Introduzione ai socket

Socket TCP

Contiene lucidi tratti da: 2006-2007 Marco Faella, Clemente Galdi, Giovanni Schmid (Università di Napoli Federico II), 2004-2005 Walter Crescenzi(Università di Roma 3).

Indirizzi TCP/IP

in netinet/ip.h:

```
struct sockaddr_in {
        sa_family_t     sin_family;
        u_int16_t       sin_port;
        struct in_addr  sin_addr;
};

struct in_addr {
        u_int32_t       s_addr;
};
```

vedere: man 7 ip

AF_INET

porta, in network order

indirizzo IP, in network order

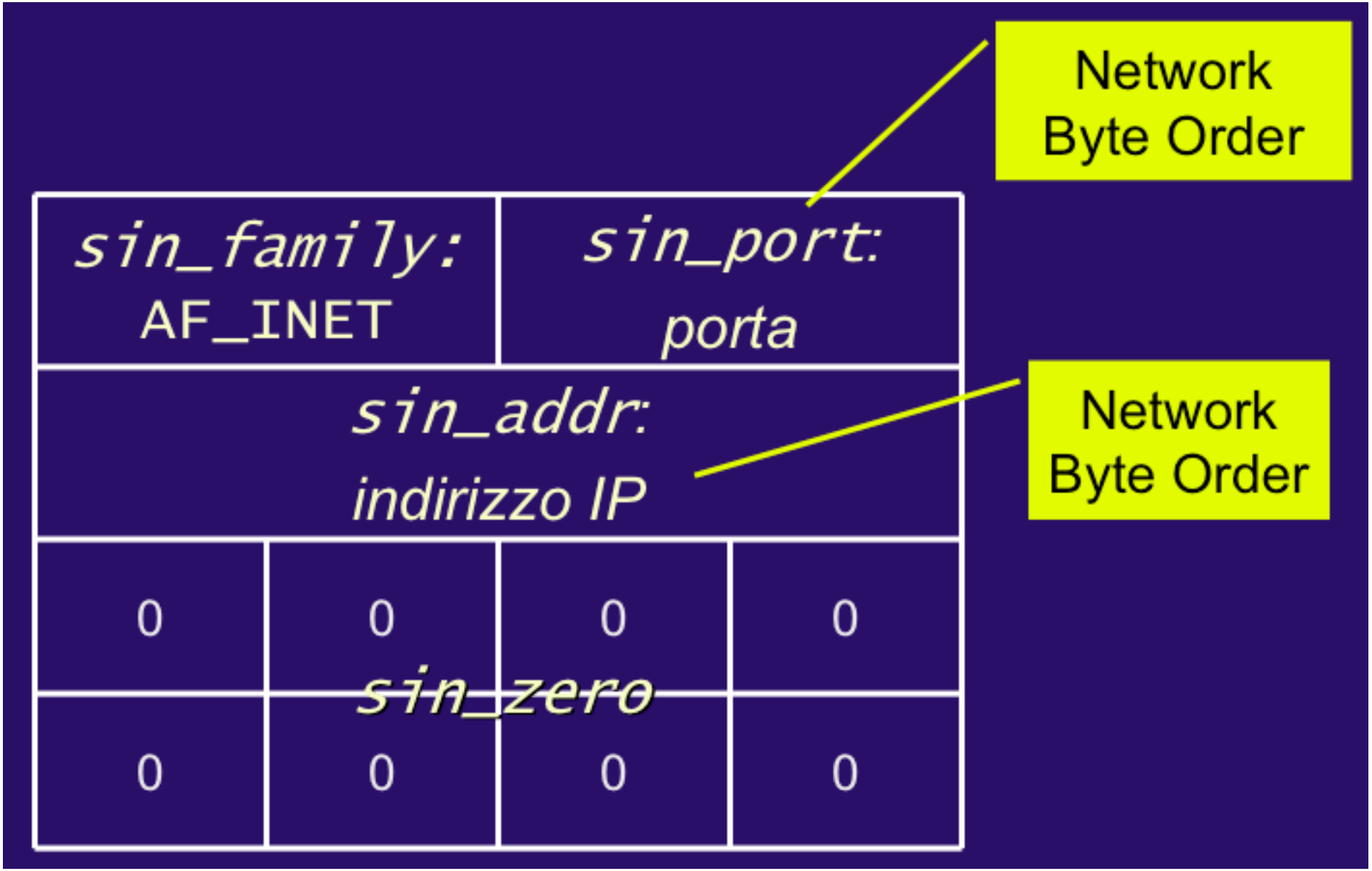
Indirizzi TCP/IP

- Siamo interessati a `sa_family = AF_INET`, cioè al dominio “internet”
- In tal caso servono 2 byte per un numero di porta e 4 byte per un indirizzo `IP`

```
#include <netinet/in.h>
struct sockaddr_in {
    short int sin_family;          // Address family
    unsigned short int sin_port;   // Port number
    struct in_addr sin_addr;       // IP Internet Address
    unsigned char sin_zero[8];     // Stessi byte di…
};                                 // …struct sockaddr
```

14 bytes

Indrizzi TCP/IP



Indirizzi TCP/IP

- Identificati da:

  1) un indirizzo IP (intero a 32 bit, es. 143.225.5.3)

  2) una porta (intero a 16 bit, da 0 a 65535)

- Porte:
  - per offrire diversi servizi dallo stesso indirizzo IP
  - da 0 a 1023: porte riservate (ai processi di root)
  - da 5000 a 32768: porte utente
  - altre: porte effimere (per i client, ai quali non interessa scegliere una porta specifica)

Esempi di porte riservate

- 21 ftp (trasferimento file)
- 22 ssh (login remoto sicuro)
- 25 smtp (invio email)
- 79 finger (informazioni sugli utenti)
- 80 http (web)
- 143 imap (lettura email)

- Lista ufficiale su: http://www.iana.org/

Ordine dei byte

- Come viene memorizzato un intero in memoria?
  - sono usati 4 bytes
  - consideriamo 65537, cioe' $2^{16} + 1$, all'indirizzo 100
  - codifica “Big Endian” (prima il byte *piu'* significativo):

00000000	00000000	10000000	00000001
100	101	102	103

  - codifica “Little Endian” (prima il byte *meno* significativo):

00000001	10000000	00000000	00000000
100	101	102	103

Ordine byte

+ byte più significativo ← byte meno significativo -

513 su 4 byte = **00000000 00000000 00000010 00000001**

Indirizzo	Big-Endian	Little-Endian
0000	**00000000**	**00000001**
0001	**00000000**	**00000010**
0002	**00000010**	**00000000**
0003	**00000001**	**00000000**

network byte order,
Motorola 680X0

Intel 80x86,
Pentium

???

host byte order

Ordine dei byte

- i processori possono essere di entrambi i tipi
  - i protocolli TCP/IP lavorano con **big endian**
  - sono necessarie funzioni di **conversione**

- conversioni tra unsigned:
  - #include <netinet/in.h>
  - uint32_t **htonl**(uint32_t x)
  - uint16_t **htons**(uint16_t x)
  - uint32_t **ntohl**(uint32_t x)
  - uint16_t **ntohs**(uint16_t x)

h	= Host
n	= Network (big endian)
l	= Long (4 bytes)
s	= Short (2 bytes)

Specificare indirizzi IP

- Usando la notazione *dotted* (puntata)

```
struct sockaddr_in indirizzo;

if (inet_aton("143.225.5.3", &indirizzo.sin_addr) == 0)
        perror("inet_aton"), exit(1);
```

- inet_aton (*ascii to network*)
- riempie direttamente una struttura in_addr
- restituisce 0 in caso di errore!

Impostare indirizzo

```
struct sockaddr_in my_addr;

// host byte order
my_addr.sin_family = AF_INET;

// short, network byte order
my_addr.sin_port = htons(MYPORT);

// long, network byte order
inet_aton("10.12.110.57", &(my_addr.sin_addr));

// a zero tutto il resto
memset(&(my_addr.sin_zero), '\0', 8);
```

Indirizzi TCP/IP per il server (bind)

- Il server chiama bind per stabilire su quale indirizzo mettersi in ascolto
- Di solito, il server sceglie solo la porta
- Come indirizzo IP, sceglie INADDR_ANY, cosi' accetta connessioni dirette a qualunque indirizzo (uno stesso host può avere più indirizzi IP)

```
struct sockaddr_in mio_indirizzo;

mio_indirizzo.sin_family      = AF_INET;
mio_indirizzo.sin_port        = htons(5200);
mio_indirizzo.sin_addr.s_addr  = htonl(INADDR_ANY);

bind(fd, (struct sockaddr *) &mio_indirizzo, sizeof(mio_indirizzo));
```

Indirizzi TCP/IP per il client (connect)

- Il client deve conoscere l'indirizzo IP e la porta del processo server

```
struct sockaddr_in indirizzo;

indirizzo.sin_family = AF_INET;
indirizzo.sin_port   = htons(5200);
inet_aton("143.225.5.3", &indirizzo.sin_addr);

connect(fd, (struct sockaddr *) &indirizzo, sizeof(indirizzo));
```

Mettersi in ascolto

int **listen**(int **sockfd**, int **lunghezza_coda**);

- Mette il socket in modalita' di ascolto
  - in attesa di nuove connessioni
- Solo per socket SOCK_STREAM e SOCK_SEQPACKET
- Il secondo argomento specifica quante connessioni possono essere in attesa di essere accettate
  - Se il numero di connessioni in attesa supera il secondo parametro, il client riceve “connection refused”
- Restituisce 0 oppure -1

Accettare una nuova connessione

int **accept**(int **sockfd**,
struct sockaddr ***indirizzo_client**,
socklen_t ***dimensione_indirizzo**);

- Il secondo e terzo argomento servono ad identificare il client
  - possono essere NULL
- Restituisce un nuovo descrittore! (oppure -1)
  - crea un nuovo socket, dedicato a questa nuova connessione
  - il vecchio socket resta in ascolto
- Blocca il processo se non vi sono connessioni in attesa
  - Il socket puo' essere marcato non-blocking.

Struttura di un server

```
int fd1, fd2;
struct sockaddr_in mio_indirizzo;

mio_indirizzo.sin_family      = AF_INET;
mio_indirizzo.sin_port        = htons(5200);
mio_indirizzo.sin_addr.s_addr  = htonl(INADDR_ANY);



fd1 = socket(PF_PF_INET, SOCK_STREAM, 0);
bind(fd1, (struct sockaddr *) &mio_indirizzo, sizeof(mio_indirizzo));

listen(fd1, 5);
fd2 = accept(fd1, NULL, NULL);
...
close(fd2);
close(fd1);
```

Esempio

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

void main() {
    int ds_sock, ds_sock_acc;
    struct sockaddr_in my_addr;
    struct sockaddr addr;
    int addrlen;

    ds_sock = socket(AF_INET, SOCK_STREAM, 0);

    my_addr.sin_family      = AF_INET;
    my_addr.sin_port        = 1999;
    my_addr.sin_addr.s_addr = INADDR_ANY;

    bind(ds_sock, &my_addr, sizeof(my_addr));
    ds_sock_acc = accept(ds_sock, &addr, &addrlen);

    close(ds_sock);
    close(ds_sock_acc);
}
```

Esempio

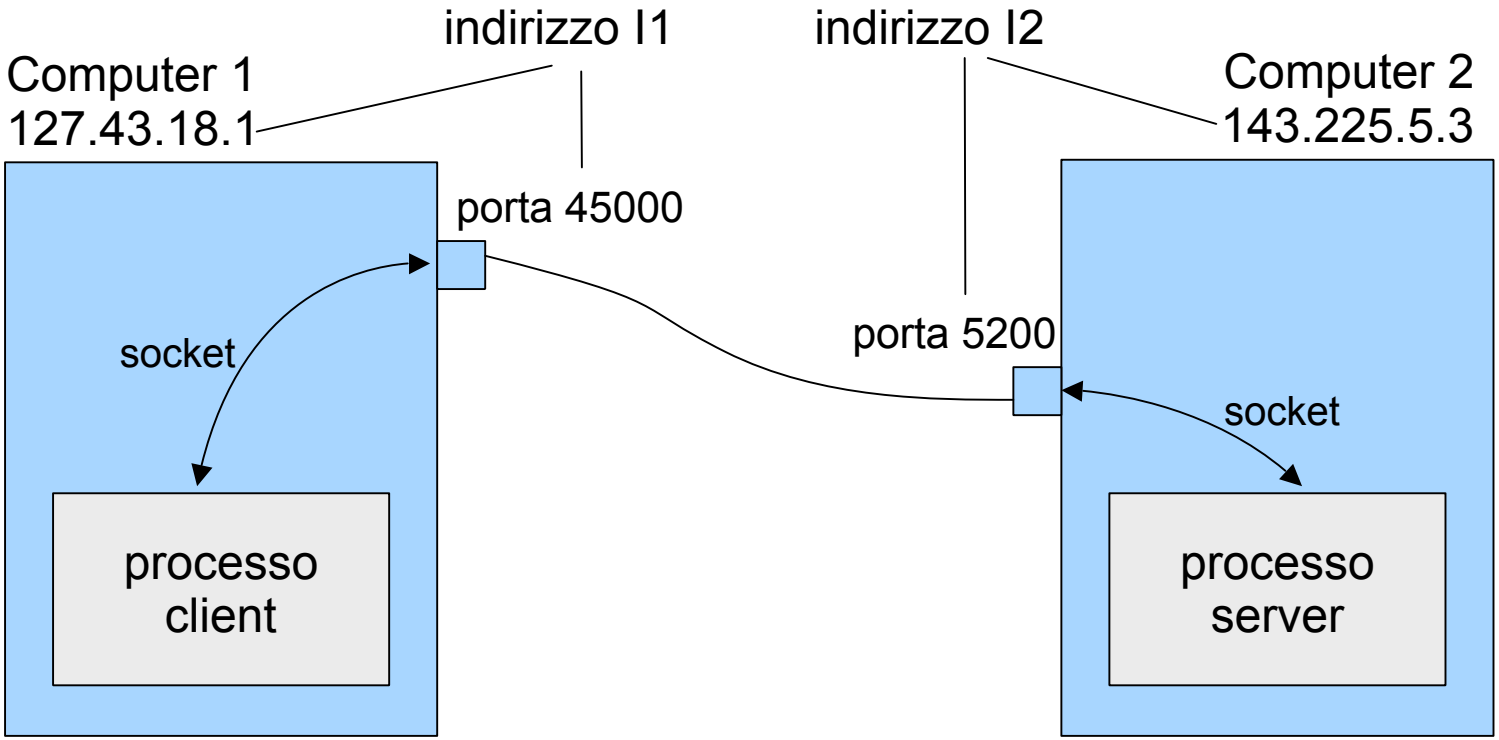
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

void main() {
    int ds_sock, length, ret;

    struct sockaddr_in addr;
    struct hostent *hp; /* utilizzato per la restituzione della chiamata */
                        /* gethostbyname() commentata nel testo          */

    ds_sock = socket(AF_INET, SOCK_STREAM, 0);

    addr.sin_family = AF_INET;
    addr.sin_port   = 1999;
    hp = gethostbyname("conde.dis.uniroma1.it");
    memcpy(&addr.sin_addr, hp->h_addr, 4);

    ret = connect(ds_sock, &addr, sizeof(addr));
    if ( ret == -1 ) Errore_("Errore nella chiamata connect");

    close(ds_sock);
}
```

```
#define h_addr h_addr_list[0]; /* indirizzo del buffer di specifica del numero IP */
struct hostent {
    char   *h_name;            /* nome ufficiale dell'host           */
    char   **h_aliases;        /* lista degli alias                  */
    int    h_addrtype;         /* tipo di indirizzo dell'host        */
    int    h_length;           /* lunghezza (in byte) dell'indirizzo */
    char   **h_addr_list;      /* lista di indirizzi dal name server */
}
```

Struttura di un client

```
int fd;
struct sockaddr_in mio_indirizzo;

mio_indirizzo.sin_family      = AF_INET;
mio_indirizzo.sin_port        = htons(5200);
inet_aton("143.225.5.3", &indirizzo.sin_addr);

fd = socket(PF_INET, SOCK_STREAM, 0);
connect(fd, (struct sockaddr *) &mio_indirizzo,
sizeof(mio_indirizzo));
...
close(fd);
```

Schema della connessione



socket(...)
connect(I2)

socket(...)
bind(I2)
listen(...)
accept(...)

Trasmissione Dati

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

- invia il contenuto del buffer `buf` al socket specificato
- si usa esclusivamente con `SOCK_STREAM`
- restituisce il numero di byte inviati oppure -1 in caso di errore
- è la stessa funzione che consente la scrittura su file

Ricezione

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

- solo per socket connessi (SOCK_STREAM)
- legge un messaggio di lunghezza massima len dal socket
- se non c'è alcun messaggio, il programma rimane sospeso
  - chiamata *bloccante*
- la funzione ritorna il numero di byte letti, -1 in caso di errore
- è la stessa funzione che consente la lettura da un file

Leggere scrivere su socket

Stessa interfaccia per scrittura su file:

- Si possono passare le sd a processi figli che possono non distingure tra sd e fd

- Possono essere usati anche con processi che lavorano su file locali.

-Esistono altre funzioni specifiche per leggere e scrivere su socket: 3 modalità di send, 3 modalità di recieve.

Leggere e Scrivere su socket

-**Send & Recv:**

-**send** è come write (richiede connessione stabilita), ma ha flag per specificare modalità di scrittura,

-**sendto** permette la scrittura su connectionless socket,

-**sendmsg** per specificare buffer multipli

- **recv** come read con flag,

-**recvfrom** per ottenere indirizzo della fonte,

-**rcvmsg** per ricevere da buffer multipli

send() e sendto()

```
#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);
```

man 2 send

- `send()` può essere utilizzata solo se `s` è stato connesso; `sendto()` sempre perché richiede di specificare l'indirizzo di destinazione
- restituisce `-1` in caso di errore oppure il numero di byte effettivamente trasmessi
- `flags` si può lasciare a zero

sendTo()

man 2 sendto

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int socket, const void *msg, int len,
                unsigned int flags,
           const struct sockaddr *to, int tolen);
```

- `sendto` può essere utilizzata sempre perché richiede di specificare l'indirizzo di destinazione
  - vedremo che esiste una variante `send` che può essere utilizzata solo se `socket` è stata connessa;
- restituisce `-1` in caso di errore oppure il numero di byte effettivamente trasmessi
- `flags` si può lasciare a zero

recv e recvfrom()

```
#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);
```

man 2 recv

- ricevono in `buf` non più di `len` byte. Se `from` non è `NULL`, la struttura `sockaddr` verrà riempita con l'indirizzo del mittente
- restituisce `-1` in caso di errore oppure il numero di byte effettivamente ricevuti
- `flags` si può lasciare a zero

recvfrom()

```
#include <sys/types.h>
#include <sys/socket.h>
int recvfrom(int s, void *buf, int len,
             unsigned int flags,
             struct sockaddr *from,
             int *fromlen);
```

man 2 recv

- ricevono in `buf` non più di `len` byte. Se `from` non è `NULL`, la struttura `sockaddr` verrà riempita con l'indirizzo del mittente
- restituisce `-1` in caso di errore oppure il numero di byte effettivamente ricevuti
- se non arriva alcun messaggio, il programma rimane sospeso (la chiamata è "bloccante")
- `flags` si può lasciare a zero

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/signal.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define MAX_DIM 1024
#define MAX_CODA 3

void main() {
    int ds_sock, ds_sock_a, rval;
    struct sockaddr_in server;
    struct sockaddr client;
    char buff[MAX_DIM];

    sock = socket(AF_INET, SOCK_STREAM, 0);

    server.sin_family      = AF_INET;
    server.sin_port        = 1999;
    server.sin_addr.s_addr = INADDR_ANY;

    bind(ds_sock, &server, sizeof(server));
    listen(ds_sock, MAX_CODA);

    length = sizeof(client);
    signal(SIGCHLD, SIG_IGN);

    while(1) {

        while( (ds_sock_a = accept(ds_sock, &client, &length)) == -1);

        if (fork()==0) {
            do {
                read(ds_sock, buff, MAX_DIM);
                printf("messaggio del client = %s\n", buff);
            } while(strcmp(buff,"quit") != 0);
            write(ds_sock, "fatto", 10);
            close(ds_sock_a);
            exit(0);
        }
        else close(ds_sock_a);
    }
}
```

Figure 9.8: Il server che gestisce acquisizione di stringhe

Esempio

Una semplice applicazione client-server che opera nel dominio PF_UNIX. Il server riceve un carattere dal client, restituendo il carattere successivo (secondo la codifica ASCII).

Client:

C1) Inclusione degli header necessari e dichiarazione delle variabili

```
#include <sys/type.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

int main()
{
    int sockfd;
    int len;
    struct sockaddr_un address; /* UNIX domain addresses */
    int result;
    char ch = 'A';
```

Esempio

C2) Creazione del socket dal lato client: il socket non abbisogna di un nome, quindi alla socket non viene fatta seguire una bind. L'argomento *protocol* può essere posto a 0 in quanto i due precedenti individuano univocamente il protocollo

```
sockfd = socket(PF_UNIX, SOCK_STREAM, 0);
```

C3) Connessione del socket locale al socket del server: si presuppone che il client conosca il nome del server-socket. Si noti il casting alla struttura di indirizzi generica nella chiamata connect

```
address.sun_family = AF_UNIX;
strcpy(address.sun_path, "server_socket");
len = sizeof(address);

result = connect(sockfd, (struct sockaddr *)&address, len);
if(result == -1) {
    perror("client");
    exit(1);
}
```

Esempio

C4) trasmissione del carattere e ricezione della risposta dal server

```
    write(sockfd, &ch, 1);
    read(sockfd, &ch, 1);
    printf("char from server = %c\n", ch);
    close(sockfd);
    exit(0);
}
```

Esempio

Server:

S1) Inclusione degli header e dichiarazione delle variabili

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

int main()
{
   int s_sfd, c_sfd;
   int s_len, c_len;
   struct sockaddr_un s_addr;
   struct sockaddr_un c_addr;
   static void myh(int); /* a signal handler */
```

S2) Gestione di una interruzione e di un kill

```
   if(signal(SIGINT,myh)==SIG_ERR |signal(SIGKILL, myh)==SIG_ERR)
```

Esempio

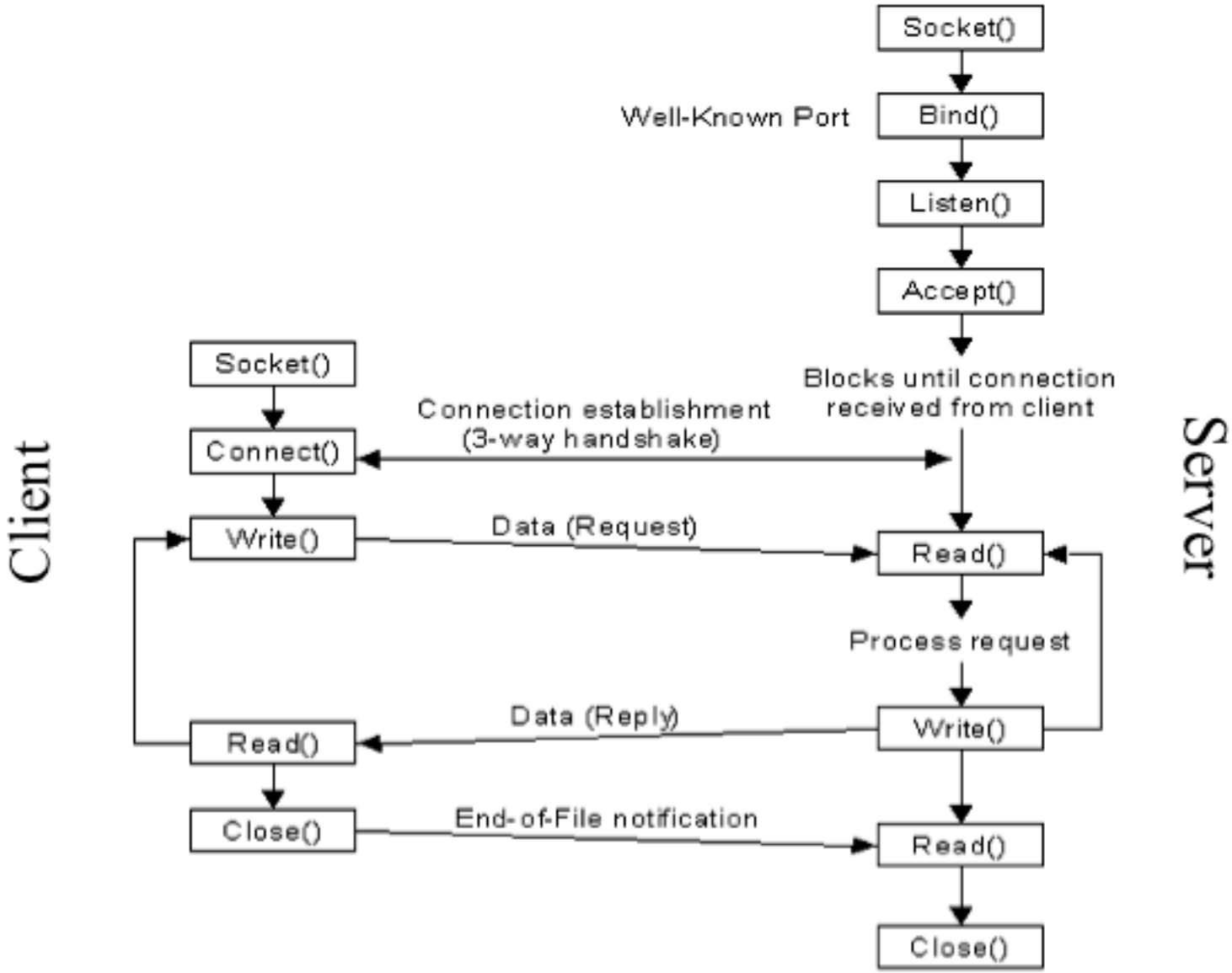
S6) Lettura del dato dal client e relativa risposta

```
    read(c_sfd, &ch, 1);
    ch++;
    write(c_sfd, &ch, 1);
    close(c_sfd);
  } /* end while */
}
```

S7) Gestore dei segnali di terminazione

```
static void myh(int signum)
{
   printf("cleaning socket file...");
   unlink("server_socket");
   return;
}
```

Comunicazione TCP/IP



Apertura Comunicazione TCP/IP

1) Il server si predispone ad accettare le richieste di connessione, mediante le chiamate a socket, bind, listen e infine accept che realizza una apertura passiva (passive open) cioè senza trasmissione di dati.

2) Il client effettua le chiamate a socket, bind ed infine alla connect che realizza una apertura attiva (active open) mediante la spedizione di un segmento TCP detto SYN segment (synchronize) in cui è settato ad 1 il flag syn, a zero il flag ack, e che trasporta un numero di sequenza iniziale (J) che è il numero di sequenza iniziale dei dati che il client vuole mandare al server. Il segmento contiene un header TCP con i numeri di porta ed eventuali opzioni su cui accordarsi, e di solito non contiene dati. Il segmento viene incapsulato in un datagram IP.

Apertura Comunicazione TCP/IP

3) Il server deve rispondere al segmento SYN del client spedendogli un segmento SYN (flag syn settato ad 1) con il numero di sequenza iniziale (K) dei dati che il server vuole mandare al client in quella connessione. Il segmento presenta inoltre nel campo Ack number il valore J+1 che indica che si aspetta di ricevere J+1, e presenta il flag ack settato ad 1, per validare il campo Ack number.

4) Il client, ricevendo il SYN del server con l'Ack numer J+1 sa che la sua richiesta di connessione è stata accettata, e dal sequence number ricevuto K capisce che i dati del server inizieranno da K+1, quindi risponde con un segmento ACK (flag syn settato a zero e flag ack settato a 1) con Ack number K+1, e termina la connect.

5) al ricevimento dell'ACK K+1 il server termina la accept.

Chiusura Comunicazione

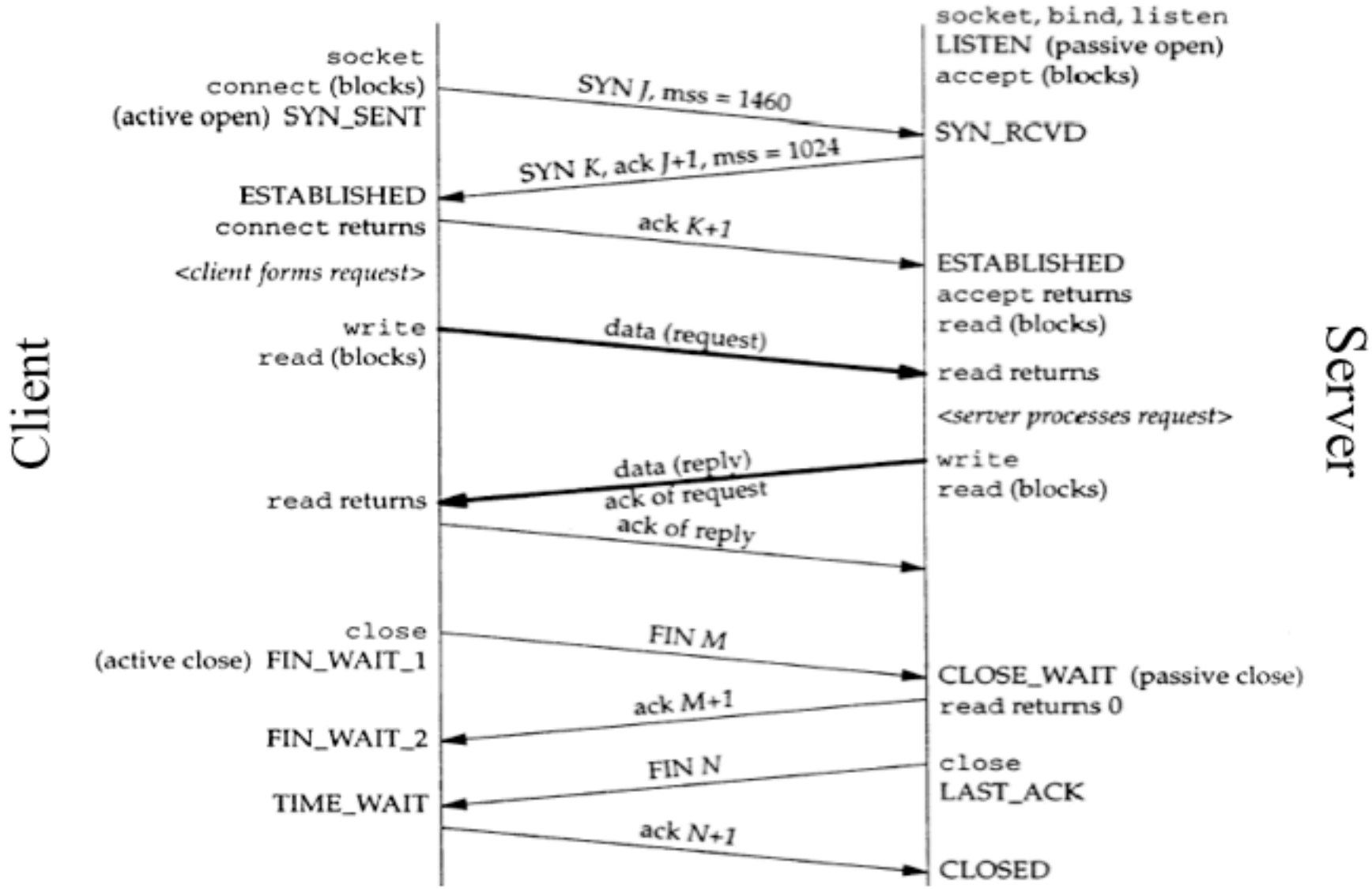
1) una delle applicazioni su un end-system effettua la chiusura attiva (active close) chiamando la funzione close che spedisce un segmento FIN (flag FIN settato a 1), con un numero di sequenza M pari all'ultimo dei byte trasmessi in precedenza più uno. Con ciò si indica che viene trasmesso un ulteriore dato di 1 byte, che è il FIN stesso.

2) l'end system che riceve il FIN effettua la chiusura passiva (passive close) all'insaputa dell'applicazione.

Per prima cosa il modulo TCP del passivo spedisce all'end-system attivo un segmento ACK con Ack number pari a M+1, come riscontro per il FIN ricevuto.

Poi il TCP passivo trasmette all'applicazione padrona di quella connessione il segnale FIN, sotto forma di end-of-file che viene accodato ai dati non ancora letti dall'applicazione. Poichè la ricezione del FIN significa che non si riceverà nessun altro dato, con l'end-of-file il TCP comunica all'applicazione che lo stream di input è chiuso.

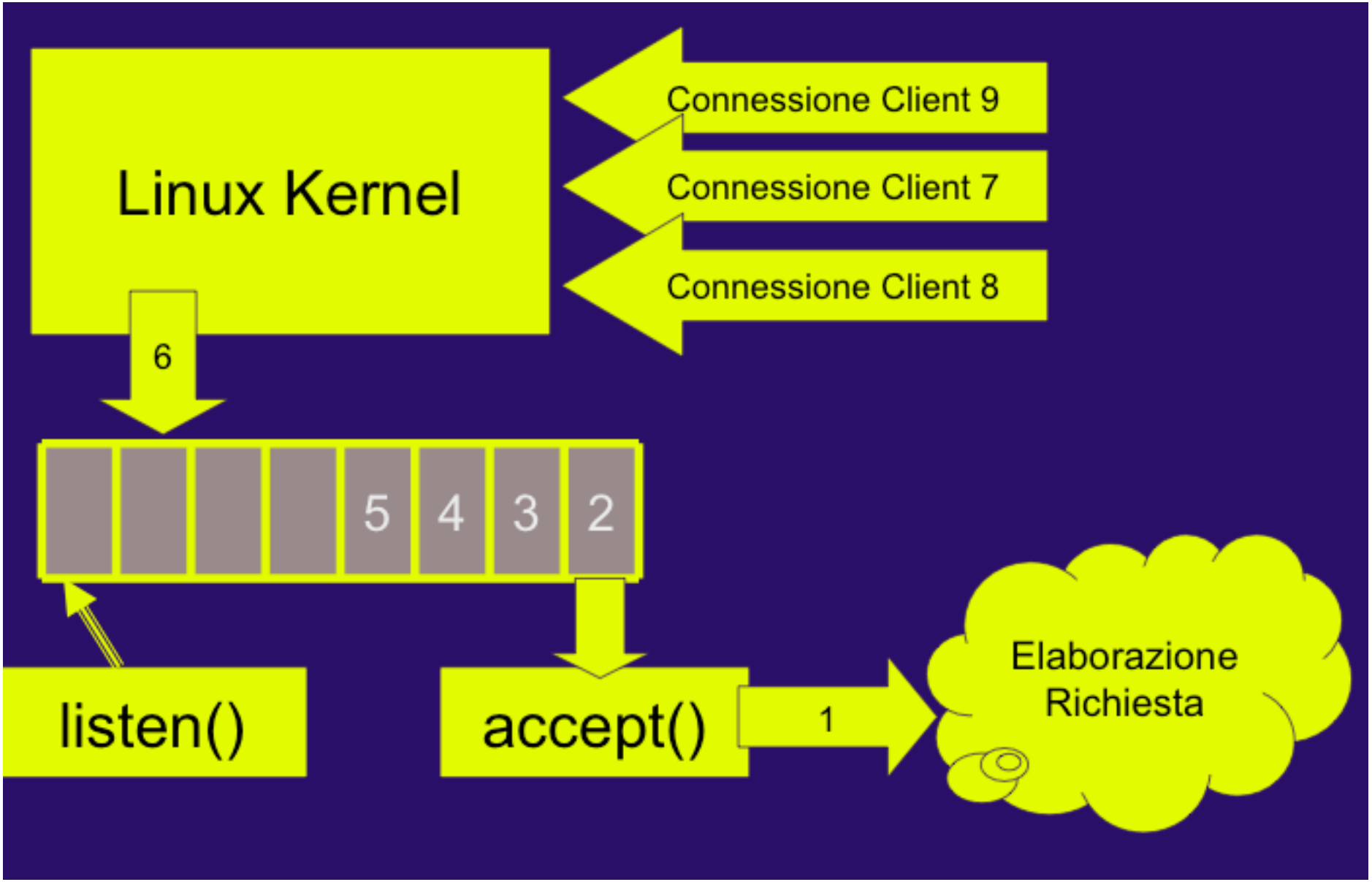
Comunicazione TCP/IP



Server a Programmazione Concorrente

- Un generico server attende le richieste di connessione su una determinata porta
- Possono arrivare richieste concorrenti e da molteplici client
- Una soluzione senza programmazione concorrente:
  - client serviti uno alla volta, finché una connessione non è terminata, non vengono serviti altri client interessati al servizio
  - N.B. comunque è il SO che gestisce le richieste concorrenti ed in effetti le serializza

La Coda di Connessione



Server a Programmazione Concorrente

- Un server che gestisce sequenzialmente i vari client è insoddisfacente
  - il tempo di attesa di ogni client
    - potrebbe risultare eccessivo
    - dipende dalla durata delle altri connessioni
  - se le connessioni durano molto, il server ben presto diverrebbe indisponibile anche solo ad accodare le nuove richieste di connessioni
- Nei server reali le richieste di vari client devono essere gestite concorrentemente

Server a programmazione Concorrente

- se il server è concorrente, per ogni client viene generato un processo ad hoc per offrire il servizio, in modo che più client possano essere serviti in modalità concorrente. In tal caso l'attesa sulla coda è limitata al tempo necessario alla gestione della richiesta del client da parte del server ed allo start-up del nuovo processo;

Schema di un server concorrente: processi

```
socket(...);
bind(...);
listen(...);

while (1) {
        fd2 = accept(fd1, (struct sockaddr *)NULL, NULL);

        if ( (pid = fork()) < 0 ) {
                perror("fork");
                exit(1);
        } else if (pid == 0) {      // processo figlio
                close(fd1); // al figlio non serve fd1
                // gestisce la connessione usando fd2
                ...
                exit(0);    // poi il figlio termina
        }
        // il processo padre chiude fd2 e ripete il ciclo
        close(fd2);
}
```

Esercizio: Server

```
#define SOCKET_NAME "/tmp/my_first_socket"

static int gestisci(int);

int main(int argc, char **argv)
{
  int listen_sd, connect_sd; // Socket descriptor

  struct sockaddr_un my_addr, client_addr;
  socklen_t client_len;

  my_addr.sun_family = AF_LOCAL;
  strcpy(my_addr.sun_path, SOCKET_NAME);

  // Server section: create a local socket
  if ( (listen_sd = socket(PF_LOCAL, SOCK_STREAM, 0)) < 0)
    perror("socket"), exit(1);
```

Esercizio: Server

```
// remove socket file if present
  unlink(SOCKET_NAME);

// bind socket to pathname
  if ( bind(listen_sd, (struct sockaddr *) &my_addr, sizeof(my_addr)) < 0)
    perror("bind"), exit(1);

  // put socket in listen state
  if ( listen(listen_sd, 1) < 0)
    perror("listen"), exit(1);

  while (1) {
    client_len = sizeof(client_addr);
     fprintf(stderr, " sizeof(client_addr)=%d \n", client_len);

    if ( (connect_sd = accept(listen_sd, (struct sockaddr *) &client_addr, &client_len)) < 0)
      perror("accept"), exit(1);
```

```
if ( (pid = fork()) < 0 ) {
            perror("fork");
            exit(1);
      } else if (pid == 0) {       // processo figlio
            close(listen_sd); // al figlio non serve listen_sd

           // gestisce la connessione usando fd2
            fprintf(stderr, " new connection \n");
            fprintf(stderr, " client address: %100s\n ", client_addr.sun_path);

            // handle the connection
            gestisci(connect_sd);
            exit(0);    // poi il figlio termina
      }
    close(connect_sd);
  }

  return 0;
}
```

Esercizio: client

```
#define SOCKET_NAME "/tmp/my_first_socket"

static int client(void);

int main(int argc, char **argv)
{
  return client();
}

// Client section: Sends integers from 0 to 4, at 1 second intervals, and then terminates
int client(void)
{
  char msg[100];
  struct sockaddr_un srv_addr;
  int sd, i;
  ssize_t temp; /* signed size_t */

  signal(SIGPIPE, SIG_IGN);

  // crea il socket
  sd = socket(PF_LOCAL, SOCK_STREAM, 0);

  if (sd < 0) perror("socket"), exit(1);

  srv_addr.sun_family = AF_LOCAL; // unsigned short int
  strcpy(srv_addr.sun_path, SOCKET_NAME);
```

Esercizio

```
 // si connette all'indirizzo predefinito
  if ( connect(sd,(struct sockaddr *) &srv_addr, sizeof(srv_addr) ) < 0)
    perror("connect"), exit(1);

  sleep(1);
  read(sd, msg, 26);
  printf(" client riceve: *%s*\n", msg);
  printf(" client riceve: %d\n", strlen(msg));
  return 0;

}
```

Schema di un server concorrente multithreaded

```
socket(...);
bind(...);
listen(...);

while (1) {
  client_len = sizeof(client_addr);
  connect_sd = accept(listen_sd,
                          (struct sockaddr *) &client_addr,
                          &client_len)

   thread_sd = (int *) malloc(sizeof(int));

   *thread_sd =  connect_sd;
   printf("server: new connection from %d \n",connect_sd);
   pthread_create(&tid, NULL, gestisci, (void *) thread_sd);

}
```

Esercizio: Server

```
#define SOCKET_NAME "/tmp/my_first_socket"

static int gestisci(int);

int main(int argc, char **argv)
{
  int listen_sd, connect_sd; // Socket descriptor

  struct sockaddr_un my_addr, client_addr;
  socklen_t client_len;

  my_addr.sun_family = AF_LOCAL;
  strcpy(my_addr.sun_path, SOCKET_NAME);

  // Server section: create a local socket
  if ( (listen_sd = socket(PF_LOCAL, SOCK_STREAM, 0)) < 0)
    perror("socket"), exit(1);
```

Esercizio: Server

```
// remove socket file if present
  unlink(SOCKET_NAME);

// bind socket to pathname
  if ( bind(listen_sd, (struct sockaddr *) &my_addr, sizeof(my_addr)) < 0)
    perror("bind"), exit(1);

  // put socket in listen state
  if ( listen(listen_sd, 1) < 0)
    perror("listen"), exit(1);

  while (1) {
    client_len = sizeof(client_addr);
     fprintf(stderr, " sizeof(client_addr)=%d \n", client_len);

    if ( (connect_sd = accept(listen_sd, (struct sockaddr *) &client_addr, &client_len)) < 0)
      perror("accept"), exit(1);
```

Esercizio: Server

```
thread_sd = (int *) malloc(sizeof(int)); // serve nuova mem!

fprintf(stderr, " new connection \n");

fprintf(stderr, " client address: %100s\n ",
client_addr.sun_path);

// handle the connection
//gestisci(connect_sd);
*thread_sd =  connect_sd;
printf("server: new connection from %d \n",connect_sd);

pthread_create(&tid, NULL, gestisci, (void *) thread_sd);

 }
```

Esercizio: Server

```
void * gestisci(void * arg){
  char buf[100];
  int n, sd;

  sd = *((int *) arg;
  printf("server: gestisci sd = %d \n",*((int *) arg));
  time_t ora;
  time(&ora);
  printf(" Ora: %s\n", ctime_r(&ora, buf));
  printf(" Ora: %d\n", strlen(buf));
  write(sd, buf, 26);
  close(sd);
  free(arg);
}
```

Opzioni Socket

```
int getsockopt(int fd, level, option, void *val, socklen_t len);
int setsockopt(int fd, level, option, void *val, socklen_t *len);
```

- leggono e scrivono le opzioni di un socket
- le opzioni si dividono in *livelli*, identificati da apposite costanti
- livelli:

livello socket	SOL_SOCKET
livello TCP	IPPROTO_TCP
livello IP	IPPROTO_IP

- il significato di val dipende dall'opzione
- len è pari alla dimensione dell'oggetto puntato da val
- restituiscono: 0 se OK, -1 altrimenti (e impostano errno)

Opzione SO_REUSEADDR

```
int getsockopt(int fd, level, option, void *val, socklen_t len);
int setsockopt(int fd, level, option, void *val, socklen_t *len);
```

- opzione di livello socket
- permette a bind di assegnare un indirizzo ancora occupato
- val deve puntare ad un intero
  - se l'intero è diverso da zero, questa opzione è attiva
  - se l'intero puntato contiene zero, l'opzione è inattiva
  - len è pari a sizeof(int)

Opzioni SO_SNDTIMEO, SO_RCVTIMEO

```
int getsockopt(int fd, level, option, void *val, socklen_t len);
int setsockopt(int fd, level, option, void *val, socklen_t *len);
```

- opzioni di livello socket
- impostano un timeout per le operazioni di lettura/scrittura
- terminato il timeout, le operazioni di lettura/scrittura vengono interrotte con valore di ritorno negativo (errore) e errno == EWOULDBLOCK

Opzioni SO_SNDTIMEO, SO_RCVTIMEO

int **getsockopt**(int **fd**, **level**, **option**, void *val, socklen_t **len**);
int **setsockopt**(int **fd**, **level**, **option**, void *val, socklen_t ***len**);

- val deve puntare ad una struttura timeval

```
struct timeval {
    time_t tv_sec;  /* seconds      */
    long   tv_usec; /* microseconds */
}
```

- len è pari a sizeof(timeval)
- i timeout si annullano impostando un nuovo timeout di zero secondi e zero microsecondi

Comandi utili: Netstat

- netstat [-t] [-all] [-p] [-n]
  - elenca tutti i socket di rete del sistema (non riguarda i socket locali)
  - “-t” mostra solo i socket TCP, cioè quelli con famiglia=PF_INET e tipo=SOCK_STREAM
  - “-all” (oppure “-a”) mostra anche i socket in ascolto
  - “-p” specifica il pid del processo che ha creato ciascun socket
  - “-n” mostra gli indirizzi e le porte in formato numerico (invece di simbolico)
  - ad es., netstat -t -a -p -n mostra tutti i socket TCP aperti, indicando il PID del processo corrispondente e mostrando gli indirizzi in formato numerico

Altri comandi utili

- /sbin/ifconfig
  - mostra l'indirizzo IP della macchina corrente
- nslookup <nome di dominio>
  - fornisce l'indirizzo IP di un host del quale conosciamo il nome
  - esempio: “nslookup www.unina.it”

Nomi di Dominio

- Ad un host può venir assegnato un nome di dominio (*domain name*), come www.unina.it
- Il servizio di rete chiamato DNS (*Domain Name Service*) converte i nomi di dominio in indirizzi IP
  - www.unina.it → 143.225.5.3
  - questa conversione prende il nome di “risoluzione del nome”, dall'inglese “domain name resolution”
- Dalla shell, il comando nslookup esegue la conversione
  - nslookup <nome>
  - esempio: nslookup www.unina.it

Nomi di Dominio

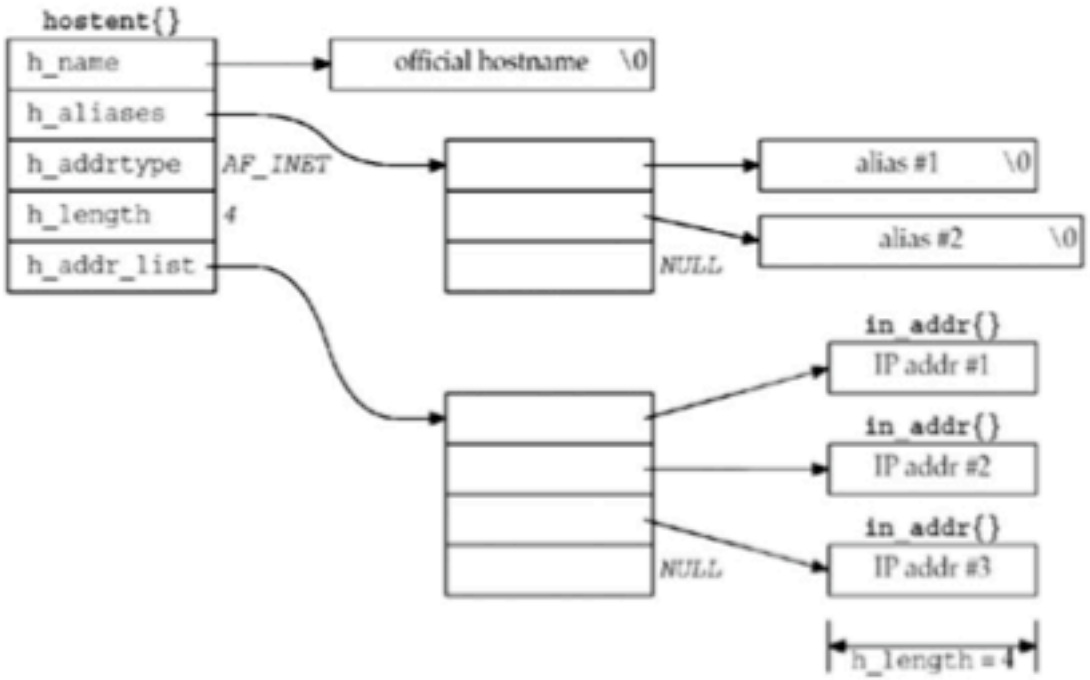
```
struct hostent *gethostbyname(const char *nome);

struct hostent {
  char    *h_name;          nome canonico dell'host
  char    **h_aliases;      lista di alias
  int     h_addrtype;       famiglia dell'indirizzo (AF_INET)
  int     h_length;         lunghezza dell'indirizzo (4)
  char    **h_addr_list;    lista di indirizzi
}
```

- Restituisce l'indirizzo IP (oltre ad altre informazioni) corrispondente al nome di dominio dato
- L'indirizzo si trova in h_addr_list[0], già in network order
- Un nome può corrispondere a più indirizzi h_addr_list[0], h_addr_list[1], ...

Nomi Dominio

- Struttura hostent

[Stevens]




```
struct sockaddr_in indirizzo;

struct hostent *p = gethostbyname("www.unina.it");
if (!p) herror("gethostbyname"), exit(1);
indirizzo.sin_addr.s_addr = *(uint32_t *)(p->h_addr_list[0]);
```

Stampare nome dominio

- Partiamo da un indirizzo IP in network order, come quello ottenuto da accept

```
char *inet_ntoa(struct in_addr in);
```

- Converte l'indirizzo in una stringa in formato dotted
- La stringa viene sovrascritta da ogni nuova chiamata
  - inet_ntoa non è thread-safe

Comadi utili: Netstat

indirizzo (INADDR_ANY)

porta (111)

netstat -t -a -p

```
Proto Recv-Q Send-Q Local Address     Foreign Address     State       PID/Program name
tcp        0      0 *:sunrpc          *:*                 LISTEN      6519/portmap
tcp        0      0 *:ipp             *:*                 LISTEN      6677/cupsd
tcp        0      0 localhost:smtp    *:*                 LISTEN      6612/master
tcp        0      0 *:ssh             *:*                 LISTEN      6873/sshd
tcp        0      0 pcfaella.na:44619 alfa.na.infn.it:ssh ESTABLISHED 17378/ssh
```

porta effimera

client ssh

server ssh

Comandi utili: telnet

- telnet <indirizzo> <porta>
  - crea un socket e lo collega all'indirizzo remoto dato
  - esempio: “telnet www.unina.it 80” si mette in comunicazione con il server http dell'università
  - quello che si digita da terminale viene mandato sul socket
  - quello che proviene dal socket viene stampato su stdout
  - telnet può quindi fungere da “client generico”

Risoluzione Indirizzi

- `gethostbyname()`
  - dato un hostname, restituisce una struttura dati che specifica anche i suoi indirizzi IP
- `gethostbyaddr()`
  - dato un indirizzo IP, restituisce una struttura dati che specifica anche il suo hostname
- `getservbyname()`
  - dato un nome di servizio e di protocollo, restituisce una struttura dati che specifica i suoi nomi e l'indirizzo di porta
- `gethostname()/getdomainname()`
  - restituiscono l'hostname della macchina
- `herror()`
  - stampa un messaggio di errore per `gethostname()`

Risoluzione Indirizzi

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
struct hostent {
  char *h_name;      // nome ufficiale dell'host
  char **h_aliases; // array NULL-terminated di alias
  int  h_addrtype;  // Tipo di ind. da restituire; spesso AF_INET
  int  h_length;    // lunghezza in byte dell'indirizzo
  char **h_addr_list;// un NULL-terminated array di ind. …
                    // … che sono in Network Byte Order
};

#define h_addr h_addr_list[0] // primo ind. di h_addr_list
```

- h_addr contiene l'indirizzo IP dell'host
- h_length contiene la lunghezza dell'indirizzo

Lookup.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
…
extern int h_errno;

int main(int argc,char **argv) {
  int x, x2;
  struct hostent *hp;

  for (x=1; x<argc; ++x) { /* Look up the host name : */
    hp = gethostbyname(argv[x]);
    if (!hp) { /* Report lookup failure */
      fprintf(stderr,   "%s: host '%s'\n",
                        hstrerror(h_errno),
                        argv[x]);
      continue;
    }
…
```

Lookup.c

```
... /* Report the findings :   */
    printf("Host %s :\n",argv[x]);
    printf("  Officially:\t%s\n", hp->h_name);
    fputs("  Aliases:\t",stdout);
    for(x2=0; hp->h_aliases[x2]; ++x2 ) {
      if (x2) fputs(", ",stdout);
      fputs(hp->h_aliases[x2],stdout);
    }
    fputc('\n',stdout);
    if (hp->h_addrtype==AF_INET) {
      printf("  Type:\t\tAF_INET\n");
      if ( hp->h_addrtype == AF_INET ) {
        for (x2=0; hp->h_addr_list[x2]; ++x2 )
          printf("  Address:\t%s\n",
            inet_ntoa(*(struct in_addr *)hp->h_addr_list[x2]));
      }
      putchar('\n');
    }
  }
```

Lookup Output

```
$ ./lookup www.yahoo.com www.redhat.com www.dia.uniroma3.it

Host www.yahoo.com :
  Officially:   www.yahoo.akadns.net
  Aliases:      www.yahoo.com
  Type:         AF_INET
  Address:      64.58.76.222
  Address:      64.58.76.223
  Address:      64.58.76.224
  Address:      64.58.76.225
  [...omissis...]
Host www.redhat.com :
  Officially:   www.redhat.com
  Aliases:
  Type:         AF_INET
  Address:      66.187.232.56

Host www.dia.uniroma3.it :
  Officially:   mail.dia.uniroma3.it
  Aliases:      www.dia.uniroma3.it
  Type:         AF_INET
```