TCP/IP szállítási réteg (socket programozási interfész)

Összeállította: Mohácsi János
BME Informatikai Központ

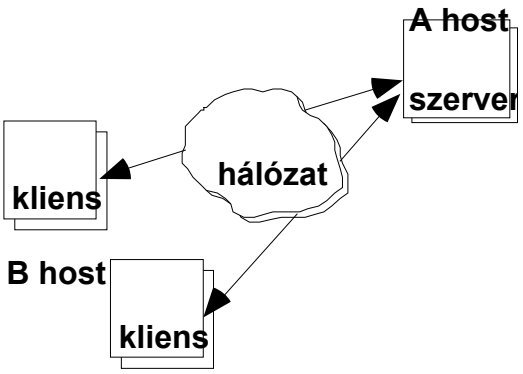
1 TCP/IP

1.1 TCP/IP ÉS SOCKET INTERFÉSZ VISZONYA

A TCP/IP réteges felépítése:

Alkalmazói program réteg
Szállítási réteg
Internet réteg
Adaptációs réteg
Hordozó hálózat réteg

A socket interfészt az 1980-as évek elején, a University of California at Berkeley (UCB) egyik laborjában dolgozták ki *TCP/IP* kezelői felületeként, *UNIX* operációs rendszerekhez, azóta *UNIX* szabványként terjedt el, implementálva van a Windowsban is WinSock néven. A socket egy API (Application Program Interface), ami eredetileg Berkeley-UNIX -ban (4.x) jelent meg. System V. Release 4 is átvette, de abban van más API is erre: TLI = Transport Layer Interface (Lásd szállítási interfész vizsgálata). A socket processzek közötti kommunikációs eszköz, mely független attól, hogy a kommunikáló processzek azonos, vagy különböző gépen vannak-e, illetve amennyire lehet, elfedik a kommunikáció megvalósításának alsóbb szintjeit. A kommunikáció formája kliens-szerver alapú, ami azt jelenti, hogy a szerver valamilyen jól definiált szolgáltatást nyújt, amit a kliensek igénybe vesznek.

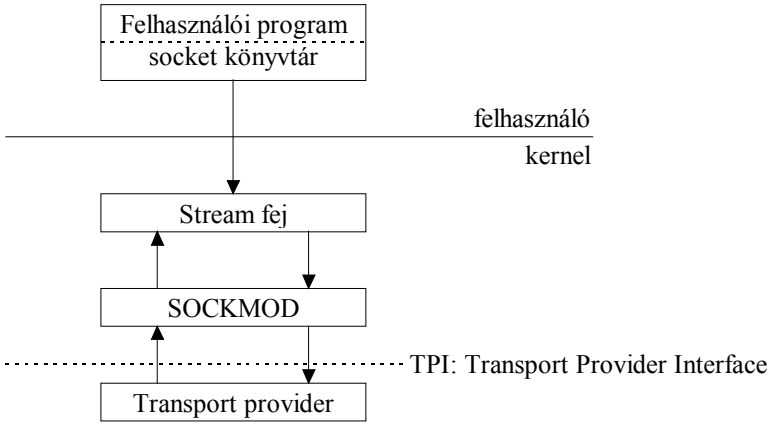


1.2

Ábra a kliens szerver kommunikációról.

Lehet kapcsolat-orientált (connection-oriented) vagy kapcsolat nélküli (connectionless): minden adatátviteli kérésnél meg kell adni a célcímet.
Szerver lehet iteratív (ilyenkor a szerver ciklusban vár, azonnal feldolgozza a beérkezett kérést, tipikusan akkoralkalmazott, ha minden kérés feldolgozás véges és rövid idejű), vagy konkurens (ilyenkor a szerver minden kiszolgáláskor `fork()`-ol, gyereke dolgozza fel, szülő újra vár ciklusban a következő kérésre.) Ezek különböző "kombinációja" is megvalósitható (preforked, filedescriptor passing, stb.) ha nem blokkolódó vagy/és aszinkron rendszerhivásokat alkalmazunk.

1.2.1.1 BSD socket szerkezete:



1. ábra: Socket szerkezet SVR4 UNIX-ban

- a felhasználói program szinte mint egy filedescriptort látja a socket-et. Bár nem minden filedescriptor müvelet müködik (pl. `lseek` -> `ESPIPE` üzenet)

1.3 A SOCKET RENDSZERHÍVÁSOK

A Socket és TLI összehasonlítása a rendelkezésre álló rendszerhívások tekintetében:

Szerep	*Funkció*	*Socket*	*TLI*
Szerver	allokáció/helyfoglalás	-	`t_alloc()`
	végpont létrehozás	`socket()`	`t_open()`
	címhez rendelés	`bind()`	`t_bind()`
	queue méret állítás	`listen()`	-
	várakozás kapcsolat kérelemre	`accept()`	`t_listen()`

	új filedescriptor létrehozás		`t_open()`
			`t_bind()`
			`t_accept()`

Kliens	allokáció/helyfoglalás	-	`t_alloc()`
	végpont létrehozás	`socket()`	`t_open()`
	címhez rendelés	`bind()`	`t_bind()`
	szerverhez kapcsolódás	`connect()`	`t_connect()`

Mindkettő	adatátvitel (kapcsolat orintált)	`read()`
		`write()`
		`recv()`
		`send()`
			`read()`
			`write()`
			`t_rcv()`
			`t_snd()`

	adatátvitel (kapcsolat nélküli)	`recvfrom()`
		`sendto()`
		`recvmsg()`
		`sendmsg()`
			`t_rcvudata()`
			`t_sndudata()`

	kapcsolat lezárás	`close()` `shutdown()`	`t_close()` `t_sndrel()`

			t_snddis()

Ezenkívül a következő rendszerhívásokat szokás használni hálózati programozáskor:

`ioctl()`	eszközvezérlő funkció
`fcntl()`	fileleírót vezérelő funkció
`select()`	szinkron/aszinkron multiplexelt I/O beállítása és várakozás egy I/O eseményre

Ezekre részletesen a multiplexelt, aszinkron I/O és nem blokkolódó I/O fejezetekben térünk ki.

1.4 A SOCKET INTERFÉSZ ÁLTALÁNOS LEÍRÁSA

1.4.1.1 A socket egy "communications domain" -hoz kapcsolódik a létrehozáskor:

- Lokális vagy UNIX domain (AF_LOCAL vagy AF_UNIX): gépen belüli kommunikációhoz
- Internet domain (AF_INET, AF_INET6): hálózaton át, a következő protokollokkal:
  - TCP: Transmission Control Protocol
  - UDP: User Datagram Protocol
  - IP: Internet Protocol
  - ICMP: Internet Control Message Protocol

(Ezeket együtt szokás TCP/IP -nek is nevezni.)

Egyéb lehetséges protokollok: ATM, IPX, Appletalk, DECNET, XNS, OSI, de egyes esetekben IPSec kulcsmenedzsment (PF_KEY), hálózati kártyaszintű (Netlink)

1.4.1.2 Az átvitel típusai:

- stream socket (**`SOCK_STREAM`**): kapcsolat-orientált byte-stream: sorrendtartó, megbízható, ismétlődés nélküli, nincs üzenethatár. Tényleges adatküldés előtt a kapcsolatot fel kell építeni.
- datagram socket (**`SOCK_DGRAM`**) : kapcsolat nélküli, üzenet-alapú, nem megbízható
- egyszerű (raw) socket (**`SOCK_RAW`**) : közvetlen hozzáférés a protokollhoz, annak adminisztrációjához, így a felhasználó építhet saját rétegeket
- sorrendes socket (**`SOCK_SEQPACKET`**) hasonló a streamhez, de

üzenethatárokat is közvetíti, kapcsolat-orientált, megbízható

- megbízható üzenet socket (**SOCK_RDM**: reliably delivered message socket): megbízható, kapcsolat nélküli, üzenet-orientált

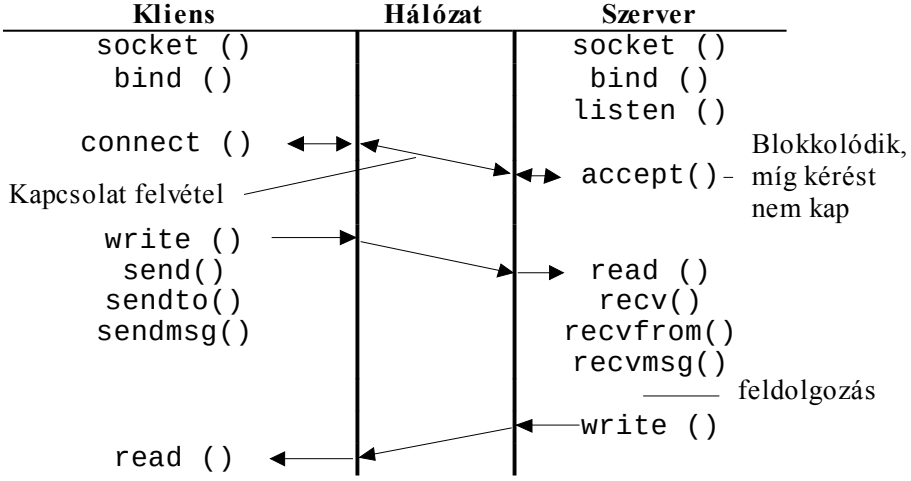
Nem minden domain tartalmaz minden socket típusra megvalósítást.

Átvitel típusa	Kapcsolat	sorrendtartó	megbízható	ismétlés nélk.	üzenethatár
SOCK_STREAM	-orientált	i	i	i	n
SOCK_DGRAM	nélküli	n	n	n	i
SOCK_RAW					
SOCK_SEQPACKET	-orientált	i	i	i	i
SOCK_RDM	nélküli	n	i	n	i

1.4.1.3 Protokoll-default-ok Internet protokollok (AF_INET és AF_INET6) esetén:

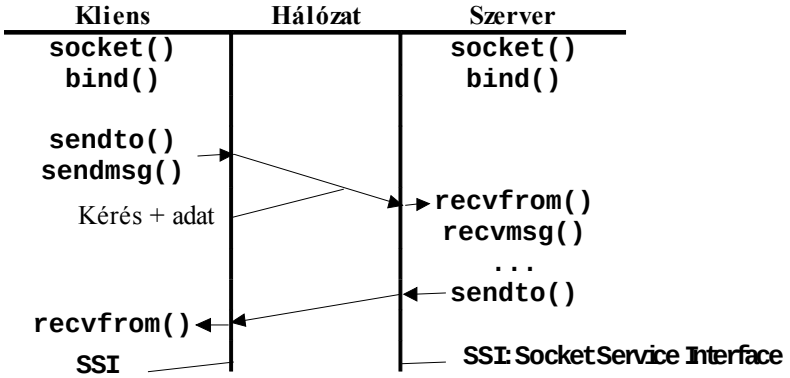
	AF_LOCAL	*AF_INET*	*AF_INET6*
SOCK_STREAM	+	TCP	TCP
SOCK_DGRAM	+	UDP	UDP
SOCK_RAW		*Protokolltól függően* IPv4/ICMPv4/IGMPv4	*Protokolltól függően* IPv6/ICMPv6
SOCK_SEQPACKET			

1.4.1.4 A kapcsolat-orientált idődiagram:



2. ábra: Kapcsolatorientált működés idődiagramja

1.4.1.5 A kapcsolat-nélküli idődiagram:



3. ábra: Kapcsolat nélküli működés idődiagramja

Minden "kapcsolatot" egy 5-ös azonosít:
{Protokoll, helyi gépcím, helyi processz portszám, távoli gépcím, távoli processz portszám}

Az 5-ös csoport megváltoztatására a a következő rendszerhívások állnak rendelkezésre attól függően, hogy milyen szerepet játszik a program:

	protokoll	***helyi gépcím, helyi process portszám***	***távoli gépcím, távoli processz portszám***
Kapcsolat orientált szerver	`socket()`	`bind()`	`listen(),accept()`
Kapcsolat orientált kliens	`socket()`	`bind()`	`connect()`
Kapcsolat nélküli szerver	`socket()`	`bind()`	`recvfrom()`
Kapcsolat nélküli kliens	`socket()`	`bind()`	`sendto()`

Lekérdezésére a getsockname() és a getpeername() rendszerhívások szolgálnak. Az accept() rendszerhívás előtt a szerver socket távoli gépcíme és távoli processz portszáma egy "mindent elfogadó értéket" tartalmaz. Az accept() hatására egy újabb socket keletkezik, és a régi socket akár képes lehet újabb kérés kiszolgálására.

1.5 PROGRAMOZÓI INTERFÉSZ

1.1.1 Socket létrehozása

1.5.1.1 Socket

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

int socket (int family,  int type,  int protocol);
```

= file leíró: OK, -1: hiba (kódja: errno)

`family:` communications domain: két konstans-készlet adhatja meg, melyek páronként azonosak:

- address family: cím formátuma
- protocol family: hálózati architektúra :
  UNIX/Local domain: `PF_UNIX = AF_UNIX`
  Internet domain: `PF_INET = AF_INET`

`type`: socket típusa: jelenleg csak

- `SOCK_STREAM, SOCK_DGRAM, SOCK_RAW` van.

`protocol:` a használandó protokoll típusa:

- 0: a default a `family` és a `type` szerint, vagy
- `IPPROTO_ICMP, IPPROTO_IGMP, IPPROTO_TCP, IPPROTO_UDP`**`,stb.`** (`<netinet/in.h>` -ban)

1.5.2 Socket hozzárendelése hálózati címhez

Létrehozása után, elsősorban szervernél kell, e nélkül (pl. a kliensnél) automatikusan egy egyedi címhez rendelődik.

1.5.2.1 Bind

```
int bind (int fd,  struct sockaddr *addrp,  int alen);
```

`Fd`: File leíró, melyet **`socket()`** adott,

`Addrp:` Címleíró struktúra címe:

```
struct sockaddr {
          ushort_t  sa_family;  /* mint socket()-ben 2
                                   byte */
          char      sa_data [14];
};
```

vagy BSD4.4-ben:

```
struct sockaddr {
          u_char sa_len;
          u_char sa_family;
          char sa_data [14];
}
```

De igazából 16 byte-nál hosszabb is lehet, de ilyenné kell a címét cast-olni.
A socket hosszának lekérdezésére és beállítására a BSD 4.4-ben sa_len szolgál.

1.5.3 Socket lezárása

1.5.3.1 Close

```
close (int fd)
```

Ekkor a még át nem adott adatok elveszhetnek, vagy még átvehetőek, lásd SO_LINGER opció.

1.5.3.2 Shutdown

Csak kapcsolat-orientált (connected) socket-re egyirányú lezárása socketnek.

```
int shutdown (int fd, int how);    /* =0 :OK, -1 :hiba */
```

how = 0: nem lehet több adatot átvenni a socket-től,
= 1: nem lehet több adatot átadni a socket-nek,
= 2: egyik sem megy (ekvivalens a close()-al)

1.5.4 Opciók beállítása ill. lekérdezése

Melyeket a socket réteg dolgoz fel, így protokoll-függetlenek.

```
int setsocketopt (int fd, int level, int cmd, char *arg, int len);
int getsocketopt (int fd, int level, int cmd, char *arg, int *lenp);
```

level: milyen szintű működést állítunk be (pl. SOL_SOCKET: socketét)
Pl. ha cmd = SO_ERROR, arg által mutatott int-be leteszi a hibakódot. Következőkben a legfontosabbakat soroljuk fel:

level	opciónév	get	set	Leírás	mód	adat

IPPROTO_IP	IP_OPTIONS	+	+	IP opciókvaló hozzáférés	beállít/leolvas	void *
	IP_INCLFDR	+	+	teljes IP header hozzáférhető	engedélyez/tilt	int
IPPROTO_TCP	TCP_MAXSEG	+	+	maximális TCP szegmens méret	beállít/leolvas	int
	TCP_NODELAY	+	+	TCP adat azonnali elküldése (nagy késleltetésű hálózaton ne alkalmazzuk)	engedélyez/tilt	int
SOL_SOCKET	SO_BROADCAST	+	+	Broadcast engedélyezése/tiltása	engedélyez/tilt	int
	SO_DEBUG	+	+	Debuggolás engedélyezése (Kernel specifikus üzenetek)	engedélyez/tilt	int
	SO_DONTROUTE	+	+	Ne routold az üzenetet.	engedélyez/tilt	int
	SO_ERROR	+		A socket hibalekérdezése.	beállít/leolvas	int
	SO_KEEPALIVE	+	+	Akkor is küldj, ha nincs mit küldeni, különben a socket lebomlik.	engedélyez/tilt	int
	SO_LINGER	+	+	close() után mennyi ideig probálkozzon az elküldetlen adat elküldésével.	beállít/leolvas	struct linger
	SO_OOBINLINE	+	+	Az OOB adatok is kerüljenek az inputra, vagy külön kelljen feldolgozni.	engedélyez/tilt	int
	SO_RCVBUF	+	+	Az olvasási buffer méretének beállítása (A teljesítmény változtatásának eszköze)	beállít/leolvas	int
	SO_SNDBUF	+	+	Az irási buffer méretének beállítása (A teljesítmény változtatásának eszköze)	beállít/leolvas	int
	SO_RCVLOWAT	+	+	olvasási buffer alsóértéke (byte-ban), ami meghatározza, hogy mikor kell a socketnek feladni a megérkezett adatot	beállít/leolvas	int
	SO_SNDLOWAT	+	+	irási buffer alsóértéke (byte-ban), ami meghatározza, hogy mikor kell a hálózatnak átadni az elküldendő adatot	beállít/leolvas	int
	SO_RCVTIMEO	+	+	olvasási buffer alsóértéke (időben), ami meghatározza, hogy mikor kell a socketnek feladni a megérkezett adatot	beállít/leolvas	struct timeval
	SO_SNDTIMEO	+	+	irási buffer alsóértéke	beállít/leolvas	struct timeval

				(időben), ami meghatározza, hogy mikor kell a hálózatnak átadni az elküldendő adatot		
	SO_REUSEADDR	+	+	engedélyezi, hogy az 5-öst azonnal újra lehessen használni ami azonosítja a kapcsolatot.	engedélyez/tilt	int
	SO_TYPE	+	IPv6	A socket típusának lekérdezése ill. cím konvertálása	beállít/leolvas	int

1.5.5 Kapcsolat létrehozása

Két socket közötti kapcsolat létrehozása, tipikusan kliensben;

- stream: virtuális áramkör jön étre
- datagram: megjegyzi a célcímet, azt nem kell az egyes átviteleknél később külön megadni, ill. átvenni csak a megadott címről jött adatokat lehet.

(Ha nincs `bind`-elve, `connect()` egy új címhez rendeli hozzá a socket-et.)

1.5.5.1 `Connect`

```
int connect (int fd, struct sockaddr *addrp, int alen);
```

`addrp`: cél-cím (címzett címe)

`fd, alen`: mint `bind()` -nél

= 0: OK, -1 : hiba, de a cím hozzárendelése (bind) ekkor is megmarad.

1.5.5.2 Connect kérések elfogadási szándékának jelzése és a queue méretének beállítása(a szerver oldalon):

Csak kapcsolat-orientált socket-nél (`SOCK_STREAM, SOCK_SEQPACKET`)

```
int listen (int fd,  int backlog);
```

`backlog`: hány feldolgozatlan `connect` kérést tárol (Nem teljesen, hanem egy exponenciális képlet szerint számolja ki.)

Ilyenkor egy ún. passzív socket keletkezik.

1.5.5.3 Connect elfogadása (a szerver oldalon):

Csak kapcsolat-orientált socket-nél.

```
int accept (in fd,  struct sockaddr *addrp,  int *alenp);
```

`addrp`:ide teszi le a kliens címét, ha nem érdekel, **`NULL`**

`*alenp`:híváskor: az addrp-ként átadott buffer hossza, visszatéréskor: a kapott cím hossza. Ha nem érdekel, alenp==NULL
= -1, ha hiba,
= új file leíró, ha sikeres, mely a kapott connect hívásnak felel meg, örökli a bemeneti `fd` tulajdonságait, pl. típus, protokoll. A bemeneti `fd` nem vesz részt az így elfogadott hívás további műveleteiben, hanem újabb accept hívások fogadására használható.

Ha nincs éppen várakozó connect kérés, accept felfüggeszti a hívó processt. Nem tud eleve elutasítani bizonyos hívásokat, de elfogadás (accept) után azonnal lezárhatjuk azokat.

1.5.6 Adatok küldése

Kapcsolat-orientált esetben használható a `write()` is, ezzel az átvitel a fileműveletekhez hasonlóan végezhető, sőt a processz esetleg nem is tudja, hogy éppen socket-en át kommunikál.

További eszközök csak kapcsolat-orientált socket-re:

1.5.6.1 Send

```
int send (int fd,  char *buf,  int len,  int flags);
    = -1: hiba, >= 0: átvitt byte-szám
```

`flags`:

- `MSG_OOB`: out-of-band: nagyobb prioritással viszi át, ha a protokoll támogatja ezt
- `MSG_DONTROUTE`: ha lehet, közvetlenül a címzettnek küldi, nem használ útvonal választást (csak lokális hálózaton működik).

Kapcsolat nélküli socket-re is:

1.5.6.2 Sendto

```
int sendto (int fd, char *buf, int len, int flags, /* mint send() */
              struct sockaddr *addrp, int alen);  /* mint connect()
                                                     */
```

= -1: hiba, >= 0 : átvitt byte-szám

Legáltalánosabb:

1.5.6.3 Sendmsg

```
int sendmsg (int fd,  struct msghdr *msgp,  int flags);
```

Visszatérési érték, `fd, flags`: mint `send()`-nél

```
struct msghdr {
  caddr_t  msg_name;       /* cél-cím, elhagyható  (char *) */
  int      msg_namelen;    /*   ennek hossza */
  iovec_t *msg_iov;        /* átviend• területek leírása */
  int      msg_iovlen;     /*   ennek elemszáma, <=
                            MSG_MAXIOVLEN */
  caddr_t  msg_accrights;  /* file leírók UNIX domain esetén */
  int      msg_accrno;     /*   ennek elemszáma */
};
```

Ebben:

```
typedef struct iovec {     /* egy átviend• terület leírása */
  caddr_t iov_base;        /* terület címe */
  int     iov_len;         /* terület hossza */
} iovec_t;                 /* típusnév */
```

Célcímet `SOCK_STREAM` esetén figyelmen kívül hagyja, előtte `connect()` kell.
`SOCK_DGRAM` esetén a célcím szerintire küldi, ha nincs célcím (pl. `write(), send()`) és előzőleg volt `connect()`, az ott megadottra.

1.5.7 Adatok átvétele

Kapcsolat-orientált esetben használható a `read()` is, lásd küldés.
Az itt ismertetendő többi eszköz bármelyik fajta socket-hez használható. `SOCK_STREAM` esetén előbb `connect()`-elni kell.
Ha adat előbb megérkezik, minthogy a címzett `bind()`-elné a socket-jét, a datagram-ot eldobja.

1.5.7.1 Recv

```
int recv (int fd,  char *buf,  int maxlen,  int flags);
```

>=0: kapott byte-szám, = -1: hiba

`flags:`

- `MSG_OOB`: csak ilyen flag-el küldött adatokat vesz át, egyébként bármelyiket, az ilyennel küldötteket előbb kapja meg, mint a többit
- `MSG_PEEK`: az átvett adatokat nem törli a bufferből, csak újabb, ilyen flag nélküli hívásnál

Datagram socket esetén ezzel nem tudható meg, ki küldte.

1.5.7.2 Recvfrom

```
int recvfrom (inf fd,  char *buf,  int maxlen,  int flags,  /* mint
                                  recv() */ struct sockaddr *addrp,
                                  int *alenp);  /* mint accept() */
```

= mint `recv()`

`*addrp`: innen jött, ide lehet a választ küldeni

1.5.7.3 Recvmsg

```
int recvmsg (int fd,  struct msghdr *msgp,  int flags);
```

Paraméterek: mint `sendmsg()`, visszatérési érték: mint `recv()`

1.5.8 Saját socket-cím lekérdezése (pl. gyerek-folyamatban):

1.5.8.1 Getsockname

```
int getsockname (int fd,  struct sockaddr *addrp,  int *alen);
```

= 0 :OK, =-1: hiba; paraméterek: mint `accept()`

1.5.9 Partner socket-címének lekérdezése:

1.5.9.1 Getpeername

```
int getpeername (int fd,  struct sockaddr *addrp,  int *alen);
```

= 0: OK, =-1: hiba; paraméterek: mint `accept()`

A 0 ... 1023 port-címeket általában privilegizált felhasználók kaphatják csak meg.

1.5.10Név <-> cím fordítás

Rutinkészlet, a cím egyes részeinek fordítására, az `/etc` könyvtárban található megfelelő file-ok alapján és a DNS alapján.
Mindegyik valamilyen struktúra címét adja vissza, amely a rutinban definiált statikus adatra mutat. Újabb hívás e területet felülírja, ezért ha később is szükség van rá, le kell másolni.

1.5.10.1 Host név <---> host Internet cím fordítás:

File: `/etc/hosts`
Ha van DNS a hálózaton, akkor ott is keresheti.

Paraméterek átvételéhez:

```
struct hostent {                    /* definíciója <netdb.h> -ban */
   char  *  h_name;                 /* host hivatalos neve */
   char  ** h_aliases;              /* host-név alias lista, végén 0
                                    pointer */
   int      h_addrtype;             /* cím-típus, kés•bbi b•vítéshez,
                                    most: AF_INET */
   int      h_length;               /* kés•bbi b•vítéshez, most: 4 */
   char  ** h_addr_list;            /* hálózati címek listája, egy
                                    hostnak több is lehet, binárisan,
                                    nagy-kicsi byte-sorrendben */

};
```

1.5.10.2 Gethostent

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

struct hostent * gethostent (void);
```

`/etc/hosts` file megnyitása, ha első hívás, első ill. következő bejegyzés olvasása, végül, vagy hiba esetén 0-t ad.

1.5.10.3 Endhostent

`void endhostent (void);`

`/etc/hosts` file olvasásának lezárása

1.5.10.4 Gethostbyaddr

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

Cím szerinti keresés `/etc/hosts` file-ban,
= struct cím, ha megvan,
=0, ha nincs, ekkor `h_errno = HOST_NOT_FOUND` jelzi az egyetlen lehetséges hibaokot, de ha van egy un. DNS is, mely domain name service-t csinál, akkor azt is használja a kereséshez, ekkor **`h_errno =`**

- `TRY_AGAIN`: átmeneti hiba,
- `NO_RECOVERY`: fatális hiba,
- `NO_DATA`: ez sem találja

Utána lezárja a file-t, ha a stayopen értéke nulla volt `sethostent()`-ben, ezért újabb `hostent()` hívás ismét elölről kezdi olvasni a file-t.

1.5.10.5 Gethostbyname

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

Név szerinti keresés `/etc/hosts` file-ban, név eredeti vagy bármelyik alias is lehet

= mint `gethostbyaddr()` esetén, ez is lezárja, ha ...

1.5.10.6 `Sethostent`

```
void sethostent (int stayopen);
```

Visszaáll a file elejére, és ha a `stayopen != 0` egyszer is, a következő `endhostent()` hívásig nem zárja majd le `gethostbyaddr()` ill. `gethostbyname()` végrehajtása után, azaz az előző `stayopen !=0` hatása megmarad.

1.5.10.7 getaddrinfo

(megírandó)

1.5.10.8 Hálózat név <---> hálózat-szám fordítás:

File: `/etc/networks` **`vagy DNS`**

1.5.10.9 Lekérdezések: mint . . . `host` . . . fv.-eknél:

```
struct netent * getnetent (void);
  void endnetent (void);
  struct netent * getnetbyaddr (long net, int type);
   struct netent * getnetbyname (char *name);
  void setnetent (int stayopen);
```

1.5.10.10 Protocol név <---> protocol-szám fordítás:

File: `/etc/protocols`

Paraméterek átvételéhez:

```
  struct protent {             /* definíciója <netdb.h> -ban */
     char  *  p_name;          /* hivatalos protocol-név */
     char  ** p_aliases;       /* protocol-név alias lista, végén 0
                                  pointer */
     int      p_proto;         /* protocol szám */
  };
```

1.5.10.11 Lekérdezések: mint . . . `host` . . . fv.-eknél:

```
  struct protent * getprotent (void);
  void endprotent (void);
  struct protent * getprotbynumber (int protno);
  struct protent * getprotbyname (char *name);
  void setprotent (int stayopen);
```

1.5.10.12 Szolgáltatás név <---> port-szám fordítás:

File: **`/etc/services`**
Paraméterek átvételéhez:

```
struct servent {            /* definíciója <netdb.h> -ban */
   char  *  s_name;         /* hivatalos szolgáltatás-név */
   char  ** s_aliases;      /* szolg.-név alias lista, végén 0
                               pointer */
   int      s_port;         /* port szám */
   char   * s_proto;        /* használandó protokoll, ha NULL, csak
                               szolgáltatás-nevet használ */
};
```

1.5.10.13 Lekérdezések: mint . . . **`host`** . . . fv.-eknél:

```
struct protent * getservent (void);
void endservent (void);
struct servent * getservbyport (int port,  char *proto);
struct servent * getservbyname (char *name,  char *proto);
void setservent (int stayopen);
```

ahol **`proto`**: protokoll névre mutat

Pl. a helyi hálózaton lévő gépek nevének, címének lekérdezése:

```
typedef struct in_addr {                      /* Internet address */
  union {
    struct { uchar_t s_b1, s_b2, s_b3, s_b4; } S_un_b;  /* bytes */
    struct { ushort_t s_w1, s_w2; } S_un_w;             /* 2-B words
                                  */
    ulong_t S_addr;                           /* 4-B word */
  } S_un;                                     /* union név */
} in_addr_t;                                  /* típusnév */

typedef struct hostelement {                  /* Lánc egy eleme */
  struct hostelement * next;                  /* => következ• */
  char               * name;                  /* host neve */
  in_addr_t            inaddr;                /* internet cím */
}  hostelement_t;                             /* típusnév */

struct utsname {   /* host rendszer adatai    [26] 382. old. */
  char sysname  [SYS_NMLN];                   /* rendszer neve */
  char nodename [SYS_NMLN];                   /* host neve */
  char release  [SYS_NMLN];                   /* op. r. neve */
  char version  [SYS_NMLN];                   /* verzió */
  char machine  [SYS_NMLN];                   /* géptípus */
};


hostelement_t * gethosts (void)  /* Host-ok adatainak kigy•jtése */
{
   struct hostent *hp;          /* => egy bejegyzés a host listában */
   hostelement_t  *rhp, *nhp;   /* => lánc eleje, új láncelem */
   struct utsname  u;           /* saját név megszerzéséhez */
   in_addr_t      *ip;          /* => internet address */
   rhp=0;                       /* üres lánc */
```

```
    if (uname(&u) < 0) error("uname() hiba");  /* saját adatok lekérd.
                                */
    while ((hp=gethostent()) !=0)  {  /* következ• host olvasása */
      if ( !strcmp(hp->h_name, "localhost") ||  /* saját maga,
                                          visszahurkoláshoz */
           !strcmp(hp->h_name, "anyhost") || /* broadcast üzenethez */
           !strcmp(hp->h_name, u.nodename))  /* saját maga */
         continue;
      if ( ! (nhp = (hostelement_t*) malloc (sizeof(hostelement_t))) )
        error ("Nincs elég memória");
      ip = (in_addr_t*) *(hp->h_addr_list);  /* 1. Internet cím címe */
      nhp->inaddr.s_addr = ip->S_addr;       /* In. cím a láncelembe */
      nhp->name = strdup(hp->h_name);        /* host név másol. */
      nhp->next = rhp;  rhp = nhp;           /* beláncolás el•re */
    } /* while */

    endhostent();                            /* olvasás lezárása */

    return rhp;
  }
```

1.6 I/O MULTIPLEXELÉS

A file illetve socket írási-olvasási kérések együtt multiplexelhetőek szinkron módon a `select()` rendszerhívással, amelynek formátuma a következő:

```
      #include <sys/time.h>
      #include <sys/select.h>
      int select(int nfds, fd_set *readfds, fd_set    *writefds,fd_set
*exceptfds, struct timeval *timeout);
```

A függvény `nfds` argumentum az utána következő három argumentumban előforduló file leírók maximális száma. A második, harmadik és negyedik argument három fileleíró halmazra mutat: az elsőbe azok a fileleírók vannak beállítva, amelyekre olvasási feltétel bekövetkeztét várjuk, a másodikba az írási, a harmadikba a kivételes műveletre váró fileleírók vannak bemaszkolva. Kivételes műveletek alatt Out-of-Band adatok érkezését, vagy signal érkezését kell érteni, erről az Out-of-Band című alfejezetben lesz szó. Akármelyik maszk lehet `NULL` is. A maszkok `long` integer értékek, bennük a file leírók bitek. Ha az adott gépen a long integer 32 bites, akkor maximum 32 file leírót tudunk a `select()` hívással szinkron módon multiplexelni. Ennek kikerlésére több trükkös megoldás létezik.
A file leírók maszkolására az alábbi makrók szolgálnak:

`void FD_SET(int fd, fd_set &fdset);` a fileleíró beállítása a maszkban
`void FD_CLR(int fd, fd_set &fdset);` a fileleíró törlésre a maszkból
`int  FD_ISSET(int fd, fd_set &fdset);` a fileleíró szerepel-e a maszkban
`void FD_ZERO(fd_set &fdset);` az egész maszk törlése, inicializálása

A `timeout` argumentum a hívásban az eseményekre való maximális várakozás idejét hivatott beállítani. Ha nincs definiálva (NULL), akkor a `select()` blokkolódik bármelyik beállított fileleírón egy esemény bekövetkeztéig, vagy egy SIGIO illetve SIGURG signal érkezéséig. A SIGIO és SIGURG szignálok az aszinkron multiplexelést hivatottak biztosítani, erről majd az Aszinkron socket-ek című fejezetben lesz szó. Ha a `timeout` 0, akkor a `select()` lekérdezi a beállított fileleírókra érkezett eseményeket és azonnal visszatér.

1.7 OUT-OF-BAND ADATOK

TCP használata esetén lehetőség van két processz között a kétirányú (duplex) csatorna mellet egy logikailag külön kezelt "jelzési csatornát" használni, úgynevezett Out-of-Band (OOB) adatokat küldeni. Az OOB adatok a többi adattól függetlenül küldhetőek, `send(3N)` rendszer-hívás harmadik, `flags` argumentumának MSG_OOB flag beállításával. Ezt a jelzési csatornát használja a telnet és az rlogin vezérlő adatok továbbítására.

1.8 NEM-BLOKKOLÓDÓ SOCKET-EK

Néhány alkalmazásnál követelmény, hogy a socket-ekre vonatkozó rendszerhívások ne blokkolódjanak. Egy olyan kérés, amely nem tud végrahajtódni azonnal, mert várakozni kell a processznek a kérés végrehajtására, nem mindig alkalmazható real-time körülmények között. Egy már létrehozott és összekötött socket-et az `fcntl(2)` rendszerhívással tehetünk nem-blokkolódóvá.
Az alábbi példa e hívás használatát mutatja be:

```
#include <fcntl.h>
#include <sys/file.h>
int   fileflags;
int   sock;
...
      sock= socket(AF_INET, SOCK_STREAM, 0);
...
      if(fileflags= fcntl(sock, F_GETFL, 0)== -1)
      {
            perror(“Get file flags“);
            exit(-1);
      }
      if(fcntl(sock, F_SETFL, fileflags | FNDELAY)== -1)
      {
```

```
            perror("Set file to non-nlocking");
            exit(-1);
      }
...
```

Ha egy nem-blokkolódó socket-en I/O történik, ellenőrizni kell az `errno` hibaváltozó tartalmát, `EWOULDBLOCK` hiba keletkezhet egy olyan hívásnál, amelyik rendesen (ezen opció beállítása nélkül) blokkolódna. Az `accept(3N)`, `connect(3N)`, `send(3N)`, `recv(3N)`, `read(2)` és `write(2)` hívásoknál fordulhat elő. Ha például a `write()` hívás nem tud teljesen befejeződni (nem tudja az adott socketre kiírni az összes byte-ot), a visszatérési érték a sikeresen elküldött byte-ok száma lesz. A nem blokkolódó tulajdonságot bármikor be lehet állítani és el lehet venni.

1.9 INTERRUPT/ESEMÉNY VEZÉRELT SOCKET-EK (ASZINKRON SOCKETEK)

Néhány esetben szükség lehet eseményvezérlés jellegű hálózati kommunikációra. Erre szolgálnak az aszinkron socket-et. A processz ilyenkor **`SIGIO`** szignált kap, ha a socket (vagy file) befejezte az adott I/O műveletet. E szignál elkapásához és feldolgozásához három dolgot kell megtenni:

1. Meg kell adni, hogy a processzben a `SIGIO` szignált melyik függvény fogja lekezelni. Ezt a `signal(2)`, `sigset(2)`, (`sigvec(2B)` a *BSD* kompatibilis *UNIX* rendszerek esetén) rendszerhívásokkal tehetjük meg.
2. Hozzá kell rendelni az adott socket-hez a **processz ID**, vagy processz csoport ID az `fcntl(2)` hívással.
3. Át kell állítani a socket-et **aszinkron** üzemmódba.

A következő példa egy `sock` socket-re állítja be, hogy a vétel pillanatában az alkalmazás a `my_handler()` függvénnyel kezelje a beérkező `SIGIO` szignált:

```
#include <signal.h>
#include <fcntl.h>
#include <sys/file.h>
...
      signal(SIGIO, my_handler);
      sigset(SIGIO, my_handler);
      if(fcntl(sock, F_SETOWN, getpid())< 0)
      {
            perror("Set file to asynchronous");
            exit(-1);
      }
```

Out-of-Band adatok aszinkron kezeléséhez hasonlóan be kell állítani a `SIGURG` szignál fogadását is programunkban.

A kövtekző példa socket-et aszinkron üzemmódba állítását mutatja be:

```
#include <fcntl.h>
#include <sys/file.h>
int   fileflags;
int   sock;
...
      sock= socket(AF_INET, SOCK_STREAM, 0);
...
      if(fileflags= fcntl(sock, F_GETFL, 0)== -1)
      {
            perror("Get file flags");
            exit(-1);
      }
      if(fcntl(sock, F_SETFL, fileflags | FNDELAY | FASYNC)== -1)
      {
            perror("Set file to non-blocking and asynchronous");
            exit(-1);
      }
```

Az aszinron üzemmódba helyezés után a `send(3N)`, `recv(3N)`, `read(2)` és `write(2)` hívásokat használhatjuk, ha beállítottuk a `SIGIO` szignál lekezelését programunkban.

1.10 A HÁLÓZATI KONFIGURÁCIÓ MEGHATÁROZÁSA

A *UNIX* `ioctl(2)` rendszerhívása `SIOCGIFCONF`, `SIOCGIFBRDADDR` és `SIOCGIFFLAGS` beállításai lehetőséget nyújtanak az interfészek paramétereinek lekérdezésre,az összes kapcsolódó hálózat broadcast címének lekérdezésére, ezzel kideríthetjük, támogatja-e az adott hálózati interfész a broadcast üzenetek küldését, vagy sem, ha igen milyen broadcast címre küldhetünk broadcast üzeneteket a **`sendto(3N)`** rendszerhívás segítségével.

1.11 BROADCAST ÜZENETEK KÜLDÉSE

Internet domainű datagram típusú socket-ek broadcast és multicast üzeneteket is küldhetnek, hogy a kapcsolódó hálózat összes host-ját vagy a gépek egy csoportját elérjék. A broadcast üzenetek igen leterhelhetik a hálózatot, hiszen minden hálózatban levő gépet az üzenetek feldolgozására kényszerítenek.
Ha egy host több hálózatra is csatlakozik, akkor a **`send(3N)`** csak az egyik hálózatra tud broadcast üzeneteket kibocsátani, kivéve, ha a speciális `INADDR_BROADCAST` címhez rendeljük hozzá a socket-et (`bind(3N)`). Az `INADDR_ANY` és az

INADDR_BROADCAST konstansok a netinet/in.h header file-ban vannak definiálva.

1.12 MULTICAST ÜZENETEK KÜLDÉSE ÉS FOGADÁSA

(megírandó)

2 példák:

2.1 UNICAST PÉLDAPROGRAMOK

2.1.1 Datagram Szerver példa:

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

/*
 * This program creates a datagram socket, binds a name to it, then reads
 * from the socket.
 */
main()
{
     int sock, length;
     struct sockaddr_in name;
     char buf[1024];

     /* Create socket from which to read. */
     sock = socket(AF_INET, SOCK_DGRAM, 0);
     if (sock < 0) {
          perror("opening datagram socket");
          exit(1);
```

```
     }
     /* Create name with wildcards. */
     name.sin_family = AF_INET;
     name.sin_addr.s_addr = INADDR_ANY;
     name.sin_port = 0;
     if (bind(sock, &name, sizeof(name))) {
          perror("binding datagram socket");
          exit(1);
     }
     /* Find assigned port value and print it out. */
     length = sizeof(name);
     if (getsockname(sock, &name, &length)) {
          perror("getting socket name");
          exit(1);
     }
     printf("Socket has port #%d\n",                    ntohs(name.sin_port));
     /* Read from the socket */
     if (read(sock, buf, 1024) < 0)
          perror("receiving datagram packet");
     printf("-->%s\n", buf);
     close(sock);
}
```

2.1.2 Datagram kliens példa

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

#define DATA "The sea is calm tonight, the tide is full . . ."

/*
 * Here I send a datagram to a receiver whose name I get from the command
 * line arguments.  The form of the command line is
 * *     dgr_cli hostname portnumber message
 */

main(argc, argv)
     int argc;
     char *argv[];
{
```

```
    int sock;
    struct sockaddr_in name;
    struct hostent *hp, *gethostbyname();

    /* Create socket on which to send. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
         perror("opening datagram socket");
         exit(1);
    }
    /*
     * Construct name, with no wildcards, of the socket to send to.
     * Gethostbyname() returns a structure including the network address
     * of the specified host.  The port number is taken from the command
     * line.
     */
    hp = gethostbyname(argv[1]);
    if (hp == 0) {
         fprintf(stderr, "%s: unknown host\n", argv[1]);
         exit(2);
    }
    bcopy(hp->h_addr, &name.sin_addr, hp->h_length);
        /* Get IP address BSD 4.2 way, indeed we receive a pointer to pointer */
    name.sin_family = AF_INET;
    name.sin_port = htons(atoi(argv[2]));
    /* Send message. */
    if (sendto(sock, argv[3], strlen(argv[3]), 0, &name, sizeof(name)) < 0)
         perror("sending datagram message");
    close(sock);
}
```

2.1.3 Stream szerver példa

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

/*
 * This program creates a socket and then begins an infinite loop. Each time
 * through the loop it accepts a connection and prints out messages from it.
```

```
 * When the connection breaks, or a termination message comes through, the
 * program accepts a new connection.
 */

main()
{
     int sock, length;
     struct sockaddr_in server;
     int msgsock;
     char buf[1024];
     int rval;
     int i;

     /* Create socket */
     sock = socket(AF_INET, SOCK_STREAM, 0);
     if (sock < 0) {
          perror("opening stream socket");
          exit(1);
     }
     /* Name socket using wildcards */
     server.sin_family = AF_INET;
     server.sin_addr.s_addr = INADDR_ANY;
     server.sin_port = 0;
     if (bind(sock, &server, sizeof(server))) {
          perror("binding stream socket");
          exit(1);
     }
     /* Find out assigned port number and print it out */
     length = sizeof(server);
     if (getsockname(sock, &server, &length)) {
          perror("getting socket name");
          exit(1);
     }
     printf("Socket has port #%d\n", ntohs(server.sin_port));

     /* Start accepting connections */
     listen(sock, 5);
     do {
          msgsock = accept(sock, 0, 0);
          if (msgsock == -1)
               perror("accept");
          else do {
               bzero(buf, sizeof(buf));
```

```
                if ((rval = read(msgsock, buf, 1024)) < 0)
                     perror("reading stream message");
                i = 0;
                if (rval == 0)
                     printf("Ending connection\n");
                else
                     printf("-->%s\n", buf);
           } while (rval != 0);
           close(msgsock);
     } while (TRUE);
     /*
      * Since this program has an infinite loop, the socket "sock" is
      * never explicitly closed.  However, all sockets will be closed
      * automatically when a process is killed or terminates normally.
      */
}
```

2.1.4 Stream kliens példa

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

/*
 * This program creates a socket and initiates a connection with the socket
 * given in the command line.  One message is sent over the connection and
 * then the socket is closed, ending the connection. The form of the command
 * line is streamwrite hostname portnumber
 */

main(argc, argv)
     int argc;
     char *argv[];
{
     int sock;
     struct sockaddr_in server;
     struct hostent *hp, *gethostbyname();
     char buf[1024];

     /* Create socket */
     sock = socket(AF_INET, SOCK_STREAM, 0);
```

```
        if (sock < 0) {
             perror("opening stream socket");
             exit(1);
        }
        /* Connect socket using name specified by command line. */
        server.sin_family = AF_INET;
        hp = gethostbyname(argv[1]);
        if (hp == 0) {
             fprintf(stderr, "%s: unknown host\n", argv[1]);
             exit(2);
        }
        bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
            /* Get IP address BSD 4.2 way, indeed we receive a pointer to pointer */
        server.sin_port = htons(atoi(argv[2]));

        if (connect(sock, &server, sizeof(server)) < 0) {
             perror("connecting stream socket");
             exit(1);
        }
        if (write(sock, argv[3], strlen(argv[3])) < 0)
             perror("writing on stream socket");
        close(sock);
   }
```

2.2 MULTICAST PÉLDA PROGRAMOK

2.2.1 Küldő Program

```
/*
 * sender.c -- multicasts "hello, world!" to a multicast group once a
second
 *
  */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <string.h>
```

```
#include <stdio.h>


#define HELLO_PORT 6789
#define HELLO_GROUP "225.0.0.111"
main(int argc, char *argv[])
{
     struct sockaddr_in addr;
     int fd, cnt;
     struct ip_mreq mreq;
     char *message="Hello, World!";

     /* create what looks like an ordinary UDP socket */
     if ((fd=socket(AF_INET,SOCK_DGRAM,0)) < 0) {
          perror("socket");
          exit(1);
     }

     /* set up destination address */
     memset(&addr,0,sizeof(addr));
     addr.sin_family=AF_INET;
     addr.sin_addr.s_addr=inet_addr(HELLO_GROUP);
     addr.sin_port=htons(HELLO_PORT);

     /* now just sendto() our destination! */
     while (1) {
          if (sendto(fd,message,sizeof(message),0,(struct sockaddr *)
&addr,
                     sizeof(addr)) < 0) {
               perror("sendto");
               exit(1);
          }
          sleep(1);
     }
}
```

2.2.2 Hallgató Program

```
/*
 * listener.c -- joins a multicast group and echoes all data it receives
from
 *              the group to its stdout...
 *
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>
#include <string.h>
#include <stdio.h>


#define HELLO_PORT 6789
```

```
#define HELLO_GROUP "225.0.0.111"
#define MSGBUFSIZE 256

main(int argc, char *argv[])
{
     struct sockaddr_in addr;
     int fd, nbytes,addrlen;
     struct ip_mreq mreq;
     char msgbuf[MSGBUFSIZE];

     /* create what looks like an ordinary UDP socket */
     if ((fd=socket(AF_INET,SOCK_DGRAM,0)) < 0) {
          perror("socket");
          exit(1);
     }

     /* set up destination address */
     memset(&addr,0,sizeof(addr));
     addr.sin_family=AF_INET;
     addr.sin_addr.s_addr=htonl(INADDR_ANY); /* N.B.: differs from
sender */
     addr.sin_port=htons(HELLO_PORT);

     /* bind to receive address */
     if (bind(fd,(struct sockaddr *) &addr,sizeof(addr)) < 0) {
          perror("bind");
          exit(1);
     }

     /* use setsockopt() to request that the kernel join a multicast
group */
     mreq.imr_multiaddr.s_addr=inet_addr(HELLO_GROUP);
     mreq.imr_interface.s_addr=htonl(INADDR_ANY);
     if (setsockopt(fd,IPPROTO_IP,IP_ADD_MEMBERSHIP,&mreq,sizeof(mreq))
< 0) {
          perror("setsockopt");
          exit(1);
     }

     /* now just enter a read-print loop */
     while (1) {
          addrlen=sizeof(addr);
          if ((nbytes=recvfrom(fd,msgbuf,MSGBUFSIZE,0,
                               (struct sockaddr *) &addr,&addrlen)) < 0)
{
               perror("recvfrom");
               exit(1);
          }
          puts(message);
     }
}
```