

TCP/IP SOCKET INTERFACE PROGRAMOZÁSA

Számítógép-hálózatok (BMEVIHA215)

Dr. Lencse Gábor
tudományos főmunkatárs
BME Hálózati Rendszerek és Szolgáltatások Tanszék
lencse@hit.bme.hu

2014. április 24.,
Budapest

TCP/IP Socket Interface fogalma

- Socket: kommunikációs végpont
- Socket és fájl formális hasonlósága és különbsége
  - Kezelésük részben hasonló: megnyit, lezár, ír, olvas
  - Feladatuk más:
    - Fájl: adattárolás (helyben)
    - Socket: információcsere egyik végpontja
- Két megfelelő módon egymáshoz rendelt socket segítségével kétirányú kommunikáció valósítható meg
- TCP/IP socket interfész: a TCP/IP protocol stack programozási felülete
- Számos nyelvhez létezik: pl. C, C++, java, perl, stb.
- A C nyelvét fogjuk megismerni, elsősorban Linux alatt.

A legfontosabb lépések

- Socket létrehozása: **`socket`**
- Helyi IP-cím és portszám megadása: **`bind`**
- Kapcsolat orientált esetben
  - a kapcsolat létrehozása: **`listen, accept, connect`**
  - kommunikáció: **`write`** és **`read`** vagy **`send`** és **`recv`**
- Kapcsolatmentes esetben
  - kommunikáció: **`sendto`** és **`recvfrom`**
- Socket lezárása: **`close`**

Socket létrehozása

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

- **`domain`**: protokollcsalád (lásd következő fólia)
- **`type`**: kommunikáció jellege, például:
  - **`SOCK_STREAM`**: megbízható kétirányú kapcsolat, megvalósítás: TCP-vel
  - **`SOCK_DGRAM`**: datagram szolgálat (kapcsolatmentes, nem megbízható), megvalósítás: UDP-vel
  - **`SOCK_RAW`**: hozzáférés a hálózatai szintű protokollhoz
- **`type`**: milyen protokoll valósítja meg, tipikusan csak 1 van
- **`return`**: socket descriptor: socketet azonosító egész szám (mint file descriptor), hiba esetén: -1

Példák protokollcsaládra

- **`AF_INET`**: IPv4 protokollcsalád
- **`AF_INET6`**: IPv6 protokollcsalád
- **`AF_UNIX, AF_LOCAL`**: helyi kommunikáció

Nevezéktan: *Protocol Family* más néven *Address Family*

- Régebbi BSD rendszerekben **`PF_`** előtagok
- Modern BSD és Linux rendszerekben **`AF_`** előtagok
- Érdekesség: **`/usr/include/bits/socket.h:`**

```
#define AF_INET PF_INET
```

Helyi IP-cím és portszám megadása

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

- **`sockfd`**: socket descriptor
- **`my_addr`**: mutató egy **`sockaddr`** típusú címstruktúrára, ami a beállítani kívánt helyi IP-címet és portszámot tartalmazza. Típusa **`AF_INET`** esetén **`sockaddr_in`**, **`AF_INET6`** esetén pedig **`sockaddr_in6`**. (Ld. köv. fóliák) A függvény hívásakor type cast: **`(struct sockaddr *)`**
- **`addrlen`**: megadja a címstruktúra méretét bájtokban
- **`return`**: hiba esetén -1, egyébként 0

IPv4: struct sockaddr_in

```
struct sockaddr_in {
  sa_family_t sin_family; /*address family: AF_INET*/
  u_int16_t  sin_port; /*port in network byte order*/
  struct in_addr sin_addr;  /* IPv4 address */
/* Pad to size of 'struct sockaddr'. */
};
/* IPv4 address: */
struct in_addr {
  u_int32_t s_addr; /*address in network byte order*/
};
```

Megjegyzés: hálózati bájtsorrend: MSB: Most Significant Bit first, big-endian. A legnagyobb helyi értékű bájt van elöl.

De a számítógép bájtsorrendje eltérő lehet! Vannak konverziós függvények, lásd később.

IPv6: struct sockaddr_in6

```
struct sockaddr_in6 {
  u_int16_t sin6_family; /* address family: AF_INET6 */
  u_int16_t sin6_port; /* port in network byte order */
  u_int32_t sin6_flowinfo;  /* IPv6 flow information */
  struct in6_addr sin6_addr; /* IPv6 address */
  u_int32_t sin6_scope_id;   /* Scope ID */
};

/* IPv6 address: */
struct in6_addr {
  unsigned char s6_addr[16]; /* address in netw. b.o.*/
};
```

Kapcsolat orientált eset

- A kapcsolat létrehozása során a szerver és a kliens szerepe aszimmetrikus
  - A szerver csatlakozásra vár, bárki csatlakozhat hozzá
    `listen`, **`accept`**
  - A kliens dönti el, hogy melyik szerverhez csatlakozik
    `connect`
- A létrehozott kapcsolaton keresztül a kommunikáció a „másik féllel” történik, nem kell megadni, hogy kivel
  `write` , **`read`** , (**`send`** , **`recv`** , sőt van még **`writev`** , **`readv`** is)

Szerver csatlakozásra kész

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

- **`sockfd`**: socket descriptor
- **`backlog`**: hagyományosan a létrehozás alatt álló kapcsolatok (TCP 3 utas kézfogás) számának felső korlátja volt. Ha ezt kimerítették, a szerver a további kapcsolatokat visszautasítja. A 2.2-es Linux kernelben ennek jelentését megváltoztatták, a már létrejött, de még **`accept()`**-tel el nem fogadott kapcsolatok maximális számát jelenti. (man 2 listen)
- **`return`**: hiba esetén -1, egyébként 0

Megjegyzés: ez egy nem blokkoló függvényhívás.

Szerver fogadja a kliens csatlakozását

```
int accept(int sockfd,
   struct sockaddr *addr,
   socklen_t *addrlen);
```

- **`sockfd`**: socket descriptor
- **`addr`**: mutató egy címstruktúrára. Ide kerül be a kapcsolódó fél IP-címe és portszáma.
- **`addrlen`**: mutató egy kétirányú paraméterre: híváskor megadja a rendelkezésre álló hely méretét, visszatéréskor pedig a kapott címadatok tényleges méretét bájtokban.
- **`return`**: siker esetén a létrejött kapcsolat eléréséhez a socket descriptor, hiba esetén -1

Megjegyzés: ez egy blokkoló függvényhívás.

Kliens csatlakozik a szerverhez

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

- **`sockfd`**: socket descriptor
- **`serv_addr`**: mutató egy **`sockaddr`** típusú címstruktúrára, ami a kívánt cél IP címet és portszámot tartalmazza. Ez **`AF_INET`** esetén **`sockaddr_in`** típusú, **`AF_INET6`** esetén pedig **`sockaddr_in6`**, mint a **`bind()`**-nál is láttuk.
- **`addrlen`**: megadja a címstruktúra méretét bájtokban
- **`return`**: ha sikerült a kapcsolódás, akkor 0, hiba esetén pedig -1

Megjegyzés: ez is blokkoló függvényhívás.

Adatok küldése kapcsolaton át

```
ssize_t write(int fd,
   const void *buf, size_t count);
```

- **`sockfd`**: socket descriptor
- **`buf`**: mutató arra a memóriaterületre, ahol az adatok találhatók. Ezek, úgy ahogy vannak, átvitelre kerülnek.
- **`count`**: az átvinni kívánt bájtok száma
- **`return`**: hiba esetén -1, egyébként az elküldött bájtok száma.

Megjegyzések:

- Azonos az (alacsony szintű) fájlba írással.
- A két félnek összhangban kell használnia a **`write()`** és a **`read()`** függvényeket!

Adatok fogadása kapcsolaton át

```
ssize_t read(int fd,
  void *buf, size_t count);
```

- **`sockfd`**: socket descriptor
- **`buf`**: mutató arra a memóriaterületre, ahova az adatok kerülnek.
- **`count`**: a fogadni kívánt bájtok száma
- **`return`**: hiba esetén -1, egyébként a fogadott bájtok száma.

Megjegyzések:

- Szintén az (alacsony szintű) fájlból való olvasás függvénye.
- A két félnek összhangban kell használnia a **`write()`** és a **`read()`** függvényeket!

Kapcsolatmentes eset

- Minden egyes küldésnél megmondjuk, hogy kinek szeretnék datagramot küldeni: **`sendto()`**
- Minden egyes vételnél megtudjuk, hogy kitől kaptuk a datagramot: **`recvfrom()`**

Adatok küldése kapcsolat nélkül

```
ssize_t sendto(int sockfd, const
  void *buf, size_t len, int flags,
  const struct sockaddr *dest_addr,
  socklen_t addrlen);
```

- **`sockfd`**: socket descriptor
- **`buf`**: mutató arra a memóriaterületre, ahol az adatok találhatók
- **`len`**: az átvinni kívánt bájtok száma
- **`flags`**: megadható számos opció (lásd: man 2 sendto), ha nem kívánunk élvi vele, írhatunk egyszerűen 0-t.
- **`dest_addr`**: a korábban megismert struktúra a cél megadására
- **`addrlen`**: a fenti struktúra mérete
- **`return`**: hiba esetén -1, egyébként az elküldött bájtok száma.

Adatok fogadása kapcsolat nélkül

```
ssize_t recvfrom(int sockfd,
  void *buf, size_t len, int flags,
  struct sockaddr *src_addr,
  socklen_t *addrlen);
```

- **`sockfd`**: socket descriptor
- **`buf`**: mutató arra a memóriaterületre, ahova az adatok kerülnek
- **`len`**: a fogadni kívánt bájtok száma
- **`flags`**: megadható számos opció, hasonlóan, mint **`sendto()`**
- **`dest_addr`**: a korábban megismert struktúra a forrás címadatainak fogadására
- **`addrlen`**: a fenti struktúra mérete – mindkét irányban!
- **`return`**: hiba esetén -1, egyébként a fogadott bájtok száma.

A kapcsolat bontása / socket lezárása

```
int close(int fd);
```

- **`sockfd`**: socket descriptor
- **`return`**: hiba esetén -1, egyébként a fogadott bájtok száma.

Mind a **`socket()`** függvényhívással megnyitott, mind az **`accept()`**-tel „kapott” socketeket ezzel a függvénnyel kell lezárni.

FONTOS, hogy a függvények visszatérési értékét mindig ellenőrizzük! Vannak a lehetséges hibák kiírására szolgáló könyvtári függvények…

Bájtsorrend konverziós függvények

- Alapvető 16 és 32 bites bájtsorrend konverziós függvények.

  Névkonvenció: <honnan>to<hova><hossz>: {h|n}to{n|h}{s|l}

  h=host, n=network; s=short, l=long (az adott helyen értve)

```
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

  Bővebben: man 3 byteorder

IP-cím konverziók

```
in_addr_t inet_addr(const char *cp);
```

A **cp** stringben megadott IP-címet adja vissza binárisan, hálózati bájtsorrendben. Problémát okozhat, hogy a -1 értéket használja hiba jelzésére, de ez megegyezik az érvényes 255.255.255.225 érték ábrázolásával. Helyette:

```
int inet_aton(const char *cp, struct in_addr
   *inp);
```

Az eredménynek külön van helye (inp), és az érvénytelen inputot a 0 visszatérési értékkel jelzi. A másik irány:

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

Bináris, hálózati bájtsorrendben megadott IP-címet szövegesen ad vissza. FIGYELEM: statikus puffert használ, tehát ki kell másolni, mert legközelebb felülírja!

Ha valaki programozni szeretne…

`int gethostname(char *name, size_t len);`

A saját gépünk nevének lekérdezése a **`name`** stringbe, ami **`len`** hosszú.

Példaprogramokban előfordulnak:

`gethostbyname(); gethostbyaddr();`

De ne használjuk, mert elavultak (obsolete)!

Helyette:

`getaddrinfo(); getnameinfo();`

Kicsit bonyolultak, de ott a man. ☺
Cserében reentránsak, IPv4-hez és IPv6-hoz is használhatók.

Példaprogram: http://whale.hit.bmehu/socket