof(addrp));

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

```

/* insrvr.c (cont.) */

// pubblicazione del socket
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 ( (connect_socket = // diversa da server_socket!
        accept(server_socket,
                (struct sockaddr *) & client_addr, // cast
                &client_addr_len)) ) {
    if (connect_socket == -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))
           > 0) {
        write(fileno(stdout), buffer, retcode);
        doserv(connect_socket);
    }
    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);
} // close accept cycle
exit(0);
}

/* insrvr.c ... */

```

```

/* insrvr.c (cont.) */

void doserv(int sock)
{
    int i;
    unsigned delay;

    // Esegue servizio maiuscole
    for (i = 0; i < strlen(buffer); i++)
        buffer[i] = toupper(buffer[i]);
    // simula un ritardo
    if ( (delay = (unsigned) (buffer[0]-'A')) < 30 )
        {printf("\nServer delay %u\n",delay); sleep(delay);}
    // risponde
    if (write(sock, buffer, strlen(buffer)+1) < 0)
        perror("replying to client");
}

```

Server parallelo per convertire in maiuscolo

```
/* inparsrvr.c */

/* versione parallela di insrvr.c
 * le differenze sono marcate da commenti "figlio:" e "padre:"
 * Provare a chiedere un servizio con ritardo 30 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> // attenti a man ip!
#include <ctype.h>

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

char buffer[MAXBUF];
void doserv(int sock);

int main(int argc, char * argv[])
{
    int server_socket, connect_socket;
    int client_addr_len, retcode;
    struct sockaddr_in client_addr, // NB: type cast later
                    server_addr;
    char * addrp;

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

    // preparazione dell'indirizzo per il socket
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(SERVERPORT);
    addrp = gethostbyname(SERVERNAME)->h_addr;
    memcpy(&server_addr.sin_addr, addrp, sizeof(addrp));

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

```

/* inparsrvr.c (cont.) */

// pubblicazione del socket
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 ( (connect_socket = // diversa da server_socket!
        accept(server_socket,
                (struct sockaddr *) & client_addr, // cast
                &client_addr_len)) ) {
    if (connect_socket == -1) {
        perror("accepting");
        close(server_socket);
        exit(-1);
    }
    if (fork() == 0) { //fi glio: 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))
                > 0) {
            write(fileno(stdout), buffer, retcode);
            doserv(connect_socket);
        }
        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);
    exit(0); // fi glio: esce, servizio eseguito
} // fi glio: endif
else
    close(connect_socket);
} // close accept cycle
}

/* inparsrvr.c ... */

```

```

/* inparsrvr.c (cont.) */

void doserv(int sock)
{
    int i;
    unsigned delay;

    // Esegue servizio maiuscole
    for (i = 0; i < strlen(buffer); i++)
        buffer[i] = toupper(buffer[i]);
    // simula un ritardo
    if ( (delay = (unsigned) (buffer[0]-'A')) < 30 )
        {printf("\nServer delay %u\n",delay); sleep(delay);}
    // risponde
    if (write(sock, buffer, strlen(buffer)+1) < 0)
        perror("replying to client");
}

```

Server-client parallelo

Server *stateless*:

- i processi “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

Esercizio:

- osservare come il 1o carattere del 1o argomento del client specifichi il tempo di computazione che il server impiegherà;
- fare eseguire al server `sleep` per il tempo specificato dalla richiesta ricevuta dal client;
- osservare come, con il server parallelo, richieste veloci possano ricevere subito risposta, anche se è pendente il servizio di richieste lunghe.

Client-server connectionless: server

```
/* clessrvr.c */
/* 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) );

/* clessrvr.c ... */
```

```

/* clessrvr.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));

    // 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

```
/* clessclnt.c */
/* 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);

/* clessclnt.c ... */
```

```

/* clessclnt.c (cont.) */

// invio del messaggio al server
strcpy(msg, argc > 1 ? argv[1] : "<>");
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); }
printf( "\nServer %s replies %d bytes:\n%s\n\n",
        inet_ntoa(server_addr.sin_addr),
        retcode, msg );

close(client_socket);
exit(0);
}

```

Client-server connectionless con connect (): client

```
/* clessclntconn.c */

/* connectionless client like 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);}

/* clessclntconn.c ... */
```

```

/* clessclntconn.c (cont.) */

// invio del messaggio al server
strcpy(msg, argc > 1 ? argv[1] : "<>");
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); }
printf( "\nServer %s replies %d bytes:\n%s\n\n",
        inet_ntoa(server_addr.sin_addr),
        retcode, msg );

close(client_socket);
exit(0);
}

```