Számítógépes Hálózatok
2008

2. Hálózati felhasználások -- socket programozás

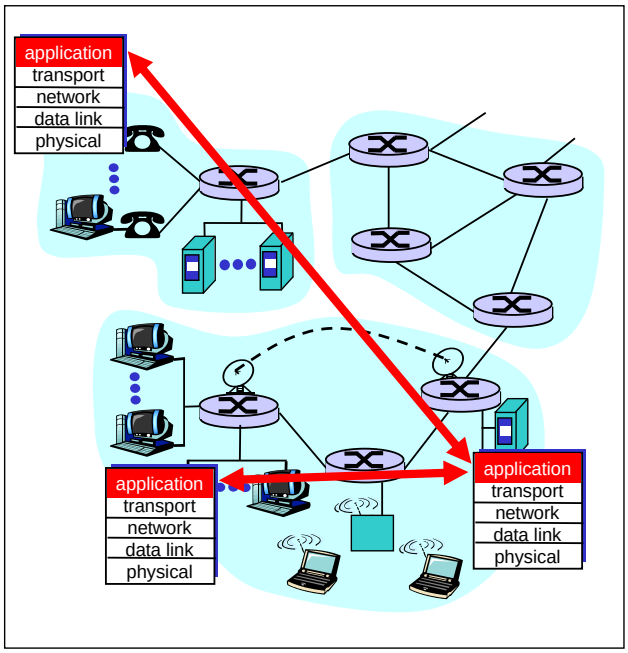
Bruce Maggs és Srinivasan Seshan (CMU)
fóliái alapján

Áttekintés

- Felhasználói réteg
  - Kliens-szerver
  - Felhasználás igényei
- Háttér
  - TCP vs. UDP
  - Byte sorrend
- Socket I/O
  - TCP/UDP szerver és kliens
  - I/O multiplexing

Felhasználások és a felhasználói réteg protokolljai

- Felhasználások (hálózati) : kommunikáló elosztott processzek
  - A hálózat végrendszerein (host) futnak (ugyanazon vagy különböző végrendszeren)
  - Üzeneteket cserélnek ki
  - Pl. email, file transfer, Web
- Felhasználói réteg protokolljai
  - Definiálják az üzeneteket, melyeket a felhasználások kicserélnek és az akciókat, amiket akkor végrehajtanak
  - A kommunikáció megvalósítása alacsonyabb rétegek protokolljai által történik



Kliens-szerver paradigma

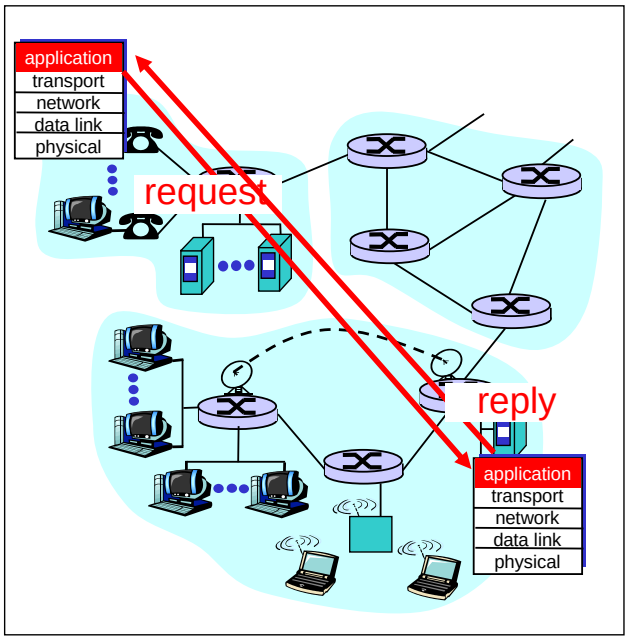
Tipikus hálózati felhasználásnak két része van: **kliens** és **szerver**

Kliens:

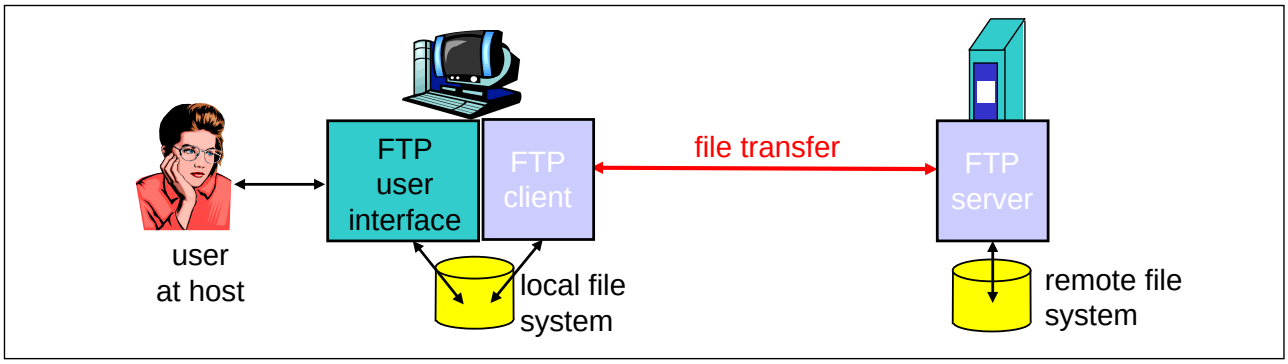
- Kezdeményezi a kapcsolatot a szerverrel
- Tipikusan egy szolgáltatást igényel a szervertől,
- Web esetén a kliens a böngészőben implementált; e-mail esetén a mail olvasó programban

Szerver:

- Az igényelt szolgáltatást bocsátja rendelkezésre a kliens számára
- pl. a web-szerver elküldi a kért web-oldalt; a mail-szerver az e-mailt



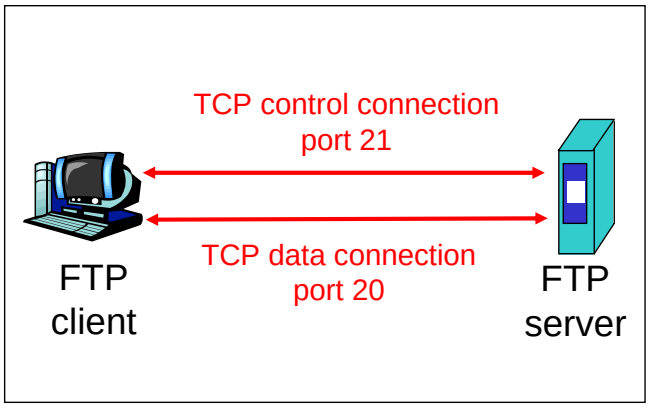
FTP: File Transfer Protocol



- Távol lévő végrendszertől/végrendszerre szállít file-t
- Kliens/szerver modell
  - Kliens: az az oldal, amely a file transzfert kezdeményezi
  - Szerver: távoli végrendszer
- ftp: RFC 959
- ftp server: port 21

FTP: Elkülönített kontroll- és adatkapcsolat

- Ftp-kliens a 21-es porton lép kapcsolatba az ftp-szerverrel és TCP-t adja meg szállítói protokollként
- Két párhuzamos TCP kapcsolat kerül megnyitásra:
  - Kontroll: parancsok és válaszok kicserélésére a kliens és a szerver között “out of band control”
  - Adat: file a szerverhez/szervertől
- Az ftp-szerver státusz-információkat tárol: aktuális könyvtár, korábbi autentifikáció



Ftp parancsok, válaszok

parancs példák:

- A kontroll csatornán küldött ASCII szöveg
- **USER *username***
- **PASS *password***
- **LIST** az aktuális könyvtár file-jainak a listájával tér vissza
- **RETR filename** letölti a file-t (get)
- **STOR filename** tárolja a file-t a távoli végrendszeren (put)

válasz példák

- status code és válasz szövegek
- **331 Username OK, password required**
- **125 data connection already open; transfer starting**
- **425 Can't open data connection**
- **452 Error writing file**

Milyen szolgáltatásokra van a felhasználásoknak szüksége a szállítói rétegtől?

Adat vesztés

- Néhány felhasználás eltűr valamennyi adatvesztést (pl. audio)
- Más felhasználások (pl. file transfer, telnet) 100% megbízható adatátvitelt igényelnek

Időzítés

- Néhány felhasználás (pl. Internet telefon, interaktív játékok) rövid késést (delay) igényelnek

Sávszélesség

- Néhány felhasználás (pl. multimedia) igényel egy minimálisan rendelkezésre álló sávszélességet
- Más felhasználások ("elastic apps") azt a sávszélességet használják amit éppen kapnak

Gyakori felhasználások igényei a szállítói réteg szolgáltatásaira

Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
web documents	no loss	elastic	no
real-time audio/ video	loss-tolerant	audio: 5Kb-1Mb video:10Kb-5Mb	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few Kbps	yes, 100's msec
financial apps	no loss	elastic	yes and no

Áttekintés

- Felhasználói réteg
  - Kliens-szerver
  - Felhasználás igényei
- Háttér
  - TCP vs. UDP
  - Byte sorrend
- Socket I/O
  - TCP/UDP szerver és kliens
  - I/O multiplexing

Szerver és Kliens

- Szerver és kliens a hálózaton üzeneteket cserélnek ki egymással a közös **socket API** által
- Socket-ek által a hálózati I/O hasonló a file I/O-hoz
- Rendszerfüggvény hívása a kontrollhoz és a kommunikációhoz
- A hálózat kezeli a routingot, szegmentálást, stb…

User Datagram Protocol(UDP)

UDP

- Egyszerű socket üzenetek küldésére/fogadására
- Nincs garancia a megérkezésre
- Nem szükségszerűen sorrendtartó
- Datagram – független csomagok
- Minden csomagot meg kell címezni
- Analógia: postai levél…

UDP

Példa UDP felhasználásokra:

Multimedia, voice over IP

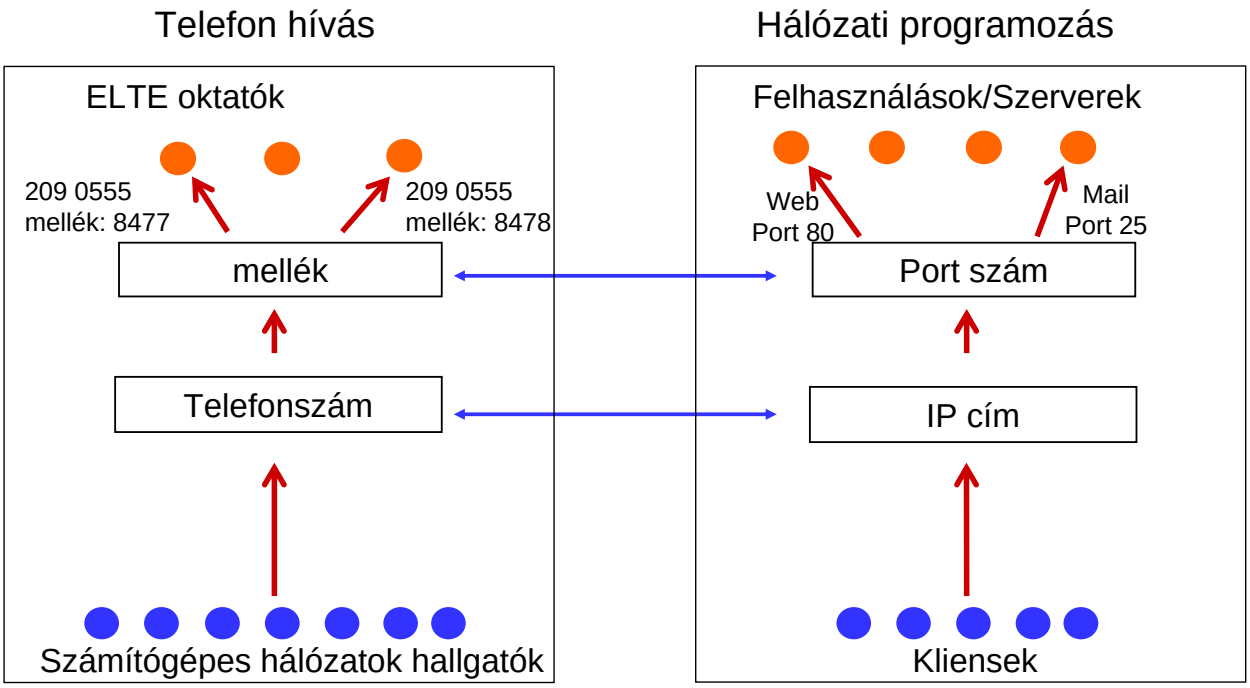
Transmission Control Protocol (TCP)

TCP

- Megbízható – megérkezés garantált
- Byte folyam – sorrendtartó
- Kapcsolat-orientált – egy socket kapcsolatonként
- A kapcsolat felépítése után adatátvitel
- Analógia: telefon hívás

Példa TCP felhasználásokra:
Web, Email, Telnet

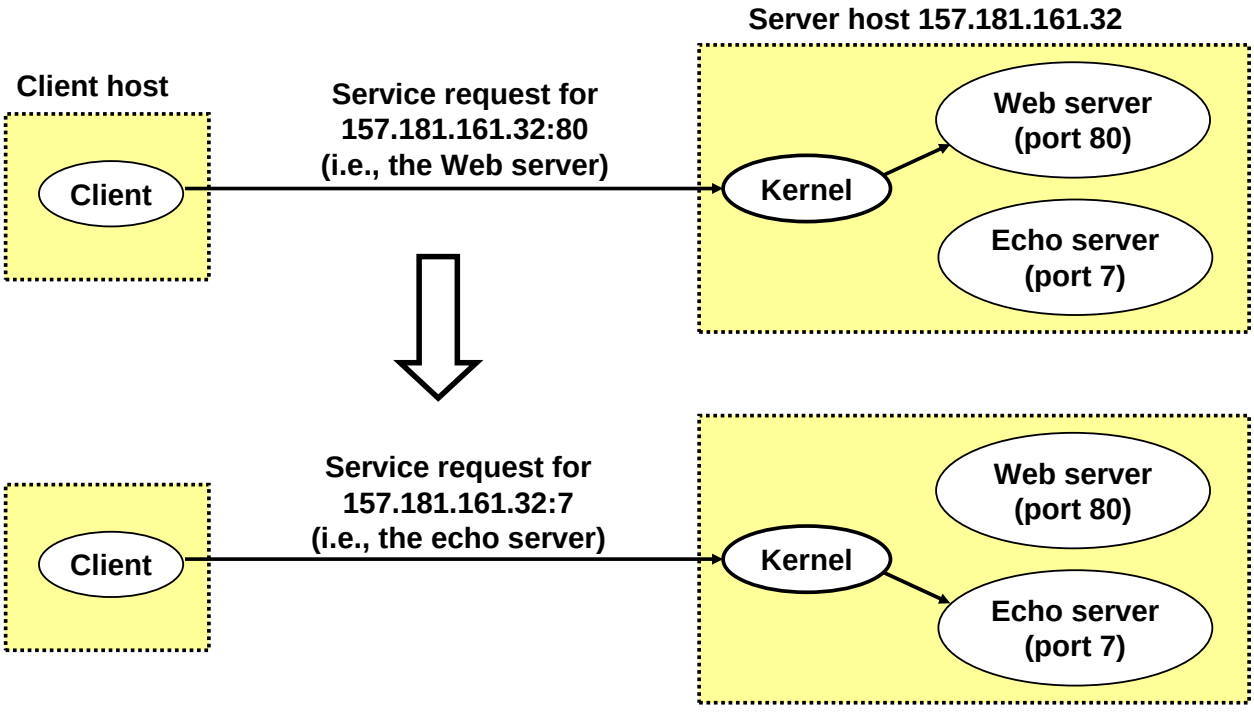
Hálózat címzési analógia



Hálózati címzés

- Az IP cím a socket-ben azonosítja a végrendszert *(pontosabban egy adaptert a végrendszerben)*
- A (jól ismert) port a szerver socket-ben azonosítja a szolgáltatást, ezáltal implicit azonosítja a végrendszerben a processzt, ami végrehajtja a szolgáltatatást
- Port szám lehet
  - „Jól ismert” port (well-known port) (port 0-1023)
  - Dinamikus vagy privát port (port 1024-65535)
- Szerverek/daemonok általában a jól ismert portokat használják
  - Minden kliens azonosíthatja a szervert/szolgáltatást
  - HTTP = 80, FTP controll = 21, Telnet = 23, mail = 25,...
  - /etc/services tartalmazza a jól ismert portok listáját Linux rendszerben
- Kliensek általában dinamikus portokat használnak
  - A kernel által futási időben hozzárendelt

Port, mint szolgáltatás azonosítója



Nevek és címek

- Az Interneten minden kapcsolódási pontnak van egy egyértelmű címe
  - amely az elhelyezkedésen alapul – a telefonszámokhoz hasonlóan
- Az ember jobban tud neveket kezelni mint címeket
  - pl. www.inf.elte.hu
  - DNS (domain name system) nevek címekre való leképezését bocsátja rendelkezésre
  - A név a végrendszer adminisztrációs hovatartozásán alapul

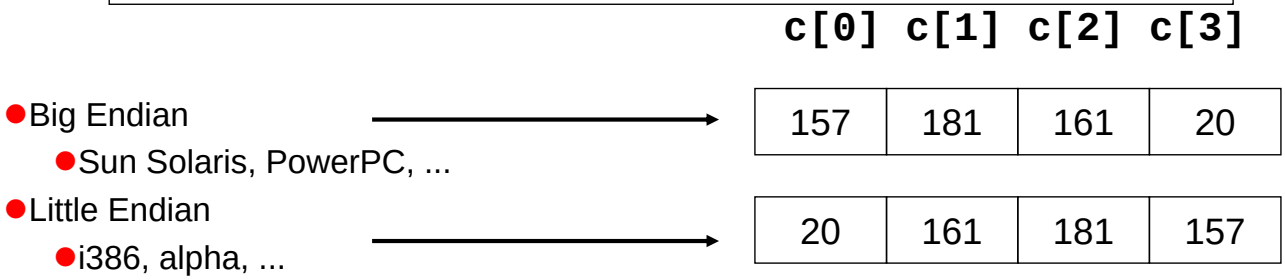
Internet címzési adatstruktúra

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

/* Internet address structure */
struct in_addr {
        u_long s_addr;          /* 32-bit IPv4 address */
};                              /* network byte ordered */

/* Socket address, Internet style. */
struct sockaddr_in {
      u_char  sin_family;       /* Address Family */
      u_short sin_port;         /* UDP or TCP Port# */
                                /* network byte ordered */
      struct in_addr sin_addr; /* Internet Address */
      char    sin_zero[8];      /* unused */
};
```

- `sin_family = AF_INET  /*selects Internet address family*/`

Byte sorrend

```
union {
   u_int32_t addr;  /* 4 bytes address */
   char c[4];
} un;
/* 157.181.161.32 */
un.addr = 0x9db5a120;
/* c[0] = ? */
```



Byte sorrend függvények

- **host byte order** és **network byte order** közötti konvertálás
  - 'h' = host byte order
  - 'n' = network byte order
  - 'l' = long (4 bytes), IP címet konvertál
  - 's' = short (2 bytes), port számot konvertál

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

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

Áttekintés

- Felhasználói réteg
  - Kliens-szerver
  - Felhasználás igényei
- Háttér
  - TCP vs. UDP
  - Byte sorrend
- Socket I/O
  - TCP/UDP szerver és kliens
  - I/O multiplexing

Socket

- Egy **socket** egy file leíró, amin keresztül a felhasználás a hálózatba ír / hálózatból olvas

```
int fd;         /* socket descriptor */
if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) }
      perror("socket");
      exit(1);
}
```

- `socket` egy egész számot ad vissza (socket descriptor: `fd`)
  - **`fd`** < 0 jelzi, hogy hiba lépett fel
  - socket leíró (socket descriptor) hasonló a file leíróhoz, a fő különbség az, ahogy a felhasználás megnyitja a socket leírót
- AF_INET: a socket-et az Internet protokoll családhoz rendeli
- SOCK_STREAM: TCP protokoll
- SOCK_DGRAM: UDP protokoll

TCP Szerver

- Például: web-szerver (port 80)
- Mit kell a web-szervernek tenni, hogy egy web-kliens kapcsolatot létesíthessen vele?

Socket I/O: socket()

- Mivel a web forgalom TCP-t használ, a web-szervernek létre kell hozni egy socket-et SOCK_STREAM tipussal

```
int fd;          /* socket descriptor */

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

- **socket** egy egész számot ad vissza (socket descriptor: **fd**)
  - **fd** < 0 jelzi, hogy hiba lépett fel
- AF_INET: a socket-et az Internet protokoll családhoz rendeli
- SOCK_STREAM: TCP protokoll

Socket I/O: bind()

- Egy socket-et egy port-hoz lehet kötni

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by bind() */

/* create the socket */

srv.sin_family = AF_INET;        /* use the Internet addr family */

srv.sin_port = htons(80);        /* bind socket 'fd' to port 80*/

/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
      perror("bind"); exit(1);
}
```

- Még nem tud kommunikálni a klienssel...

Socket I/O: listen()

- **listen** jelzi, hogy a szerver kapcsolatot akar fogadni

```
int fd;                      /* socket descriptor */
struct sockaddr_in srv;      /* used by bind() */

/* 1) create the socket */
/* 2) bind the socket to a port */

if(listen(fd, 5) < 0) {
      perror("listen");
      exit(1);
}
```

- **listen** második paramétere a queue maximális hossza a függőben lévő kapcsolatkéréseknek (lásd később)
- Még mindig nem tud kommunikálni a klienssel...

Socket I/O: accept()

- **accept** blokkolja (felfüggeszti) a szervert, várakozik a kapcsolatkérésre

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by bind() */
struct sockaddr_in cli;          /* used by accept() */
int newfd;                       /* returned by accept() */
int cli_len = sizeof(cli);       /* used by accept() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
       perror("accept");   exit(1);
}
```

- **accept** egy új socket-et ad vissza (**newfd**) ugyanolyan tulajdonságokkal, mint az eredeti socket (**fd**)
  - **newfd** < 0 jelzi, ha hiba történt

Socket I/O: accept() folytatás...

```
struct sockaddr_in cli;        /* used by accept() */
int newfd;                     /* returned by accept() */
int cli_len = sizeof(cli);     /* used by accept() */

newfd = accept(fd, (struct sockaddr*) &cli, &cli_len);
if(newfd < 0) {
      perror("accept");
      exit(1);
}
```

- Honnan tudja szerver, melyik kliens kapcsolódik hozzá?
  - **cli.sin_addr.s_addr** tartalmazza a kliens IP címét
  - **cli.sin_port** tartalmazza a kliens port számát
- Mostmár a szerver adatokat tud kicserélni a klienssel **read** és **write** funkciókat használva a **newfd** leírón.
- Miért kell, hogy **accept** egy új leírót adjon vissza?
  (gondoljunk egy szerverre, ami több klienst szimultán szolgál ki)

Socket I/O: read()

- **read** egy socket-tel használható
- **read** blokkol, a szerver az adatokra várakozik a klienstől, de nem garantálja, hogy **sizeof(buf)** byte-ot olvas

```
int fd;                              /* socket descriptor */
char buf[512];                       /* used by read() */
int nbytes;                          /* used by read() */

/* 1) create the socket */
/* 2) bind the socket to a port */
/* 3) listen on the socket */
/* 4) accept the incoming connection */

if((nbytes = read(newfd, buf, sizeof(buf))) < 0) {
       perror("read"); exit(1);
}
```


TCP Kliens

- Példa: web kliens
- Hogy kapcsolódik egy web-kliens a web-szerverhez?

IP címek kezelése

- IP címeket “157.181.161.32” sztringként szokás írni, de a programok az IP címeket egész számként kezelik

Sztringek egész címmé konvertálása:

```
struct sockaddr_in srv;

srv.sin_addr.s_addr = inet_addr(“157.181.161.32”);
if(srv.sin_addr.s_addr == (in_addr_t) -1) {
      fprintf(stderr, "inet_addr failed!\n"); exit(1);
}
```

Egész címek sztriggé konvertálása:

```
struct sockaddr_in srv;
char *t = inet_ntoa(srv.sin_addr);
if(t == 0) {
      fprintf(stderr, “inet_ntoa failed!\n”); exit(1);
}
```


Nevek címre fordítása

- **gethostbyname** DNS-hez bocsát rendelkezésre interfészt
- Egyéb hasznos hívások
  - **gethostbyaddr** – visszatér **hostent**-el, ami az adott **sockaddr_in**-hez tartozik
  - **getservbyname**
    - szogáltatás leírás lekérdezésére szokták használni (tipikusan port szám)
    - visszatérő érték: **servent** a név alapján

```
#include <netdb.h>

struct hostent *hp; /*ptr to host info for remote*/
struct sockaddr_in peeraddr;
char *name = “www.inf.elte.hu”;

peeraddr.sin_family = AF_INET;
hp = gethostbyname(name)
peeraddr.sin_addr.s_addr = ((struct in_addr*)(hp->h_addr))->s_addr;
```

Socket I/O: connect()

- **connect**: a kliens blokkolódik, amíg a kapcsolat létre nem jön
  - Miután folytatódik, a kliens kész üzeneteket kicserélni a szerverrel az **fd** leíróval.

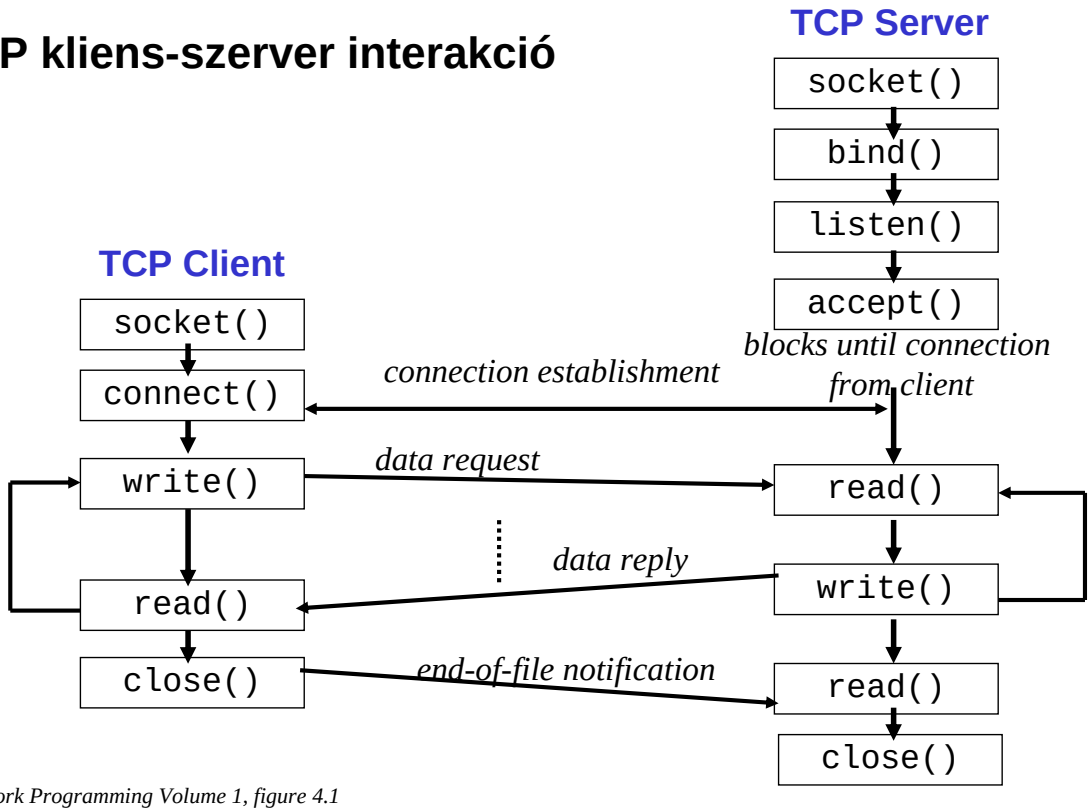
```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by connect() */

/* create the socket */

/* connect: use the Internet address family */
srv.sin_family = AF_INET;

/* connect: socket 'fd' to port 80 */
srv.sin_port = htons(80);

/* connect: connect to IP Address "157.181.161.52" */
srv.sin_addr.s_addr = inet_addr("157.181.161.52");

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


Socket I/O: write()

- **write** egy socket leíróval használható

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by connect() */
char buf[512];                   /* used by write() */
int nbytes;                      /* used by write() */

/* 1) create the socket */
/* 2) connect() to the server */

/* Example: A client could "write" a request to a server */
if((nbytes = write(fd, buf, sizeof(buf))) < 0) {
      perror("write");
      exit(1);
}
```

TCP kliens-szerver interakció



UNIX Network Programming Volume 1, figure 4.1

UDP szerver példa

- Példa: NTP (Network Time Protocol) daemon (port 123)
- Mit kell egy UDP szervernek tenni, hogy egy UDP kliens kapcsolódhasson hozzá?

Socket I/O: socket()

- A UDP szervernek létre kell hozni egy **datagram** socket-et

```
int fd;           /* socket descriptor */

if((fd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
      perror(“socket”);
      exit(1);
}
```

- **socket** egy egészet ad vissza (socket descriptor **fd**)
  - **fd** < 0 jelzi, ha hiba történt

- AF_INET: a socketet az Internet protokoll családdal asszociálja
- **SOCK_DGRAM**: UDP protokoll

Socket I/O: bind()

- Egy socket-et egy port-hoz köthetünk

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;    /* used by bind() */

/* create the socket */

/* bind: use the Internet address family */
srv.sin_family = AF_INET;

/* bind: socket ‘fd’ to port 123*/
srv.sin_port = htons(123);

/* bind: a client may connect to any of my addresses */
srv.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(fd, (struct sockaddr*) &srv, sizeof(srv)) < 0) {
      perror("bind"); exit(1);
}
```

- Ezután már a UDP szerver csomagokat tud fogadni…

Socket I/O: recvfrom()

- **read** nem bocsátja a kliens címét a UDP szerver rendelkezésére

```
int fd;                          /* socket descriptor */
struct sockaddr_in srv;          /* used by bind() */
struct sockaddr_in cli;          /* used by recvfrom() */
char buf[512];                   /* used by recvfrom() */
int cli_len = sizeof(cli);       /* used by recvfrom() */
int nbytes;                      /* used by recvfrom() */

/* 1) create the socket */
/* 2) bind to the socket */

nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,
                  (struct sockaddr*) &cli, &cli_len);
if(nbytes < 0) {
      perror("recvfrom"); exit(1);
}
```


Socket I/O: recvfrom() folytatás...

```
nbytes = recvfrom(fd, buf, sizeof(buf), 0 /* flags */,
                  (struct sockaddr*) cli, &cli_len);
```

- A **recvfrom** által végrehajtott akciók
  - visszaadja az olvasott byte-ok számát (**nbytes**)
  - **nbytes** adatot másol **buf**-ba
  - visszaadja a kliens címét (**cli**)
  - visszaadja **cli** hosszát (**cli_len**)
  - ne törődjünk a flag-ekkel

UDP kliens példa

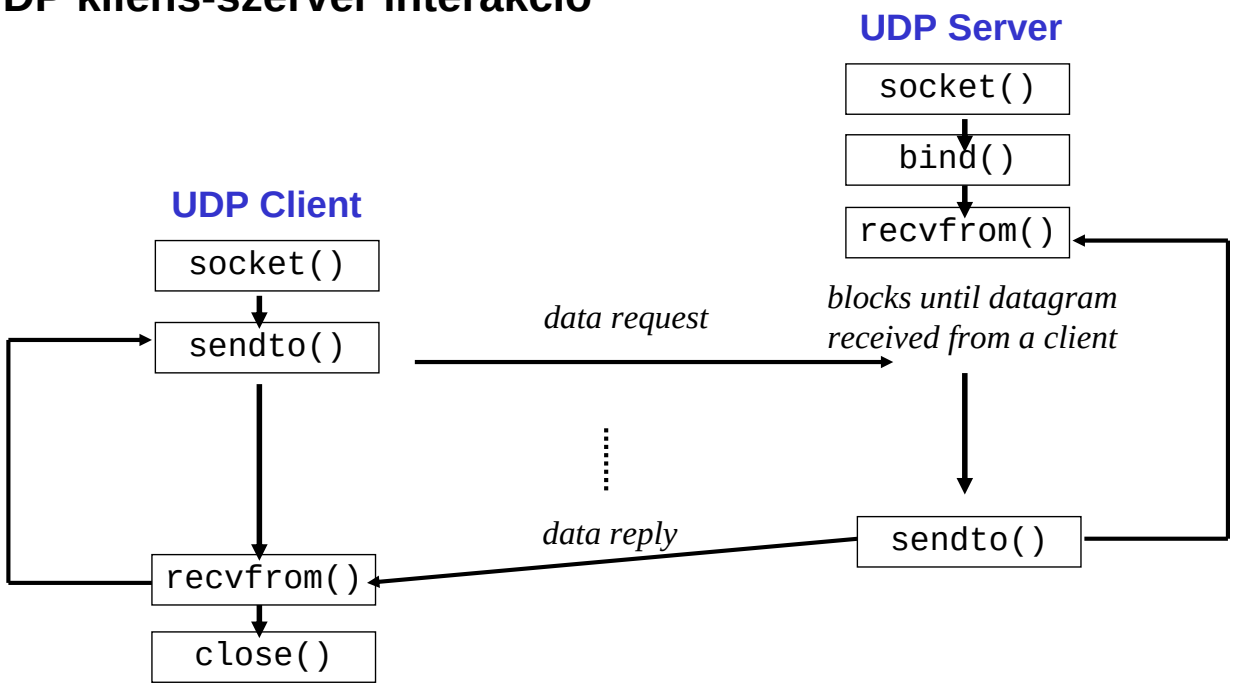
- Mit kell tenni egy UDP kliensnek, hogy kommunikálhasson egy UDP szerverrel?

Socket I/O: sendto()

- **write** nem megengedett!
- Figyeljük meg, hogy a UDP kliensnél nincs port szám kötés (bind)
  - egy port szám **dinamikusan** rendelődik hozzá az első **sendto** hivásakor

```
int fd;                              /* socket descriptor */
struct sockaddr_in srv;              /* used by sendto() */

/* 1) create the socket */

/* sendto: send data to IP Address “157.181.161.32” port 123 */
srv.sin_family = AF_INET;
srv.sin_port = htons(123);
srv.sin_addr.s_addr = inet_addr(“157.181.161.32”);

nbytes = sendto(fd, buf, sizeof(buf), 0 /* flags */,
                 (struct sockaddr*) &srv, sizeof(srv));
if(nbytes < 0) {
       perror(“sendto”);   exit(1);
}
```

UDP kliens-szerver interakció



from UNIX Network Programming Volume 1, figure 8.1

UDP szerver

- Hogy tud a UDP szerver több klienst szimultán kiszolgálni?

UDP Szerver: két port kiszolgálása

```
int s1;                          /* socket descriptor 1 */
int s2;                          /* socket descriptor 2 */

/* 1) create socket s1 */
/* 2) create socket s2 */
/* 3) bind s1 to port 2000 */
/* 4) bind s2 to port 3000 */

while(1) {
      recvfrom(s1, buf, sizeof(buf), ...);
      /* process buf */

      recvfrom(s2, buf, sizeof(buf), ...);
      /* process buf */
}
```

- Milyen probléma ezzel a kóddal?

Socket I/O: select()

- `select` szinkron I/O multiplexálást enged meg

```
int s1, s2;                    /* socket descriptors */
fd_set readfds;                /* used by select() */

/* create and bind s1 and s2 */
while(1) {
       FD_ZERO(&readfds);      /* initialize the fd set */
       FD_SET(s1, &readfds);   /* add s1 to the fd set */
       FD_SET(s2, &readfds);   /* add s2 to the fd set */

       if(select(s2+1, &readfds, 0, 0, 0) < 0) {
              perror("select");
              exit(1);
       }
       if(FD_ISSET(s1, &readfds)) {
              recvfrom(s1, buf, sizeof(buf), ...);
              /* process buf */
       }
       /* do the same for s2 */
}
```

Socket I/O: select()

```
int select(int maxfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

FD_CLR(int fd, fd_set *fds);   /* clear the bit for fd in fds */
FD_ISSET(int fd, fd_set *fds); /* is the bit for fd in fds? */
FD_SET(int fd, fd_set *fds);   /* turn on the bit for fd in fds */
FD_ZERO(fd_set *fds);          /* clear all bits in fds */
```

- **maxfds**: tesztelendő leírók (descriptors) száma
  - (0, 1, ... maxfds-1) leírókat kell tesztelni
- **readfds**: leírók halmaza, melyet figyelünk, hogy érkezik-e adat
  - visszaadja a leírók halmazát, melyek készek az olvasásra (ahol adat van jelen)
  - Ha az input érték **NULL**, ez a feltétel nem érdekel
- **writefds**: leírók halmaza, melyet figyelünk, hogy írható-e
  - visszaadja a leírók halmazát amelyek készek az írásra
- **exceptfds**: leírók halmaza, melyet figyelünk, hogy exception érkezik-e
  - visszaadja a leírók halmazát amelyeken kivétel érkezik

Socket I/O: select()

```
int select(int maxfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);

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

- **timeout**
  - ha **NULL**, várakozzunk addig amíg valamelyik leíró I/O-ra kész
  - különben várakozzunk a **timeout**-ban megadott ideig
    - Ha egyáltalán nem akarunk várni, hozzunk létre egy timeout structure-t, melyben a timer értéke 0
- Több információhoz: man page

Néhány részlet egy web-szerverről

Hogy tud egy web-szerver több kapcsolatot szimultán kezelni?

Socket I/O: select()

```
int fd, n=0;                        /* original socket */
int newfd[10];                      /* new socket descriptors */
while(1) {
       fd_set readfds;
       FD_ZERO(&readfds); FD_SET(fd, &readfds);

       /* Now use FD_SET to initialize other newfd's
          that have already been returned by accept() */

       select(maxfd+1, &readfds, 0, 0, 0);
       if(FD_ISSET(fd, &readfds)) {
              newfd[n++] = accept(fd, ...);
       }
       /* do the following for each descriptor newfd[i], i=0,…,n-1*/
       if(FD_ISSET(newfd[i], &readfds)) {
              read(newfd[i], buf, sizeof(buf));
              /* process data */
       }
}
```

- Ezután a web-szerver képes többb kapcsolatot kezelni...

Socket programozás referenciák

- Man page
  - használat: man <function name>
- Beej's Guide to Network Programming: http://beej.us/guide/bgnet/
- W. R. Stevens: Unix Network Programming : Networking APIs: Sockets and XTI (Volume 1)
  - 2, 3, 4, 6, 8 fejezet