I Socket

Reti di Elaboratori

Matteo Sereno

Interfacce tra programmi e protocolli

Il software relativo ai protocolli TCP/IP risiede nel sistema operativo

TCP/IP è progettato per operare su diversi tipi di macchine (indipendenza dalle rappresentazioni interne di una certa macchina)

Gli standard TCP/IP non specificano i dettagli di come le applicazioni si interfacciano con i protocolli

Interfacce tra programmi e protocolli

Questa modo *losely specified* presenta vantaggi e svantaggi

- grande flessibilità
- TCP/IP può utilizzare svariati sistemi operativi
- possibilità di scegliere le interfacce più opportune (per esempio di tipo procedurale o di tipo message passing)
- Questa mancanza di specificazione impone che i progettisti devono fornire i dettagli relativi alla loro implementazione per ogni S.O.
  - nuove interfacce hw --> nuove interfacce sw, le applicazioni diventano poco portatili

Interfacce tra programmi e protocolli

Vantaggi per chi deve progettare sistemi operativi

Svantaggi per i programmatori

- rende le applicazioni meno portatili

In pratica esistono poche varianti di interfacce verso TCP/IP

Interfaccia **socket** (inizialmente proposta nella versione di Unix BSD)

Funzionalità delle Interfacce

allocare risorse necessarie per la comunicazione
specificare gli endpoint della comunicazione (locale e remoto)
iniziare una connessione (lato client)
attendere una connessione (lato server)
inviare e ricevere dati
determinare quando i dati arrivano
generate dati urgent
gestire l'arrivo di dati urgent
terminare una connessione (gracefully)
gestire la terminazione che arriva dalla controparte
abort di una comunicazione
gestire condizioni di errore oppure abor di connessione
rilascio risorse locali quando la comunicazione termina

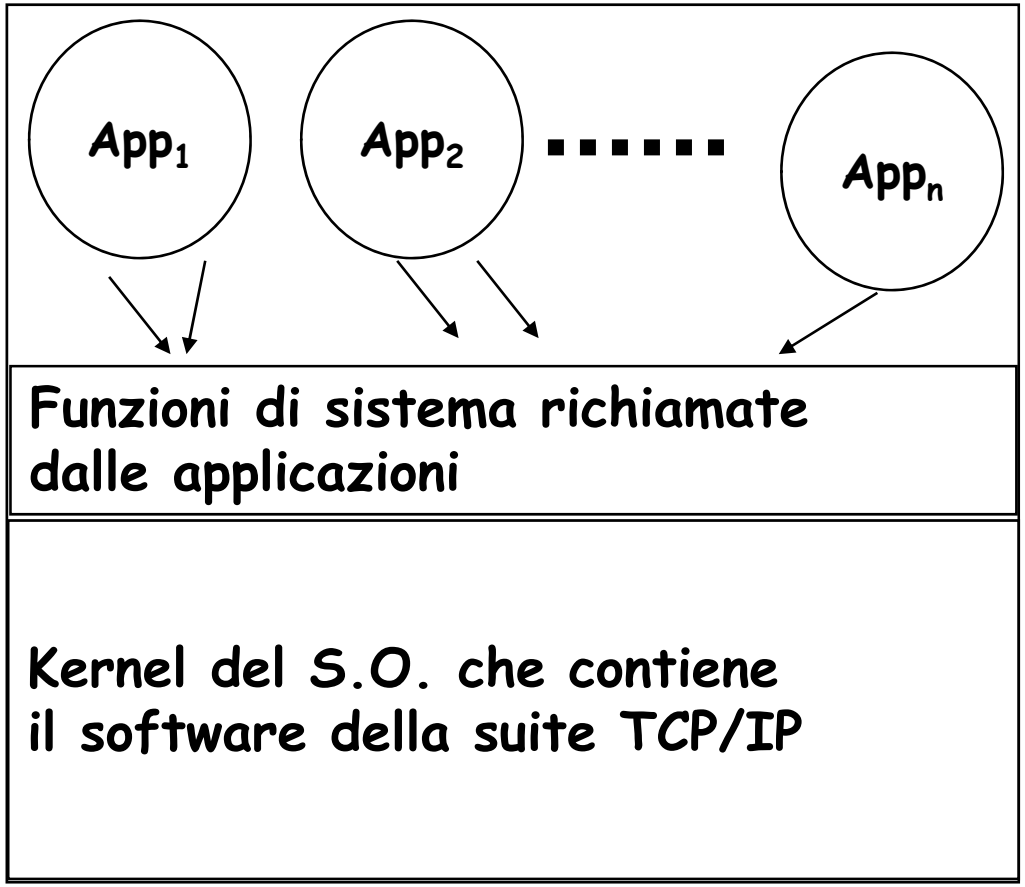
Le System call

I protocolli TCP/IP specificano una sorta di interfaccia concettuale per l'interazione tra protocolli e programmi applicativi

- questa interfaccia viene utilizzata come esemplificazione dell'interazione tra TCP/IP e i programmi applicativi

Le system call rappresentano un meccanismo che moltissimi S.O. usano per trasferire il controllo tra le applicazioni e le procedure del S.O.

Le System call



User address Space

System call interface

Software dei protocolli
System address space

Due possibili approcci

Inventare nuove system call che i programmi possono utilizzare per accedere alle funzionalità offerte da TCP/IP

Utilizzare le system call per I/O per accedere a TCP/IP

Normalmente si cerca di evitare la creazione di nuove system call quando è possibile utilizzare quelle esistenti

Funzioni di base per I/O in Unix

Operazione	Significato
open	Prepara il dispositivo o il file per operazioni di I/O
close	Fine utlizzo del dispositivo (o file) precedentemente aperto
read	Leggere dei dati dal dispositivo (o file) e immetterli nell'area di memoria dell'applicazione
write	Trasferire dei dati dall'area di memoria dell'applicazione al dispositivo (o file)
lseek	Spostamento ad una specifica posizione nel file o dispositivo (si applica a file oppure a dispositivi tipo dischi)
ioct	Controllo di in dispositivo o del software usato per l'accesso (per esempio specifica della dimensione di un buffer

Funzioni di base per I/O in Unix

Esempio

```
int desc;
desc = open(“filename”, O_RDWR, 0);


read(desc, buffer, 128);

close(desc);
```

Specifica di interfacce per i protocolli

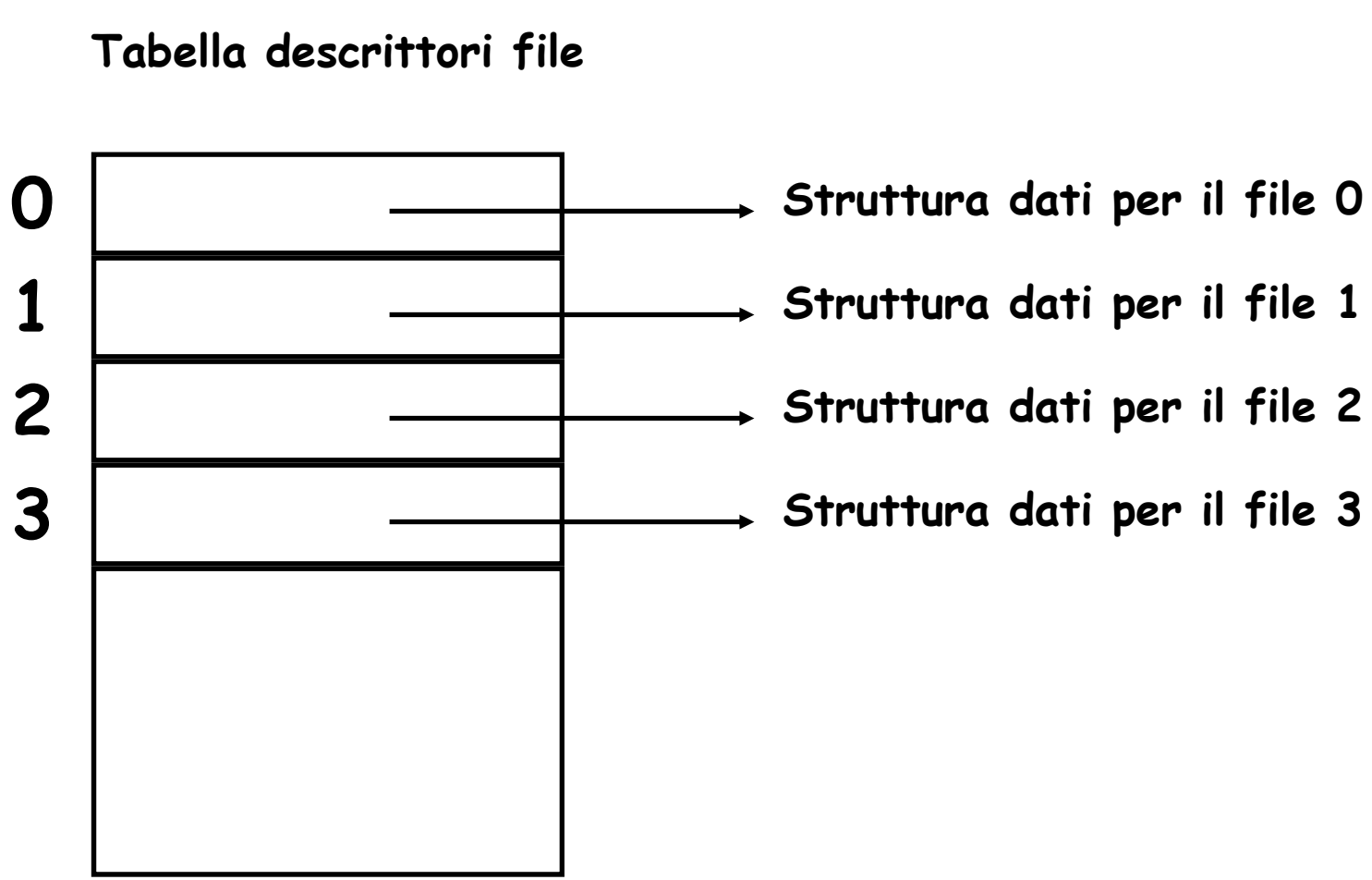
Definire delle funzioni che sono specifiche per la comunicazione mediante TCP/IP

Definire delle funzioni che sono adatte per comunicazioni mediante rete e che mediante opportuni parametri possono supportare TCP/IP

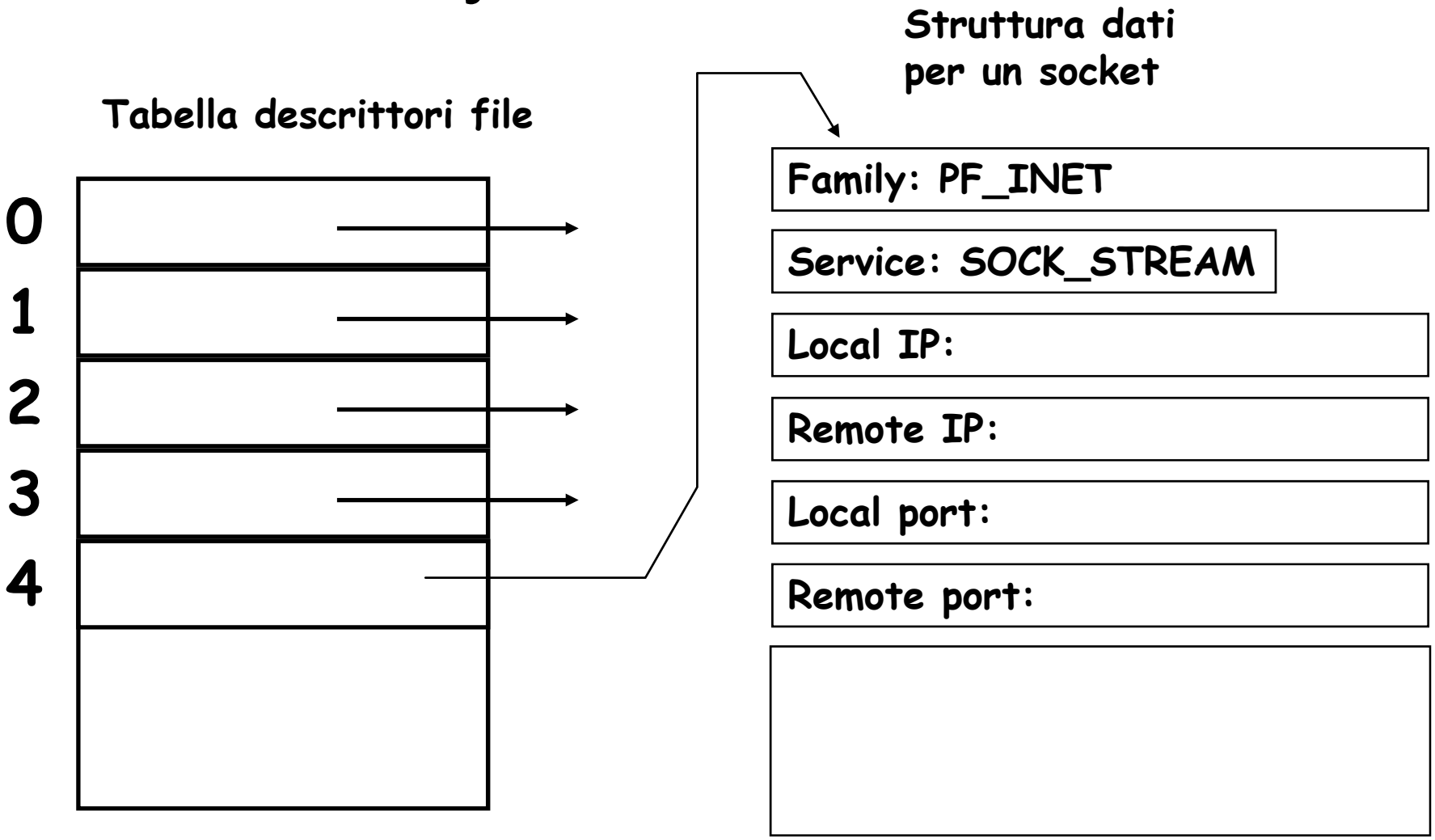
Es: *maketcpconnection* oppure *makeconnection(parametri)*

I socket forniscono delle funzioni per la comunicazione in rete utilizzabili da multe possibili famiglie di protocolli. TCP/IP è una di queste famiglie

Descrittori di socket e di file



System call *socket*



System call *socket*

```
#include <sys/types.h>
#include <sys/socket.h>

int desc;
desc=socket(PF_INET, SOCK_STREAM, 0);
```

socket(protofamily, type, protocol)

protofamily	PF_INET per TCP/IP PF_PUP per i protocolli di Xerox PF_APPLETALK per i protocolli Appletalk PF_UNIX per specificare il file system di Unix
type	SOCK_STREAM per connection oriented (TCP) SOCK_DGRAM per connectionless (UDP) SOCK_RAW usi con privilegi
protocol	per specificare il particolare protocollo all'interno famiglia e del tipo

Uso di un socket

Una volta creato un socket può essere usato per comunicare o per attendere una comunicazione

Un socket usato da un server attende la comunicazione (*passive socket*)

Un socket usato da un client inizia la comunicazione (*active socket*)

Specificare un endpoint

Appena creato un socket non contiene alcuna informazione su come verrà utilizzato (attivo o passivo) e non ha informazioni sui numeri di porta o indirizzi IP (locali e remoti)

Communication endpoint **(IP, #porta)**

I socket offrono dei meccanismi per specificare gli indirizzi (address family)

TCP/IP usa la famiglia di indizzi denotata AF_INET (evitare la confusione con PF_INET)

Specificare un endpoint

I socket offrono due possibilità: definire una forma di endpoint generico oppure definire un formato dell'endpoint specifico per quella famiglia di protocolli

Gli endpoint TCP/IP sono composti dai seguenti campi:

- 2 byte per identificare il tipo di indirizzo (costante AF_INET);
- 2 byte per il numero di porta
- 4 byte per l'indirizzo IP

Specificare un endpoint

Struttura sockaddr_in

```
struct sockaddr_in  {           /* struct to hold an address */
      u_char sin_len;  /* total length */
      u_short sin_family;       /* type of address */
      u_short sin_port;         /* protocol port number */
      struct in_addr sin_addr; /* IP address *
      char sin_zero[8];         /* unused (set to zero) */
}

struct in_addr {        /* struct for IP address */
      u_long s_addr;  /* IP address */
};
```

Conversione tra rappresentazioni di interi

TCP/IP utilizza la rappresentazione network byte order (interi con il byte più significativo per prima)

Utilizzo di funzioni di conversione per esempio per il campo protocol port in *sockadd_in*

- htons() “host to network short”
- htonl() “host to network long”
- ntohs() “network to host short”
- ntohl() “network to host long”

Collegare il socket all'endpoint locale: la system call *bind* (server side)

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

sockfd = descrittore di un socket

my_addr = puntatore ad una struct sockaddr_in che contiene informazioni sull'endpoint locale

addrlen = sizeof(struct sockaddr)

Esempio di uso della *bind*

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#define MYPORT 3490
  main()
  {
      int sockfd;
      struct sockaddr_in my_addr;
      sockfd = socket(PF_INET, SOCK_STREAM, 0);
      my_addr.sin_family = AF_INET;
      my_addr.sin_port = htons(MYPORT);
      my_addr.sin_addr.s_addr = inet_addr("132.241.5.10");
      bzero(&(my_addr.sin_zero), 8);
      bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
```

Esempio di uso della *bind*

Alcune varianti

```
my_addr.sin_port = 0;

my_addr.sin_addr.s_addr = INADDR_ANY;
```

La bind restituisce -1 in caso di errore

- tentativi di assegnare al socket una porta < 1024
- o una porta già assegnata

La system call *listen()* (server side)

Un socket (lato server) deve essere “reso” passivo

listen(int sockfd, int qlen);

sockfd = descrittore di un socket

qlen = il numero massimo di connessioni in attesa (sul socket passivo)

La system call *accept()* (server side)

- serve ad accettare una richiesta di connessione

int accept(int sockfd, void *addr, int *addrlen);

sockfd = descrittore di un socket

addr = puntatore ad una struttura sockaddr_in dove verranno memorizzate le informazioni sulla connessione (indirizzo IP e porta locale del chiamante)

addrlen = sizeof(struct addr)

La accept crea un nuovo socket per ogni nuova richiesta di connessione

Il server utilizzerà il nuovo socket per la connessione mentre accetterà altre richieste di connessione mediante il socket originale (quello con sockfd)

Esempio di uso della *accept*

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#define MYPORT 3490    /* the port users will be connecting to */
#define BACKLOG 10     /* how many pending connections queue will hold */
main()
 {
    int sockfd, new_fd, sin_size;
    struct sockaddr_in my_addr, their_addr;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(MYPORT);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 8);
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
    listen(sockfd, BACKLOG);
    new_fd = accept(sockfd, &their_addr, &sin_size);
       .
```

La system call *accept()* (server side)

- Quando una richiesta di connessione arriva la chiamata ad *accept* termina
- Il server può gestire le richieste sia iterativamente che concorrentemente
- Nel primo caso è il server stesso che gestisce la richiesta, la serve e dopo provvede a chiudere (*close*) il socket creato dalla *accept* e successivamente ritorna ad eseguire la accept per la prossima richiesta di connessione
- Nell'approccio concorrente dopo la terminazione della *accept* il master server crea uno slave server per gestire la richiesta (il master server chiude il socket ottenuto dalla *accept*)
- Meccanismo degli endpoint per gestire la concorrenza

La system call *connect()* (client side)

connect(int sockfd, struct sockaddr *serv_addr, int addrlen);

sockfd = descrittore di un socket

serv_addr = puntatore ad una struct sockaddr_in che contiene informazioni sull'endpoint remoto quello a cui ci si connette

addrlen = sizeof(struct sockaddr)

Esempio di uso della *connect*

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#define DEST_IP   "132.241.5.10"
#define DEST_PORT 23
 main()
   {
    int sockfd;
    struct sockaddr_in dest_addr;
    sockfd = socket(PF_INET, SOCK_STREAM, 0);
    dest_addr.sin_family = AF_INET;        /* host byte order */
    dest_addr.sin_port = htons(DEST_PORT);
    dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
    bzero(&(dest_addr.sin_zero), 8);
    connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
        .
```

La system call *connect()*

La connect esegue 4 compiti:

- controlla che il socket (da connettere) non sia già connesso
- memorizza l'endpoint remoto nella struttura connessa al socket
- sceglie un endpoint locale (se il socket non è collegato ad già ad un endpoint locale)
- inizia una connessione TCP e restituisce un valore che indica successo o fallimento

La system call *connect()*

La connect esegue 4 compiti:

- controlla che il socket (da connettere) non sia già connesso
- memorizza l'endpoint remoto nella struttura connessa al socket
- sceglie un endpoint locale (se il socket non è collegato ad già ad un endpoint locale)
- inizia una connessione TCP e restituisce un valore che indica successo o fallimento

La system call *send()*

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int sockfd,char *buff, int nbytes,int flags)
```

- **buff* contiene l'indirizzo di memoria del dato da spedire
- *nbytes* = strlen(buff)
- *flags* può essere 0 oppure una combinazione di:
  - MSG_OOB: invia dati urgenti (PSH)
  - MSG_DONTROUTE: bypassa il routing
- Restituisce il numero di byte effettivamente inviati

La system call *sendto() e sendmsg()*

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockfd,char *buff, int nbytes,int flags, (struct sockaddr_in *)
    destraddr, int addresslen)
int sendmsg(int s, const struct msghdr *msg, int flags)
```

- *destraddr* è una struttura che contiene l'endpoint della destinazione

- *msghdr* è una struttura che contiene sia il messaggio da spedire che l'endpoint della destinazione

Esempio di uso della *send*

```
char *msg = “Ciao a tutti! ”;
 int len, byte_sent;
.
.
len = strlen(msg);
byte_sent = send(sockfd, msg, len, 0);
.
.
```

La system call *recv()*

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int sockfd,char *buff, int nbytes,int flags)
```

- **buff* contiene l'indirizzo di memoria dove viene memorizzato i dati ricevuti
- *nbytes* = strlen(buff)
- *flags* può essere 0 oppure una combinazione di:
  - MSG_OOB: riceve dati urgenti
  - MSG_PEEK: legge i dati senza rimuoverli dal buffer di sistema
- Restituisce il numero di byte effettivamente letti

La system call *recvfrom() e recvmsg()*

```
#include <sys/types.h>
#include <sys/socket.h>
int recvfrom(int sockfd,char *buff, int nbytes,int flags, (struct sockaddr_in *)
    sndaddr, int addresslen)
int recvmsg(int s, const struct msghdr *msg, int flags)
```

- *snaddr* è una struttura che contiene sia l'endpoint del mittente

- *msghdr* è una struttura che contiene sia il messaggio ricevuto che l'endpoint del mittente

send(), recv() vs *sendto(), recvfrom()*

- Le sistem call send() e recv() possono essere utilizzate solo con i socket connessi
- sendto(), recvfrom() si usano con i socket non connessi
- Differenza tra l'uso di un protocollo di transport connection oriented/connectionless e l'uso di socket connessi
  - si possono usare anche i socket SOCK_DGRAM connessi !!!

System call *close()* e *shutdown()*

close(int sockfd);

chiude la connessione (se il socket è connesso) e rimuove il socket

shutdown(int sockfd, int direction);

chiusura selettiva

direction=0 impedisce ulteriori ricezioni
direction=1 impedisce ulteriori spedizioni
direction=2 simile alla close

Altre procedure per la gestione dei socket

Int getppername(int sockfd, struct sockadd_in *addr, int *addrlen)

restituisce informazioni sulla controparte remota (si usa solo con i socket SOCK_STREAM)

```
telnet pianeta.di.unito.it
Trying 130.192.239.1...
Connected to pianeta.
```

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

struct hostent {
        char    *h_name; /* nome ufficiale dell'host */
        char    **h_aliases; /* una lista di nomi "alternativi" per l'host
        int     h_addrtype; /* il tipo di indirizzo --- AF_INET */
        int     h_length; /* lunghezza dell'indirizzo */
        char    **h_addr_list; /* lista di indirizzi di rete per l'host */
    };
    #define h_addr h_addr_list[0] /* primo indirizzo in h_addr_list
```

Interfacciamento con il DNS

Copia nel campo *sin_addr* di una struttura *sockaddr_in* l'indirizzo IP del calcolatore *pianeta.di.unito.it*

```
hostent *hp;
struct sockaddr_in serv_addr;

hp = gethostbyname("pianeta.di.unito.it");
bcopy(hp->h_addr,(char*)&serv_addr.sin_addr,hp->h_length);
```

Interfacciamento con il DNS (cont)

- Stampa dell'indirizzo simbolico dell'host avente indirizzo IP 130.192.239.1

```
hostent *hp;
struct in_addr IPAddr;

IPAddr.s_addr=inet_addr("130.192.239.1");
hp = gethostbyaddr((char*)&IPAddr,sizeof(IPAddr),AF_INET);
printf("%s\n",hp->h_name);
```

Esempio di utilizzo di *gethostbyname()*

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
 int main(int argc, char *argv[])    {
      struct hostent *h;
      if (argc != 2) {  /* error check the command line */
          fprintf(stderr,"usage: getip address\n");
          exit(1);
      }
      if ((h=gethostbyname(argv[1])) == NULL) {  /* get the host info */
          herror("gethostbyname");
          exit(1);
      }
  printf("Host name  : %s\n", h->h_name);
  printf("IP Address : %s\n",inet_ntoa(*((struct in_addr *)h->h_addr)));
  return 0;
  }
```

Impostazione dei parametri di un socket

- setsockopt() e getsockopt() permettono di impostare e leggere i valori di alcuni parametri di un socket

```
#include <sys/types.h>
#include <sys/socket.h>
int setsockopt(int sockfd,int level,int oname,char *oval,int olen)
int getsockopt(int sockfd,int level,int oname,char *oval,int *olen)
```

Conversione di indirizzi

inet_ntoa(): converte un indirizzo IP in notazione dotted decimal

inet_addr(): converte un indirizzo da notazione dotted decimal in formato IP (gia' in network order)

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
char *inet_ntoa(struct in_addr IPaddr)
unsigned long inet_addr(char *DottedAddr)
```

Blocking

Alcune system call sono bloccanti (recv, ecc.)

È possibile definire alcune system call non-bloccanti

```
#include <unistd.h>
#include <fcntl.h>
….
sockfd=socket(PF_INET, SOCK_STREAM, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

- In questo modo è possibile controllare diversi socket per verificare l'arrivo di dati
  - il risultato della *recv* (o simile) sarà -1 e *errno* restituirà EWOULDBLOCK
  - … gestione inefficiente (busy-wait)

select(): attesa non deterministica

La funzione select consente ad un processo di bloccarsi in attesa di dati su piu' socket simultaneamente

```
#include <sys/types.h>
#include <sys/time.h>
int select(int maxfd,fd_set *readfs,fd_set *writefs,fd_set *exc, struct timeval *timeout)
```

- readfs = socket pronti per la lettura
- writefs = socket pronti per la scrittura
- exc = socket su cui si verifica un errore
- timeout = tempo massimo di attesa
- valori di ritorno:
  - -1 se vi e' un errore
  - 0 se si verifica un timeout
  - n > 0 se n descrittori diventano pronti

select(): attesa non deterministica (cont)

fd_set: array di bit in cui l'i-mo elemento corrisponde all'i-mo descrittore di socket manipolabile tramite le macro

- FD_ZERO (fd_set *fdset): imposta a 0 tutti i bit di fdset
- FD_SET(int fd,fd_set *fdset): imposta ad 1 il bit corrispondente ad fd
- FD_CLR(int fd,fd_set *fdset): imposta a 0 il bit corrispondente ad fd
- FD_ISSET(int fd,fd_set *fdset): testa il bit corrispondente ad fd

Attenzione: il primo descrittore corrisponde all'elemento 0 dell'array fdset

select(): attesa non deterministica (cont)

Attesa che i socket associati ai descrittori *s1* ed *s2* (*s2* > *s1*) diventino pronti in lettura

```
fd_set F;
FD_ZERO(&F);
FD_SET(s1,&F);
FD_SET(s2,&F);
status = select(s2+1,&F,NULL,NULL,timeout);
if (status > 0)
   if (FD_ISSET(s1))  //sono arrivati dei dati su s1
   if (FD_ISSET(s2)) //sono arrivati dei dati su s2
}
else //timeout
```

select(): attesa non deterministica (cont)

Per gestire il timeout, si utilizza la struttura *timeval*

```
struct timeval {
   long tv_sec ;//secondi
   long tv_used;//microsecondi
};
```

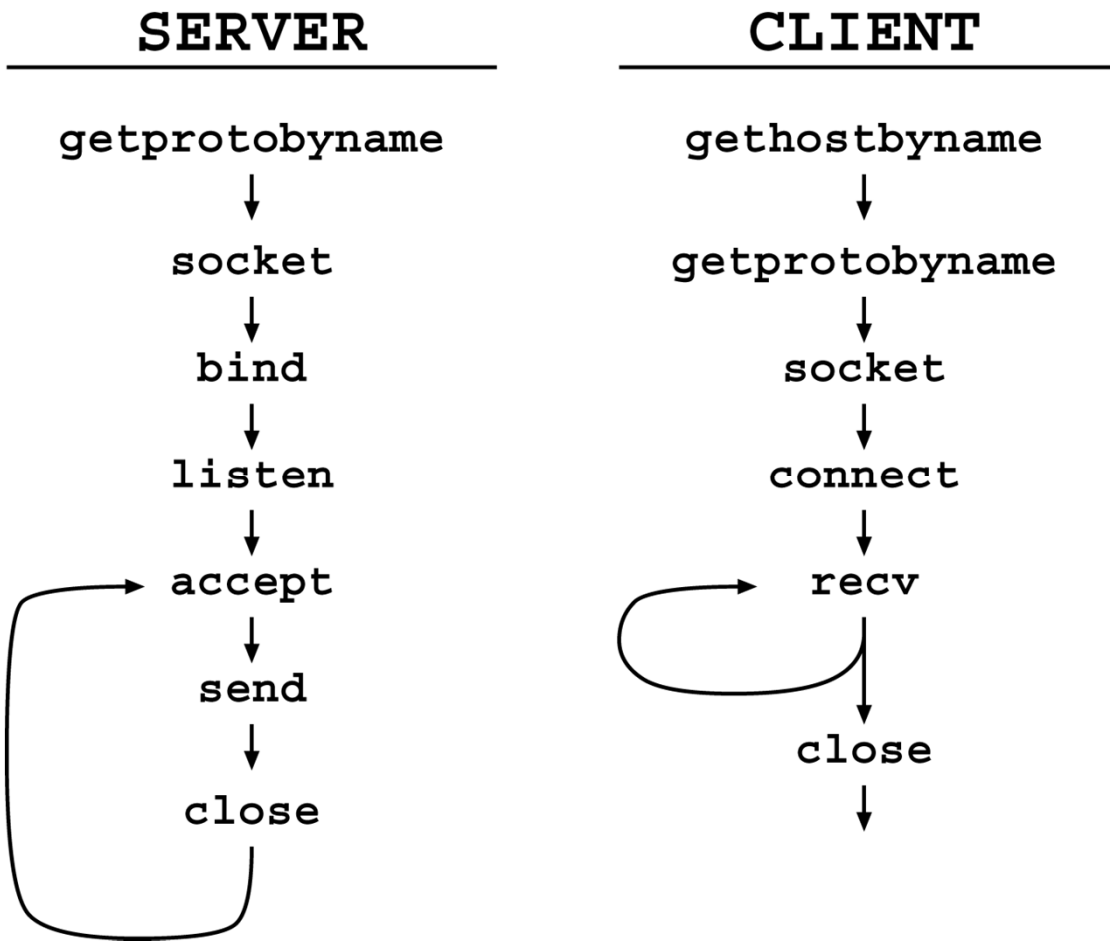
- Attesa per 3.045 secondi

```
struct timeval T;
T.tv_sec = 3;
T.tv_usec = 45000;
status = select(s2+1,&F,NULL,NULL,timeout);
```

Esempio di utilizzo di *select()*

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#define STDIN 0  /* file descriptor for standard input */
 main()       {
    struct timeval tv;
    fd_set readfds;
    tv.tv_sec = 2;
    tv.tv_usec = 500000;
    FD_ZERO(&readfds);
    FD_SET(STDIN, &readfds);
    select(STDIN+1, &readfds, NULL, NULL, &tv); /* writefds e exceptfds non
   sono controllati */
      if (FD_ISSET(STDIN, &readfds))
         printf("A key was pressed!\n");
      else
            printf("Timed out.\n");
      }
```

Uso delle system call in applicazioni CL/SRV



Progettazione di client

Identificare l'endpoint dell'applicazione server

- l'endpoint memorizzato come costante (poca flessibilità)
- endpoint fornito come input al momento dell'esecuzione del client
- informazione recuperata da un file o da mediante l'uso di una system call
- utilizzare un protocollo per ottenere l'endpoint del server (mediante messaggi di broadcast o multicast ai quali il server risponde)

Progettazione di client

```
struct servent {
   char *s_name;  /* nome del servizio */
   char **s_aliases; /* altri alias */
   int s_port; /* num. di porta */
   char *s_proto; /* protocollo da utilizzare */
};

struct servent *sptr;
if(sptr = getservbyname(“smtp”, “tcp”)) {
   /* il numero di porta adess è memorizzano in sprt->s_port */
else { /*errore */}
}
```

Progettazione di client (con TCP)

Trovare l'endpoint del server

Allocare un socket

Specificare che la connessione necessita di una porta arbitraria (locale)

Connettere il socket al server

Comunicare con il server

Chiudere la connessione

Progettazione di client (con UDP)

- Trovare l'endpoint del server
- Allocare un socket
- Specificare che la comunicazione necessita di una porta arbitraria (locale)
- Specificare l'endpoint remoto (del server) al quale i messaggi vanno inviati (struttura sockaddr_in)
- Comunicare con il server (sendto e recvfrom)
- Chiudere il socket

Esempio: Client e Server Daytime

Semplice server che invia a ciascun client che richiede una connessione l'ora indicata dall'orologio locale del calcolatore sul quale e' in esecuzione

Il client non invia nessun comando esplicito

Il server attende le connessioni sulla porta TCP 2000

Client daytime - 1/3

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

#define MAXLINE 256
#define PORT 2000

int main(int argc, char **argv)
{
        int                     sockfd, n;
        char                    recvline[MAXLINE + 1];
        struct sockaddr_in      servaddr;
        struct hostent          *hp;
        struct in_addr          **pptr;

        if (argc != 2){
                fprintf(stderr,"usage: %s <IPaddress>\n", argv[0]);
                exit (1); }
```

Client daytime - 2/3

```
        if ( (sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0){
                perror("socket error");
                exit (1);
        }

        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port   = htons(PORT);

        /* Map host name to IP address, allowing for dotted
decimal */
        if ( (hp = gethostbyname(argv[1])) == NULL)
          fprintf(stderr, "hostname error for %s", argv[1]);

        pptr = (struct in_addr **) hp->h_addr_list;
        memcpy(&servaddr.sin_addr, *pptr, sizeof(struct
in_addr));
```

Client daytime - 3/3

```
if (connect(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr)) < 0){
      perror("connect error");
      exit (1);
   }

 while ( (n = recv(sockfd, recvline, MAXLINE,0)) > 0) {
      recvline[n] = 0;
      if (fputs(recvline, stdout) == EOF){
          perror("fputs error");
          exit (1);
       }
    }

if (n < 0){
  perror("read error");
  exit (1);
 }
 close(sockfd);
exit(0);
}// of main
```

Server daytime: struttura di base

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

#define MAXLINE 256
#define PORT 2000

int main(int argc, char **argv)
{
        int                     listenfd, connfd, addrlen;
        struct sockaddr_in      servaddr;
        char                    buff[MAXLINE];
        time_t                  ticks;

        < INIZIALIZZAZIONE>
        < CICLO DI ATTESA ED ELABORAZIONE RISPOSTE>
}
```

Server daytime: inizializzazione

```
listenfd = socket(AF_INET, SOCK_STREAM, 0);
if ( listenfd <0){
                perror("opening socket");
                exit(1);
 }

bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family      = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port        = htons(PORT);
a=bind(listenfd, (struct sockaddr *) &servaddr,
   sizeof(servaddr);
if (a < 0){
    perror("Error in binding");
    exit(1);
}

listen(listenfd, 5);
```

Server daytime: server iterativo

```
 for ( ; ; ) {
    connfd = accept(listenfd, (struct sockaddr *) NULL, NULL);
    ticks = time(NULL);
    snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
    send(connfd, buff, strlen(buff),0);
    close(connfd);
  }
}
```

Servizio daytime: server concorrente

```
while(1) {
 addrlen = sizeof(claddr);
 connfd = accept(listenfd, (struct sockaddr *)&claddr,&addrlen);
 if(connfd>0) {
    if(fork()>0) close(connfd); //processo padre
    else {
       close(listenfd);
       ticks = time(NULL);
       snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
       send(connfd, buff, strlen(buff),0);
       close(connfd);
       exit(0);
    }
    while(waitpid(-1,NULL,WNOHANG) > 0);
 }
 else printf("Errore in accept\n");
}
```

Ottenere l’indirizzo del client connesso

```
connfd = accept(listenfd, (struct sockaddr *)&claddr,&addrlen);
printf("Connessione con %s, porta %d\n",
   inet_ntoa(claddr.sin_addr),ntohs(claddr.sin_port));
```

Uso di getsockname, getpeername Client 1/2

```
int main(int argc, char **argv)
{
        int                     sockfd, n,cliaddr_len;
        char                    recvline[MAXLINE + 1];
        struct sockaddr_in      servaddr, cliaddr;
        struct hostent          *hp;
        struct in_addr          **pptr;

        if (argc != 2){
                fprintf(stderr,"usage: %s <IPaddress>\n", argv[0]);
                exit (1);
        }

        if ( (sockfd = socket(PF_INET, SOCK_STREAM, 0)) < 0){
                perror("socket error");
                exit (1);
        }
        bzero(&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        servaddr.sin_port   = htons(PORT);
```

Uso di getsockname, getpeername Client 2/2

```
if ( (hp = gethostbyname(argv[1])) == NULL)
  fprintf(stderr, "hostname error for %s", argv[1]);
pptr = (struct in_addr **) hp->h_addr_list;
memcpy(&servaddr.sin_addr, *pptr, sizeof(struct in_addr));
if(connect(sockfd,(struct sockaddr *) &servaddr,sizeof(servaddr))< 0){
                perror("connect error");
                exit (1);
        }
getsockname(sockfd, (struct sockaddr *) &cliaddr, &cliaddr_len);
printf("Connessione con %s, porta %d\n",
inet_ntoa(cliaddr.sin_addr),ntohs(cliaddr.sin_port));
    while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
        recvline[n] = 0;
          if (fputs(recvline, stdout) == EOF){
                  perror("fputs error");
                  exit (1);
          }
       }
    if (n < 0){ /* read error */ }
}
```

Uso di getsockname, getpeername Server 1/3

```
int main(int argc, char **argv)
{
 int                    cliaddr_len;
 int                    listenfd, connfd;
 struct sockaddr_in     cliaddr, servaddr;
 char                   buff[MAXLINE];
 time_t                 ticks;
 if ( (listenfd = socket(PF_INET, SOCK_STREAM, 0)) <0){
         perror("opening socket");
         exit(1);
 }
 bzero(&servaddr, sizeof(servaddr));
 servaddr.sin_family      = AF_INET;
 servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
 servaddr.sin_port        = htons(PORT);
 if(bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0)
            perror("Error in binding");
            exit(1);
 }
 listen(listenfd, 5);
 signal(SIGCHLD, gestisci_zombie);
```

Uso di getsockname, getpeername Server 2/3

```
for ( ; ; ) {
     cliaddr_len = sizeof(cliaddr);
     for( ; ; ){
       connfd = accept(listenfd, (struct sockaddr *) &cliaddr,
                               &cliaddr_len);
       if (connfd > 0)              break;

       if (errno != EINTR){
            perror("Errore in accept");
            exit(-1);
       }
    }
    printf("1. Indirizzo remoto\t %s, porta %d\n",
            inet_ntoa(cliaddr.sin_addr), ntohs(cliaddr.sin_port));
```

Uso di getsockname, getpeername Server 3/3

```
    if (fork() == 0){
       struct sockaddr_in  child_servaddr;
       close(listenfd);
       cliaddr_len = sizeof(cliaddr);
       getpeername(connfd, (struct sockaddr *) &cliaddr, &cliaddr_len);
       printf("2. Indirizzo remoto\t %s, porta %d\n",
          inet_ntoa(cliaddr.sin_addr), ntohs(cliaddr.sin_port));
       cliaddr_len = sizeof(child_servaddr);
       getsockname(connfd, (struct sockaddr *)
              &child_servaddr,&cliaddr_len);
       printf("3. Indirizzo locale\t %s, porta %d\n",
          inet_ntoa(cliaddr.sin_addr), ntohs(child_servaddr.sin_port));
       ticks = time(NULL);
       snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
       write(connfd, buff, strlen(buff));
       close(connfd);
       exit(0);
      }
      close(connfd); /* nel padre */
    }
}
```

Per evitare gli zombi

```
#include <signal.h>
#include <sys/wait.h>

void
gestisci_zombie (int segnale)
{

        int status;

        while(waitpid(-1, &status, WNOHANG)>0)
                ;
        return;

}
```