Beej útmutatója a hálózati programozáshoz

Internet Socketek használatával

Brian "Beej" Hall

beej@piratehaven.org

Fordította: Hajdu Gábor
triasz@inf.elte.hu
(Az esetleges hibákért semmi felelõsséget nem vállalok!)

Copyright © 1995-2001 by Brian "Beej" Hall

Módosított kiadások (Revision History):

Revision Version 1.0.0 August, 1995	Revised by: beej
Kezdeti verzió.	
Revision Version 1.5.5 January 13, 1999	Revised by: beej
Legújjab HTML verzió.	
Revision Version 2.0.0 March 6, 2001	Revised by: beej
DocBook XML formátumba konvertálva, helyesbítések, pótlások.	
Revision Version 2.3.1 October 8, 2001	Revised by: beej
Kijavított sajtóhibák, szintaktikai hibák a client.c-ben, új anyagok a Q&A (kérdés/válasz) részhez.	

Tartalom

1. **Bevezetõ**
   1. Az olvasónak
   2. Platform és fordító
   3. Hivatalos honlap
   4. Megjegyzés a Solaris/SunOS programozóknak
   5. Megjegyzés a Windows programozóknak
   6. EMail irányelvek
   7. Tükrözések (Mirror)
   8. Megjegyzés a fordítókhoz
   9. Copyright and Distribution
   10. A fordító hozzászól
2. **Mi is az a socket?**
   1. Az Internet Socketek két típusa
   2. Alacsony szintû zagyvaság és hálózati elmélet
3. ***structs* és adatkezelés**
   1. Convert the Natives!
   2. IP cím, Hogyan foglalkozzunk velük?
4. **Rendszer hívások vagy fagyások**
   1. `socket()` - Vegyük a fájlleírót!
   2. `bind()` - Milyen porton vagyok?
   3. `connect()` - Hey, te!
   4. `listen()` - Felhívna valaki?
   5. `accept()` - "Köszi hogy hívtál 3490-es port."
   6. `send()` és `recv()` - Szólj hozzám bébi!
   7. `sendto()` és `recvfrom()` - Beszélj hozzám, DGRAM-stílus

8. `close()` és `shutdown()` - Viszlát!
9. `getpeername()` - Ki vagy?
10. `gethostname()` - Ki vagyok?
11. DNS - Amire te azt mondod "pandora.inf.elte.hu", én azt mondom "157.181.160.0"

5. **Kliens-Szerver háttér**
   1. Egy egyszerû Stream szerver
   2. Egy egyszerû Stream kliens
   3. Datagram socketek
6. **Kicsit fejlettebb teknikák**
   1. Blokkolás
   2. `select()` - Szinkron I/O Multiplexelés
   3. Parciális `send()`-ek kezelése
   4. Az adatbeágyazás leszármazottja
7. **További referencia**
   1. **man** oldalak
   2. Könyvek
   3. Webes referenciák
   4. RFC-k
8. **Általános kérdések**
9. **Helyreigazítás és segítségkérés**

1. Bevezetõ

Hali! Csak nem socketet akarsz programozni? Talán egy kicsit nehéznek találod a **man** oldalakból kiböngészni, hogy mi a pálya? Internetet akarsz programozni, de nincs idõd beleveszni a `struct`-ok adagjaiba hogy megpróbálj rájönni, hogy kell-e `bind()`-ot hívnod `connect()`-elés elõtt, stb, stb.

Nos, tudod mit? Én éppen most végeztem ezzel a kényes dologgal, és majd meghalok, hogy megoszthassam az információimat mindenkivel! A legjobb helyre jöttél. Ha valami már mocorog a C nyelv hallatán, akkor ezt a doksit végiggyûrve talán még hálózatot is meg tanulsz programozni. - Legyen úgy...:-)

1.1. Az olvasónak

Ez a doksi tanítási célt szolgál, azaz ez nem egy referencia. Talán a legjobb azoknak, akik még sohasem próbálkoztak a socket programozással, és valami talpalávalóra vágynak. Természetesen ez nem egy *teljes* útikalaúz a socket programozáshoz.

Remélem mindamellet, hogy elég lesz ahhoz, hogy a man oldalak láttán már valami halvány fogalmad legyen az ügyrõl...:-)

1.2. Platform és fordító

Az ebben a doksiban írt kódot Linuxos PC-n fordítottam a Gnu **gcc**-jét használva. Ez elvileg minden olyan platformon fordítható, amely **gcc**-t használ. Természetesen ez nem kóser, ha Windowsban programozol - lásd a Windows programozásáról szóló részt lejjeb.

1.3. Hivatalos honlap

A hivatalos helye ennek a dokumentumnak a következõ helyen van: California State University, Chico, a következõ címen:

`http://www.ecst.csuchico.edu/~beej/guide/net/`[1].
Ez fordítás a következõ helyen található meg:
`http://people.inf.elte.hu/triasz/socket/net/`.

1.4. Megjegyzés a Solaris/SunOs programozóknak

Solaris illetve SunOS alatt szükség van még pár extra parancssor kapcsolóra, hogy a megfelelõ könyvtárakat hozzákapcsoljuk a programohoz. Csak annyit kell tenni, hogy a "`-lnsl -lsocket -lresolv`" kapcsolókat a fordító parancssor végéhez adjuk eképp:

```
$ cc -o server server.c -lnsl -lsocket -lresolv
```

Ha még mindig hibákat kapsz, akkor próbáld ki a "`-lxnet`" plussz hozzáadását az elõbbi parancssorhoz. Nem tudom pontosan mit csinál ez, de páran úgy tapasztalták, hogy szükség van rá.

Egy következõ hely ahol hibákba ütközhetsz, a `setsockopt()` függvény hívása. Az õsalak eltér attól, ami Linuxban megy, így a következõ helyett:

```
int yes=1;
```

írd ezt:

```
char yes='1';
```

Mivel nekem nincs Sun rendszerem, ezért még nem állt módomban tesztelni az elõbb adott információkat - ezek olyan dolgok voltak, amiket mások küldtek nekem emailben.

1.5. Megjegyzés a Windows programozóknak

A Windowstól különleges idegenkedés fog el, és szeretnék mindenkit ösztönözni arra, hogy inkább próbáljon meg Linux, BSD, vagy Unixot használni helyette. Ez az én véleményem, de ettõl függetlenül használhatod Windows alatt is ezt az egészet.

Elõször is ne foglalkozz azokkal a header fájlokkal, amiket itt említek. Neked csak a következõt kell includolnod:

```
#include <winsock.h>
```

Várj! Emellett még kell intézned egy hívást a `WSAStartup()` függvénnyel, mielõtt bármit is tennél a socket könyvtáraddal. A kód ehhez valahogy így néz ki:

```
#include <winsock.h>

{
      WSADATA wsaData; // ha ez nem mûködne
      //WSAData wsaData; // próbáld meg ezt helyette

      if (WSAStartup(MAKEWORD(1, 1), &wsaData) != 0) {
            fprintf(stderr, "WSAStartup failed.\n");
            exit(1);
      }
```

Ezenkívül még utasítanod kell a fordítód, hogy létesítsen kapcsolatot a Winsock könyvtárban egy fájllal, amit általában `wsock32.lib` vagy `winsock32.lib` vagy valami ilyesminek hívnak. VisualC++ alatt ezt a `Project` menü alatt a `Settings...` menüpontban teheted meg. Klikkelj a `Link` fülre, és keress egy olyan mezõt aminek "Object/library modules" a neve. Ehhez a felsoroláshoz add hozzá a "wsock32.lib" fájlt.

Valahogy így hallottam.

Végezetül meg kell még hívnod a `WSACleanup()` függvényt, amikor végeztél a socket könyvtárral. Részletekért lásd az online segítséget.

Miután egyszer ezeket megtetted, a többi példa is többnyire alkalmazható ebben a leírásban, eltekintve egypár kivételtõl. Még egy dolog: itt nem használhatod a `close()` függvényt a socket bezárására - ehelyett a `closesocket()` függvény van. Ezenkívül a `select()` csak socket leírókkal mûködik, fájlleírókkal nem (mint `0` az `stdin`nek).

Ezenkívül van még egy socket osztály amit használhatsz: `CSocket`. Nézz utánna a fordítód segítség részében további információért.

A Winsockról további információkat a Winsock FAQ[2]-ban olvashatsz.

Végezetül azt hallottam, hogy a Windowsnak nincs `fork()` rendszerhívása, amelyet sajnos jó pár példámban használtam. Valószínû egy POSIX könyvtárhoz kell kapcsolódni, ha használni szeretnéd, vagy használhatod a `CreateProcess()` függvényt helyette. A `fork()` függvénynek nincsenek argumentumai, ezzel szemben a `CreateProcess()` 48 billiót tartalmaz. Ha ez nem dobott fel, a `CreateThread()` egy kicsitt egyszerûbben emészthetõ... sajnos a többszálúság (multithreading) ennek a dokumentumnak a hatókörén kívül esik. Tudod milyen sokat tudnék csak errõl beszélni!

1.6. EMail irányelvek

Többnyire elérhetõ vagyok segítségnyújtás szempontjából, így írj nyugodtan emailt ha elakadtál, igaz, választ nem garantálok. Elég sûrû az életem így vannak idõkszakok, amikor egyszerûen nem tudok visszaírni a kérdésedre. Ilyen helyzetben rendszerint letörlöm az üzenetet. Semmi személyes; pont nem lesz annyi idõm, hogy olyan részletes választ adjak, mint szeretnél.

Rendszerint minél összetettebb egy kérdés, annál kisebb a valószínûsége, hogy válaszolok. Ha szûkszavakban minden szükséges információval elláttad a kérdésed (mint platform, fordító, kapott hiba üzenetek, és minden egyéb amirõl azt gondolod segíthet a segítségadásban), akkor elég nagy esélyed van hogy választ is kapsz. Több útmutatáshoz olvasd el az ESR dokumentumát, How To Ask Questions The Smart Way[3]. (Hogyan tegyünk fel kérdéseket gyors és találó módon)

Ha nem kapsz visszajelzést, kísérletezz még egy kicsit, hogy kitaláld a választ, és ha még ezek után is megfoghatatlan a probléma, írj újra az információkkal amiket találtál és reménykedj, hogy az elég nekem, ahhoz hogy kisegítselek.

Eleget szekáltalak már azzal, hogy hogyan írj illetve ne írj, így már csak azt szeretném, ha tudnád, hogy teljes mértékben nagyra értékelem a dícséreteket, amiket az elmúlt évek alatt kaptam erre az útmutatóra. Ez a közérzetemet rendesen megnövelte, és igazán megörvendeztet, hogy azt hallom, jó célokra használják! :-) Köszönöm!

1.7. Tükrözések (Mirror)

Nagyon jó néven veszem, ha tükrözni akarod ennek az oldalnak a lelõhelyét, akár publikációs, akár magán célra. Ha publikációs célra tükrözöd az oldal lelõhelyét és szeretnéd, hogy a fõoldalamról csináljak egy linket rá, akkor írj egy sort a következõ címre: "beej@piratehaven.org".

1.8. Megjegyzés a fordítónak

Ha le szeretnéd fordítani ezt a cuccost egy másik nyelvre, akkor írj nekem a "beej@piratehaven.org" címre és akkor csinálok egy linket a fordításodhoz a fõoldalamról.

Nyugodtan írd rá a neved és az email-címedet a fordításra.

Bocsi, de helyproblémák miatt nem áll módomban a saját gépemen tárolni a fordításod.

1.9. Copyright and Distribution

Beej's Guide to Network Programming is Copyright (C) 1995-2001 Brian "Beej" Hall.

Ez az útmutató szabadon másolható bármilyen adathordozóra olymódon, hogy a tartalmát megõrzi, az teljességében marad, a copyright felirattal együtt.

Az oktatókat szertném buzdítani, hogy javasolják és lássák el ezzel az anyaggal a diákjaikat.

Ez az útmutató szabadon fordítható bármely nyelvre, megvédve a tartalmát, de csak teljes tartalmával együtt másolható. A fordítás tartalmazhatja a fordító nevét és elérési adatait.

Ebben a dokumentumban bemutatott C forráskód ezúton meg van adva a publikus domainban.

Kapcsolat további információért: "beej@piratehaven.org" .

1.10. A fordító hozzászól

Üdv mindenkinek! Úgy tûnik végre sikerült "lefordítani" ezt a kis cuccost! Úgy, ahogy... Valószínû tele van hibákkal, de legalább MAGYAR!!! :-) Hurrá!!!
Használd egészséggel, és ne nézd a helyesírást, így mind a ketten jól járunk...:-)
A fordítás minõségéért és a tartalomért természetesen semmi felelõsséget sem vállalok. Ha biztosra akarsz menni, az eredeti angol dokumentumot megtalálod a már fentebb említett helyen.
Ha esetleg valami hasznos megjegyzésed lenne, vagy csak áldani akarod jótettemet, itt megteheted: triasz@inf.elte.hu .
Sok sikert!

2. Mi is az a socket?

Sokszor hallod, hogy a "socketekrõl" beszélnek, és valószínû kíváncsi vagy, hogy mi is az pontosan. Nos, ez: egy mód, hogy más programokkal kommunikáljunk, amelyek standard Unix fájlleírót használnak.

Mi van?

Na jól van - talán már hallottad pár Unix hacker állítását, "'Jeez', a Unixban *minden* egy fájl!" Az amirõl beszélhetett azaz, hogy amikor a Unix programok valamilyen fajta I/O mûveletet hajtanak végre, azt egy fájlleíró olvasásával vagy írásával teszik A fájlleíró egyszerûen egy egész szám (integer), ami egy megnyitott fájlhoz van társítva. De (és itt a poén), az a fájl akár egy hálózati kapcsolat is lehet, egy FIFO, egy 'pipe', egy terminál, egy valódi on-the-disk fájl, vagy akármi más. Minden a Unixban egy fájl! Így amikor kommunikálni szeretnél egy másik programmal az Interneten keresztül, azt egy fájlleírón keresztül fogod megtenni, - és ezt jobb, ha elhiszed.

"És mégis honnan vegyem ezt a fájlleírót a hálózati kommunikációhoz, Mr. Okos-tojás?" valószínû a jelenlegi kérdés a fejedben, de én megpróbálom megválaszolni: hívást indítasz a `socket()` rendszer rutinhoz. Ez visszaadja a socket leírót, és te ezen keresztül kommunikálsz a specializált `send()` és `recv()` **(man send[4],**

man recv[5]) socket hívásokkal.

"De hey!" kiállthatod most el magad. "Ha ez egy fájlleíró, akkor hogy az ördögben nem használhatom a normális *read()* és `write()` hívásokat a socketen keresztüli kommunikációhoz?" A rövid válasz: "Használhatod!"
A hosszab válasz pedig: "Használhatod, viszont a `send()` és a `recv()` sokkal nagyobb kontrollt ad az adatszállításra."

Hogyan tovább? Mi legyen a következõvel: minden fajta socketek vannak. Vannak a DARPA Internet címek (Internet socketek), útvonal nevek egy helyi node-on (Unix Sockets), CCITT X.25 címek (X.25 Sockets, amit biztonságosan figyelmen kívül hagyhatsz), és valószínüleg nagyon sok más is, attól függ, hogy milyen Unixot futtatsz. Ez a dokumentum csak az elsõvel, az Internet Socketekkel foglalkozik.

2.1. Az Internet Socketek két típusa

Mi ez? Két fajtája van az Internet socketnek? Igen. Nos, nem. Hazudtam. Több van, de nem akartalak megrémíszteni. Én most csak két fajtáról szeretnék itt beszélni. Eltekinteve ettõl a mondattól, amit most olvasol, mert itt elmondom, hogy a "Raw Sockets"-ek szintén nagyon elterjedtek és érdemes lenne utánna nézned.

Na jól van már. Mi az a két típus? Az egyik a "Stream Sockets"; a másik a "Datagram Sockets", amelyekre ezekután a következõ módon fogok hivatkozni: "`SOCK_STREAM`" és "`SOCK_DGRAM`". A Datagram socketeket néha "kapcsolatnélküli socketeknek" hívják. (Mindamellett `connect()`-elheted õket, ha tényleg akarod. Lásd a `connect()`-et lejjebb.)

A Stream socketek megbízható kétirányú kapcsolat kommunikációs folyamok. Ha elküldesz két tételt a socketen "1,2" sorrendben, akkor azok "1,2" sorrendben fognak megérkezni a túloldalra. Ezentúl hibamentesek maradnak. Bármilyen más hiba, amivel találkozol, a te agyad szüleménye, és azok nem érdemelnek tárgylást itt.

1. ábra - Adat beskatulyázódás:

Mi használja a stream socketeket? Nos, ugye hallottál már a **telnet** alkalmazásról? Ez stream socketet használ. Minden karakter, amit begépelsz, ugyanúgy kell megérkezzen, ahogy begépelted. Rendben? A Web böngészõk a HTTP protokolt használják, amely a stream socketet használja az oldalak lehívásához. Csakugyan, ha te telnetezel egy web oldalra a 80-as porton, és beírod "`GET /`", akkor visszadobja neked a HTML-t!

Hogyan tudja a stream socket megvalósítani ezt a magas szintû adatszállítási minõséget. A "The Transmission Control Protocol"-t azza a "TCP"-t használja erre (lásd az RFC-793[6] -at bõségesebb infoért a TCP-rõl.). A TCP biztosítja, hogy az adatok sorrendtartón és hibamentesen érkezzenek meg. Már hallhattál a "TCP"-rõl, mint a "TCP/IP" jobbik része, ahol az "IP" az "Internet Protocolt" jelenti (lásd RFC-791[7].) IP bánik elsõdlegesen az Interneten való irányítással és ez nem vállal felelõsséget az adatok helyességéért.

Király. Mi van a Datagram socketettel? Miért hívják kapcsolatnélkülinek? Mi is a megállapodás itt mindenesetre? Miért megbízhatatlanok? Nos, itt van pár eset: ha te elküldesz egy datagramot, valószínû megérkezik. Nem feltétlenül sorrendben érkezik meg. Ha megérkezik, az adat a csomagon belül hibamentes.

A Datagram socketek az IP-t használják irányításra, de nem használják a TCP-t, hanem a "User Datagram

Protocol"-t, azaz "UDP"-t használják. (lásd RFC-768[8].)

Miért kapcsolatnélküliek? Nos, alapjába véve, nincs szükség nyitott kapcsolat fenntartásához, ellenben a srtream sockettel. Csak készítesz egy csomagot, hozzácsapsz egy IP headert célinformációkkal, és kiküldöd. Nincs szükség kapcsolatra. Az információ packet-by-packet típusú szállítására használják. Ilyen alkalmazások például: **tftp**, **bootp**, stb.

"Elég!" - kiállthatsz most fel. "Hogyan mûködnek ezek a programok, ha fennáll a veszélye, hogy a datagramok elvesznek?!" Nos barátom, mindegyik tartalmaz egy saját protokolt az UDP tetején. Például, a tftp protocol minden elküldendõ csomagnak azt mondja, hogy a fogadó oldalról küldjön vissza egy csomagot, ami azt mondja "Megkaptam!" (egy "ACK" csomag.) Ha az eredeti csomag küldõje nem kap választ, mondjuk 5 másodperc múlva, akkor újra küldi a csomagot, amíg végül kap egy ACK-t. Ez az elismerési folyamat nagyon fontos amikor `SOCK_DGRAM` alkalmazásokat implementálunk.

2.2. Alacsony szintû zagyvaság és hálózati elmélet

Amióta szóltam a protokolok rétegzõdésérõl, azóta itt az ideje beszélni arról, hogy a hálózatok valójában hogy is mûködnek, és nézni pár példát arra, hogy a `SOCK_DGRAM` csomagok hogyan is épülnek fel valójában.

Hé gyerekek, itt az ideje tanulni valamit az *adatbeágyazódásról*! Ez nagyon fontos! Fõleg úgy, hogy a hálózati kurzus keretében valószínû fogsz róla tanulni itt a Chico State-en ;-). Alapjában véve arról szól, hogy a csomag létrejön, aztán bebugyolálódik ("beskatulyázódik") a headerben (és ritkán a footerben) az elsõ protokol által (mondjuk a TFTP protocol által), aztán az egész cucc (beleértve a TFTP headert is) újra beskatulyázódik a következõ protokolba (mondjuk az UDP-be), aztán a következõbe (IP), és aztán a végsõ protokolba a hardver (fizikai) rétegen (mondjuk Ethernet).

Amikor egy másik számítógép megkapja a csomagot, a hardver lefejti az Ethernet headert, a kernel lefejti az IP és az UDP headert, aztán a TFTP program a TFTP headert, és végül megkapja az adatot.

Most már végre tudok beszélni a hírhedt *rétegelt hálózati modell*rõl (*Layered Network Model*). Ez a hálózati modell írja le hálózti függõségek rendszerét, amelynek több elõnye van a többi modellel szemben. Például olyan socket programokat tudsz írni, melyek pontosan ugyanolyanok anélkül, hogy tõrõdnének az adat fizikai továbbításának módjával (serial, thin Ethernet, AUI, stb.) mivel a programok egy alacsonyabb szinten elbánnak már ezzel neked. Az aktuális hálózati hardver és topológia láthatatlan a socket programozó számára.

Minden további nélkül be fogom mutatni a teljes modell rétegeit. Emlékezz majd ezekre a hálózati vizsgákon:

- Alkalmazás (Application)
- Beállítási (Presentation)
- Végrehajtási (Session)
- Szállítási (Transport)
- Hálózati (Network)
- Adat kapcsolati (Data Link)
- Fizikai (Physical)

A fizikai réteg a hardver (serial, Ethernet, stb.). Az alkalmazási réteg pedig olyan messze van a fizikai rétegtõl, amennyire csak el tudod képzelni - ez az a hely, ahol a felhasználó kölcsönhatást létesít a hálózattal.

Ez a modell túl általános, igy max csak úgy tudod használni mint egy autószerelõ kézikönyvet, ha akarod. A rétegelt modell alkalmazhatóságai Unix alatt a következõk lehetnek:

- Alkalmazási réteg (Application Layer) (*telnet, ftp, stb.*)
- Host-Host szállítási réteg (Host-to-Host Transport Layer) (*TCP, UDP*)
- Internet réteg (Internet Layer) (*IP és útvonalkeresés*) (*IP and routing*)
- Hálózati hozzáférési réteg (Network Access Layer) (*Ethernet, ATM, vagy bármi más*)

Most ezen a ponton valószínû már láthatod, hogy ezek a rétegek hogyan vannak kapcsolatban az eredeti adat beágyazódásával.

Lássuk, hogy mennyi munkába is kerül felépíteni egy sima csomagot?! Hûha! Nos, elõször is neked kell megírni a csomag headert a "**cat**" használatával! Na jól van, csak vicceltem. Az össz dolgod a stream socketnek, hogy ki-`send()`-eled az adatokat. Az össz dolgod a datagram socketekkel, hogy beágyazd a csomagot az általad kiválasztott eljárásba (method) és ki-`sendto()`-lod. A kernel létrehozza a szállítási és az Internet réteget, a hardver pedig a hálózati hozzáférési réteget (Network Access Layer). Oh, a modern technológia...

Így befejeztük az eligazítást a hálózti elméletbe való betekintésbõl. Ó igen, mindent elfelejtettem elmondani az útvonal keresésrõl (routing): azaz semmit sem! Ez így van, nem szándékozok errõl beszélni. A router leválasztja a csomagot az IP headerrõl, megnézi a routing tábláját, bla,bla,bla. Ha nagyon érdekel a dolog, utánna nézhetsz az IP RFC[9]-ben. Nem leszel nagy veszélyenk kitéve az életed során, ha nem tanulsz róla.

3. *struct*ok és adatkezelés

Nos, végül ide értünk. Itt az ideje beszélni a programozásról. Ebben a fejezetben megtárgyaljuk az adattípusok különbözõ fajtáinak használatát socket interfésszel, párat közülük kicsit jobban is szemügyre veszünk.

Elõször egy egyszerût: egy socket leíró. A socket leíró a következõ típus:

```
int
```

Csak egy szokásos `int`.

A dolgok viszont innentõl bebonyolódnak, igy csak olvasd át és légy türelemmel velem. Ezt tudod: két falyta bájtelhelyezés van: legértékesebb bájt (néha "octet"-nek hívják) elõször, vagy a legkevésbé értékes bájt elõször. Az imént említettet "Network Byte Order"-nek hívják. Pár gép eltárolja a saját a számait a Network Byte Order belsejében, valamelyik pedig nem. Amikor azt mondtam, hogy valamelyik eltárolja a Network Byte Orderben, akkor neked egy függvényt (olyat mint a `htons()`) kell meghívnod, hogy megváltoztasd "Host Byte Order"-rõl. Ha nem mondom, hogy "Network Byte Order", akkor Host Byte Orderként kell hagynod az értékeket.

(A pontosság kedvéért, a "Network Byte Order" "Big-Endian Byte Order"-ként is ismert.)

Az elsõ Structom[TM]-`struct sockaddr`. Ez a struktúra tárolja a címinformációt a legtöbb sockettípushoz:

```
struct sockaddr {
```

```
        unsigned short sa_family; // címcsalád, AF_xxx
        char sa_data[14]; // a protocol cím 14 bájtja
};
```

sa_family sokfajta lehet, de ebben a dokumentumban mi mindig `AF_INET`-ként használjuk.
sa_data a cél címét és portszámát tartalmazza a socketnek. Ez nagyon kényelmetlen mivel te nem akarod unalmasan kézzel bepötyögni a címet az *sa_data*-ba.

A struct sockaddr-al való foglalkozáshoz a programozók létrehoztak egy hasonló struktúrát: `struct sockaddr_in`("in" az "Internet"-hez.)

```
struct sockaddr_in {
        short int sin_family; // cím család
        unsigned short int sin_port; // port szám
        struct in_addr sin_addr; // internet cím
        unsigned char sin_zero[8]; // ugyanakkora méretben, mint a struct sockaddr-ben
};
```

Ez a struktúra egyszerûvé teszi a socket address elemeihez való hivatkozást. Megjegyzés, hogy a *sin_zero*-t (amelyet azért vettük bele, hogy a `struct sockaddr` hosszára egészítsük ki a struktúránkat) a `memset()` függvénnyel végig nullára kell állítani. Ezenfelül, és ez a fontos **???bit**, egy `struct sockaddr_in`-re mutató pointert egy `struct sockaddr`-ra castolhatunk és fordítva. Így még akkor is ha a `socket()` egy `struct sockaddr*` mutatót akar, akkor is te még mindig használhatod a `struct sockaddr_in`-t és castolhatod az utolsó percben! Ezenfelül jegyezd meg, hogy a *sin_family* megegyezik az *sa_family*-vel a struct sockaddr-bõl, és mindenképp "`AF_INET`"-re kell állítani. Végezetül a *sin_port* és a *sin_addr*-nek *Network Byte Order*-nek kell lennie!

"De" - tiltakozhatsz - "hogyan tud az egész struktúra, `struct in_addr sin_addr`, Network Byte Orderben lenni?" Ez a kérdés a `struct in_addr` struktúra óvatos megvizsgálását igényli, egyik a legrosszabb létezõ unionokból:

```
// Internet address (cím) (egy struktúra kiemelkedõ okokkal)
struct in_addr {
        unsigned long s_addr; // ez egy 32 bites long, vagy 4 bájt
};
```

Nos, ezt unionként haszánltuk, de azok a napok úgy tûnik, hogy elmúltak. Hál' Istennek, megszabadultunk tõlük! Így ha deklarálunk egy *ina* nevû `struct sockaddr_in` típusú változót, akkor az *ina.sin_addr.s_addr* egy 4 bájtos IP címre hivatkozik (Network Byte Orderben). Megjegyezzük, hogy ha a te rendszered esetleg még mindig azt az Istenverte union-t használja a struct in_addr-hez, akkor is ugyanolyan módon hivatkozhatsz rá, mint ahogyan én az elõbb tettem (ez a `#defines`-nak köszönhetõ.)

3.1. Convert the Natives!

Egy újabb fejezetbe vezettek minket. Már így is sokat beszéltünk erről a hálózat - Host Byte Order konverzióról - itt az ideje, hogy csináljunk is végre valamit!

Rendben. Két típus van, amit konvertálhatsz: `short` (2 bájt) és `long` (4 bájt). Ezek a függvények az `unsigned` variációkkal szintén mûködnek. Azt mondod, hogy egy `short` típust szeretnél Host Byte Orderrõl Network Byte Orderbe konvertálni. Kezd "h"-val a "host" miatt, ezt kövesse "to", aztán "n" a "network" miatt, aztán "s" a "short" miatt: h-to-n-s, vagy `htons()` (olvasd: "Host to Network Short").

Ez majdnem hogy túl könnyû...

Az összes kombinációt használhatod, akár "n","h","s" és "l" az amit szertnél, nem számolva a hülyeségeket.

Például NINCS `stolh()` ("Short to Long Host") függvény. De a következõk léteznek:

- `htons()` - "Host to Network Short"
- `htonl()` - "Host to Network Long"
- `ntohs()` - "Network to Host Short"

- `ntohl()` - "Network to Host Long"

Most úgy gondolhatod, hogy eleget tudsz. Azt is gondolhatod, "Mit fogok csinálni, ha egy byte order char-t kell megváltoztatnom?". Aztán azt is gondolhatod, "Ó, ne aggódj." Esetleg azt is gondolhatod, hogy mivelhogy 68000 gép már network byte ordert használ, nem kell `htonl()`-t hívnod az IP címeiden. Igazad lehet, DE ha egy olyan gépre akarsz csatlakozni, amely network byte ordertõl különbözõt használ, akkor a programod megbukik. Legyél portabilis! Ez egy Unix világ! (Ugyanannyira amennyire Bill Gates az ellenkezõjét szeretné gondolni.) Emlékezz: tedd a bájtjaidat Network Byte Orderbe mielõtt a hálózatba teszed õket.

A befejezõ pont: miért kell hogy a *sin_addr* és a *sin_port* Network Byte Orderben legyen a `struct sockaddr_in`-ben, ellenben a *sin_family*-vel, aminek nem kell? A válasz: a *sin_addr* és *sin_port* az IP és illetõlegesen az UDP rétegbe tokozódik be. Ezért kell, hogy ezek Network Byte Orderben legyenek. Bármennyire is a *sin_family* mezõt csak a kernel használja hogy megállapítsa, hogy a struktúra milyen típusú címet tartalmaz, így ezért kell Host Byte Ordernek lennie. Így mindaddig, amíg a *sin_family* nem lesz a hálózatra küldve, addig maradhat Host Byte Order-ben.

3.2. IP cím, Hogyan foglalozzunk velük?

Szerencsédre egy csokor függvény létezik, amelyek könnyû kezelhetõséget adnak az IP címekhez. Nincs szükség kézzel való beállítgatásokra, és berakni egy `long`-ba az << operátorral.

Elõször mondjuk, hogy van egy ina nevû `sockaddr_in` struktúrád, és van egy "`10.12.110.57`"-es IP címed, amelyet el szeretnél benne tárolni. A függvény, amit használni szeretnél, `inet_addr()`, egy IP címet számok-és-pontok jelölésrendszert konvertál egy unsigned long típusba. A feladatot a következõ módon oldhatjuk meg:

```
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

Vedd észre, hogy az `inet_addr()` a címet Network Byte Orderben adja már vissza - így nem kell meghívni a `htonl()` függvényt.
Klassz!

Az alábbi kódrészlet nem túl robosztus, mert nincs benne hibafigyelés. Lásd `inet_addr()` hiba esetén -1 -et ad vissza. Emlékszel a bináris számokra? (`unsigned`(elõjel nélküli))-1 úgyesik, hogy pont a `255.255.255.255`-ös IP címmel egyezik meg! Ez a broadcast cím! Wrongo. El ne felejtsd a hiba kezelést normálisan megcsinálni.

Jelenleg van egy tisztább interfész amit az `inet_addr()` helyett használhatsz: ennek a neve `inet_aton()` ("aton" az "ascii to network"-öt jelenti):

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

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

És itt van egy használati minta, amíg egy `struct sockaddr_in` -t csomagolsz (ez a példa már ad egy kis rálátást a `bind()` és a `connect()` fejezetekhez is.)

```
struct sockaddr_in my_addr;

my_addr.sin_family = AF_INET; // host byte order
my_addr.sin_port = htons(MYPORT); // short, network byte order
inet_aton("10.12.110.57", &(my_addr.sin_addr));
```

```
memset(&(my_addr.sin_zero), '\0', 8); // nulla a strutúra maradék részében
```

`inet_aton()`, *különbözik tulajdonképpen minden más elmondott socket függvénytõl*, nem nullát ad vissza sikeres végrehajtás esetén, és nullát hibánál. A címet pedig visszaadja *inp*-be.

Sajnos nem minden felületre van implementálva az `inet_aton()`, habár ennek a használata jobban kedvelt, ennek az útmutatónak a további részében az `inet_addr()` függvényt használjuk.

Rendben, most már tudsz string típusú IP címet a bináris megfelelõjére konvertálni. Mi van a másik iránnyal? Mi van, ha van egy `in_addr` struktúrád és számokkal és pontokkal akarod kiiratni? Ebben az esetben az `inet_ntoa` ("ntoa" a "network to ascii"-t jelent) függvényt szeretnéd használni, mint ez:

```
printf("%s", inet_ntoa(ina.sin_addr));
```

Ez ki fogja irni az IP cimet. Vedd észre, hogy az `inet_ntoa()` egy `in_addr` struktúrát vesz paraméterként, nem pedig long típust. Azt is vedd észre, hogy ez egy char típusra mutató mutatót (pointer) ad vissza. Ez egy statikusan tárolt karaktertömbre mutat az `inet_ntoa()`-ban így minden egyes alkalommal amikor meghívod az `inet_ntoa()` függvényt, akkor az utoljára lekérdezett IP cím felülíródik az újjal. Például:

```
char *a1, *a2;
.
.
a1 = inet_ntoa(ina1.sin_addr); // ez a 192.168.4.14
a2 = inet_ntoa(ina2.sin_addr); // ez a 10.12.110.57
printf("address 1: %s\n",a1);
printf("address 2: %s\n",a2);
```

a következõt fogja kiírni:

```
address 1: 10.12.110.57
address 2: 10.12.110.57
```

Ha szükséged van arra, hogy elmentsd a címet, akkor használd a `strcpy()` függvényt, hogy átmásold a saját karaktertömbödbe.

Ez minden errõl a témáról mostanra. Késõbb megfogod tanulni, hogy hogyan konvertálj string típust mint például a "whitehouse.gov" a neki megfelelõ IP címre (lásd a DNS címû részt alább.)

4. Rendszer hívások vagy fagyások

Ebben a fejezetben a rendszerhívásokat fogjuk tárgyalni. Azokat, amelyek hozzáférést engednek egy Unixos doboz hálózati függvényihez. Amikor te meghívsz egyet ezek a függvények közül, akkor a kernel veszi kezelésbe a dolgokat, és mindent elintéz neked autómatikusan. Hát nem csodálatos?

A hely, ahol a legtöbb ember elakad, hogy milyen sorrendben hívja meg ezeket a dolgokat. Ebben az esetben a **man** oldalak sem segítenek, ahogy már biztosan te is felfedezted. Nos, hogy segíthessek ezen a rettenetes helyzeten, megpróbáltam felsorakoztatni a rendszerhívásokat a következõ fejezetekben *pontosan* (körülbelül) abban a sorrendben, ahogy neked a programodban meg kell majd hívnod õket.

Ezeket itt-ott kisebb kóddarabkákkal párosítottam, egy kis tej és süti (amelyeket félek, hogy fel is használsz majd öncélúan), és pár nyers bél és bátorság, és aztán úgy fogod az adatokat sugározni az Interneten mint Csernobil 86-ban!

4.1. `socket()` - Vegyük a fájlleírót!

Szerintem nem kell túl hosszú idõt rászánnom - a `socket()` függvényrõl kell beszélnem. Itt az elemzés:

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

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

De mik ezek a paraméterek? Az elsõ, a *domain* "`AF_INET`"-ként szükséges hogy be legyen állítva, éppen úgy, mint a `sockaddr_in` struktúrában (feljebb). A következõ, a *type* paraméter mondja meg a kernelnek, hogy milyen fajta socket is ez: `SOCK_STREAM` vagy `SOCK_DGRAM`. Végezetül, csak állítsd a *protocol* paramétert "0"-ra, hogy a socket()-et hagyjuk kiválasztani a *type*-ra épülõ helyes protokolt. (Megjegyzés: sokkal több *domain* van, mint amennyit én felsoroltam. Sokkal több *type* van, mint amiket felsoroltam. Lásd a socket() man oldalát. Ezenfelül van egy "jobb" módszer is a *protocol* meghatározására. Lásd a `getprotobyname()` man oldalt.)

A `socket()` egyszerûen egy socket leíróval tér vissza, melyet a késõbbi rendszerhívásoknál használhatsz, illetve -1 értéket ad hiba esetén. A globális *errno* változó a hiba értékére állítódik (lásd a `perror()` man oldalt.)

Pár dokumentációban, látni fogod egy misztikus "`PF_INET`" említését. Ez egy vízduzzasztó gáttal ellátott földöntúli állatság, amit ritkán látni a természetben, de én egy kicsit megvílágosítom itt neked. Egyszer volt, hol nem volt, egyszer azt gondolták, hogy lehetne egy cím család (address family) (amit az "AF" jelképez az "`AF_INET`"-ben) amik támogathatnának pár protocolt amelyekre a sajár protokol családjuk által hivatkozhattak (amit a "PF" jelképez a "`PF_INET`"-ben). Ez nem történt meg. Ó jó. A helyes megoldás, hogy `AF_INET`-et használsz a `sockaddr_in` struktúrádban és `PF_INET`-et a `socket()` felé irányuló hívásaidban. De praktikusabb arról beszélni, hogy `AF_INET`-et haszánlhatsz mindenhol. És amit W. Richard Stevens csinál a könyvében, én is azt csinálom itt.

Vége, vége, vége, de miért is jó a socket? A válasz az, hogy önmagában tényleg nem jó, folytani kell az olvasást és csinálni pár rendszerhívást, hogy tapasztalatot szerezz róla.

4.2. `bind()` - Milyen porton vagyok?

Mikor van egy socketed, akkor társítanod kell a helyi géped egy portjához. (Ez rendszerint elvégzõdik, amikor bejövõ csatlakozásokhoz *listen()* (hallgatsz) kifele egy meghatározott porton - MUDok teszik ezt meg mikor megkérnek, hogy "telnetelj az x.y.z-re a 6969-es porton" (telnet to x.y.z port 6969).) A portszámot a kernel használja, hogy összekapcsoljon egy bejövõ csomagot egy meghatározott process socket leírójával. Ha csak egy `connect()` (kapcsolatot) akarsz létrehozni, akkor nem szükséges. Minden esetre olvasd el, csak úgy heccbõl.

Itt van a `bind()` rendszerhívás vázlata:

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

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

A *sockfd* a `socket()` álltal visszaadott socket fájlleíró. A *my_addr* egy mutató a `sockaddr` struktúrához amely információkat tartalmaz a címedrõl, név, port és IP cím. Az *addrlen* beállítható a `sizeof(struct sockaddr)` paranccsal.

Húha. Ez egy kicsit erõs elsõ nekifutásra. No de nézzünk egy példát:

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

#define MYPORT 3490

main()
{
      int sockfd;
      struct sockaddr_in my_addr;
      sockfd = socket(AF_INET, SOCK_STREAM, 0); // csináld meg a hibavizsgálatot!

      my_addr.sin_family = AF_INET; // host byte order
      my_addr.sin_port = htons(MYPORT); // short, network byte order
      my_addr.sin_addr.s_addr = inet_addr("10,12,110,57");
      memset(&(my_addr.sin_zero), '\0', 8); // a struktúra többi részét kinullázza

      // el ne felejtsd megcsinálni a hiba vizsgálatot a bind()-nek:
      bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
      .
      .
      .
```

Volna egy két megjegyzésem: a *my_addr.sin_port* Network Byte Orderben van. Szintúgy a *my_addr.sin_addr.s_addr* is. Egy másik dolog amire figyelni kell, hogy header fájlok különbözõek lehetnek rendszerrõl rendszerre. Hogy biztos legyél, vizsgáld át a helyi **man** oldalakat.

Végül, a bind() témájával kapcsolatban meg kell említenem, hogy azon folyamatok közül, melyek a saját IP címedet és/vagy portodat bevizsgálják, néhány automatizálható:

```
my_addr.sin_port = 0; // válassz egy szabad portot véletlenszerûen
my_addr.sin_addr.s_addr = INADDR_ANY; // haszáld az én IP címemet
```

Látod, ha a *my_addr.sin_port*-t nullára állítod, akkor a `bind()`-nak azt mondod, hogy válassza ki a portot saját maga neked. Hasonlóan a *my_addr.sin_addr.s_addr*-t `INADDR_ANY`-re állítva megkéred, hogy automatikusan állítsa be az IP címét a gépnek amelyen a folyamat fut.

Ha észre szoktál venni kisebb dolgokat, akkor biztosan láttad, hogy nem tettem az INADDR_ANY-t Network Byte Orderbe! Azt a rakoncátlan fajtámat! Akárhogy is, van egy bizalmas információm: `INADDR_ANY` valóban nulla! A nulla nulla a biteken is, még akkor is ha átrendezed a bájtokat. Akárhogy is, a nyelvmûvelõ tud mutatni egy hasonló dimenziót, ahol INADDR_ANY, mondjuk 12 és a kódom nem mûködik ott. Velem minden rendben:

```
my_addr.sin_port = htons(0); // válassz egy használaton kívüli portot véletlenszerûen
my_addr.sin_addr.s_addr = htonl(INADDR_ANY); // használd az én IP címem
```

Most olyan hordozható kódot csináltunk, hogy el sem hiszed. Csak arra akartam rámutatni, hogy bármennyi kódon is jöttél keresztül eddig, ne zavartasd magad futtatni az `INADDR_ANY`-t `htonl()` függvényen keresztül.

`bind()` szintén -1 értéket ad hiba esetén és az *errno*-ban tárolja a hiba értékét.

Egy másik dolog, amire figyelned kell amikor meghívod a bind() függvényt: ne menj egy bizonyos szám alá a portjaidnál. Minden port 1024 alatt le van foglalva (hacsak nem te vagy a superuser)! Ez felett bármelyik portszámot veheted egészen 65535-ig. (feltéve, hogy egy másik program már nem használja.)

Néha azt veheted észre, hogy megpróbálsz újrafuttatni egy szervert és a `bind()` elszáll, állítva, hogy "A cím már használatban van." Mit jelenthet ez? Nos, a socket egy kis darabja, amit csatlakoztattál, még mindig a kernelben játszik az idõvel, és foglalja a portodat. Várhatsz, amíg felszabadul (egy perc, vagy tovább), vagy teszel egy kódot a programba, ami megengedi, hogy újrahasználd a portot, úgy mint ez:

```
     int yes=1;

//char yes='1'; // Solaris esetében ezt kell használni

     // az idegekre menõ "Address already in use" hiba üzenet
     if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
          perror("setsockopt");
          exit(1);
     }
```

Még egy utolsó apró megjegyzés a `bind()` függvényrõl: vannak olyan esetek, amikor nem kell teljesen meghívni. Ha `connect()` függvénnyel kapcsolódsz egy távoli géphez, és nem lényeges, hogy mi a te helyi portszámod (ez van a **telnet** esetében is, ahol csak a távoli porttal kell törõdni), ekkor hívhatsz egyszerûen egy `connect()` függvényt, ez meg fogja vizsgálni, hogy a socket szabad-e és ha szükséges, a `bind()` függvényt használja egy használaton kívüli helyi porthoz.

4.3. `connect()` - Hey, te!

Most csináljunk úgy egy pár percig, mintha te egy telnet alkalmazás lennél. A te felhasználód azt parancsolja, hogy vegyed a socket fájlleírót. Te teljesíted és meghívod a `socket()` függvényt. Következõben a felhasználó azt mondja, hogy csatlakozz a "`10.12.110.57`"-es címre a "`23`"-as porton keresztül (a szabványos telnet port.) Hoppá! Most mit csinálsz?

Szerencséd van, program, olvasd át figyelmesen a `connect()` függvényrõl szóló részt - hogyan csatlakozzunk egy távoli hosthoz. Olvasd át õrült vágtában. Ne vesztegess több idõt!

A `connect()` hívás olyan mint a következõ:

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

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

A *sockfd* a mi baráti szomszédságunk socket fájlleírója, ami a `socket()` hívás visszatérõ értéke, a *serv_addr* egy `sockaddr` struktúra, amely tartalmazza a cél port és IP címét, az *addrlen* beállítható a `sizeof(struct sockaddr)`-al.

Ez a kezdet csinált egy kis ráérzést? Vegyünk egy példát:

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

     #define DEST_IP "10.12.110.57"
     #define DEST_PORT 23

     main()
     {
          int sockfd;
          struct sockaddr_in dest_addr; // ez fogja tárolni a cél címét

          sockfd = socket(AF_INET, SOCK_STREAM, 0); // csináld meg a hibavizsgálatát!

          dest_addr.sin_family = AF_INET; // host byte order
          dest_addr.sin_port = htons(DEST_PORT); // short, network byte order
          dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
          memset(&(dest_addr.sin_zero),'\0',8); // a struktúra többi részének kinullázása

          //ne felejtsd el megcsinálni a hibavizsgálatot
```

```
connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
.
.
.
```

Még egyszer, biztos legyél benne, hogy megvizsgáltad a `connect()` visszatérési értékét - hiba esetén -1 értéket ad, és beállítja az *errno* értékét.

Ugyancsak vedd észre, hogy nem hívtuk meg a `bind()` függvényt. Alapjában véve, nem törõdünk a helyi port számunkkal; csak azzal foglalkozunk, hogy hova megyünk (a távoli porttal). A kernel fog választani nekünk egy helyi portot, és a site ahova csatlakozunk, automatikusan megkapja ezt az információt tõlünk. Semmi veszõdség.

4.4. `listen()` - Felhívna valaki?

Ok, itt az ideje az irányváltásra. Mi van akkor, ha nem akarsz csatlakozni egy távoli hosthoz. Mondjuk, csak úgy heccbõl, bejövõ csatlakozásokra akarsz várni, és kezelni szeretnéd õket különbözõ utakon. A folyamat két lépésbõl áll: elõször te `listen()` (hallgatsz), aztán `accept()` (elfogadod) (lásd alább.)

A listen hívás egyszerûen világos, bár egy kis magyarázatot igényel:

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

A *sockfd* a szokásos socket fájlleíró a `socket()` rendszerhívásból. A *backlog* azoknak a csatlakozásoknak a száma, amelyek a bejövõ sorban meg vannak engedve. Ez mit jelent? Nos, a bejövõ csatlakozásoknak ebben a sorban kell várniuk amíg te el nem fogadod (`accept()`) õket (lásd lejjebb) és ez az a megszabott határ, amennyien sorba állhatnak. Pár rendszer alattomban ezt a számot 20-ra korlátozza; te valószínû ezt eltávolíthatod `5`-re vagy `10`-re állítva.

Újra, mint ahogyan az már szokásos a `listen()` hiba esetén -1-el tér vissza, és beállítja az *errno*-t hiba esetén.

Nos, ahogyan azt te valószínû el tudod képzelni, nekünk meg kell hívnunk a `bind()` függvényt, mielõtt mi meghívnánk a `listen()` függvényt, vagy a kernel egy véletlenül kiválasztott porton hallgattat minket. így ha bejövõ csatlakozásokre szeretnél figyelni, a rendszerhívások menete a következõ:

```
socket();
bind();
listen();
/* accept() ide jön */
```

Bent hagyom a kódban, minthogy az magától érthetõdõ.(A kód az `accept()` részében, alább, sokkal teljesebb.) Az igazi trükkös része ennek az egész sha-bang-nek az `accept()` meghívása.

4.5. `accept()` - "Köszi, hogy hívtál 3490-es port."

Kösd fel a gatyád, mert az `accept()` egy undorító fajta! Mi történik akkor, ha valaki nagyon-nagyon messzirõl próbál csatlakozni egy olyan portodra amire neked egy `listen()` függvényed figyel. A csatlakzozási kérvények sorba állnak és várakoznak, hogy az `accept()` függvénnyel elfogadd õket. Meghívod az `accept()` függvényt és megmondod neki, hogy vegye a feltételezett kapcsolatot. Ez egy vadi új socket fájlleíróval fog visszatérni, hogy egy egyedüli csatlakozáshoz használja! Ez így rendben is van, hirtelen *két socket fájlleíród* van egyért cserében! Az eredeti még mindig az adott portodat figyeli, az újabban létrehozott pedig készen áll hogy `send()` (küldjön) és `recv()` (fogadjon). Végre elérkeztünk ide is!

A hívás a következõ képpen néz ki:

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

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

A *sockfd* a `listen()` függvény socket leírója. Eddig még könnyû. Az *addr* általában egy `sockaddr_in` struktúrára mutató poiner lesz. Ez az, ahol a bejövõ csatlakozással kapcsolatos információ menni fog (és ennek segítségével meg tudod határozni, hogy melyik host melyik portrol hív téged). Az *addrlen* egy helyi egész (integer) változó amelyet a `sizeof(struct sockaddr_in)` függvénnyel kell beállítani, mielõtt a címét továbbküldjük az `accept()` függvénynek. Az accept azon a sok bájton kívül nem tesz többet az *addr*ba. Ha kevesebbet tesz bele, akkor az meg fogja változtatni az *addrlen* értékét kifejezve ezt.

Ki találtad már? Az `accept()` hiba esetén -1 értékkel tér vissza és az *errno*ba tárolja a hibaértékét.

Mint az elõbb is, itt egy darabka kódminta, hogy tudjál mit áttanulmányozni:

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

#define MYPORT 3490 // a port ahova a felhasználók csatlakozni fognak

#define BACKLOG 10 // mennyi elintézetlen kapcsolatot enged sorban állni

main()
{
      int sockfd, new_fd; // listen a sock_fd-n, új kapcsolat a new_fd-n
      struct sockaddr_in my_addr; // az én címinformációm
      struct sockaddr_in their_addr; // a csatlakozó fél címinformációja
      int sin_size;

      sockfd = socket(AF_INET, SOCK_STREAM, 0); // a hiba figyelése a te dolgod!

      my_addr.sin_family = AF_INET; // host byte order
      my_addr.sin_port = htons(MYPORT); // short, network byte order
      my_addr.sin_addr.s_addr = INADDR_ANY; // automatikusan kitölti az én IP-mel
      memset(&(my_addr.sin_zero), '\0', 8); // kinullázza struktúra többi részét

      // El ne felejtsd megcsinálni a hibavizsgálatát ezeknek a hívásoknak:
      bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

      listen(sockfd, BACKLOG);

      sin_size = sizeof(struct sockaddr_in);
      new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
      .
      .
      .
```

Mégegyszer, jegyezd meg, hogy a *new_fd* socket leírót fogjuk használni az összes `send()` és `recv()` hívásokhoz. Hogyha te mindig csak egy egyedüli kapcsolatot létesítesz, akkor a `close()` függvénnyel befejezheted a sockfd figyelését a porton azért, hogy megakadályozd több kapcsolat létrejöttét ugyanazon a porton, ha úgy kívánnád.

4.6. `send()` és `recv()` - Szólj hozzám bébi!

Ez a két függvény a stream socketen vagy kapcsolt datagram socketen keresztüli kommunikációra szolgál. Abban az esetben, ha kapcsolat nélküli datagram socketet akarsz használni, akkor szükséges lesz elolvasnod a `sendto()` és `recvfrom()` függvényekrõl szóló fejezetet, lentebb.

A `send()` hívás:

```
int send(int sockfd, const void *msg, int len, int flags);
```

A *sockfd* az a socket leíró ahova adatokat szeretnél küldeni (akár a `socket()` vagy akár az `accept()` által adott adat.) Az *msg* egy mutató arra az adatra, amit küldeni szeretnél, a *len* pedig az adat hossza bájtokban. A *flags* értékét csak állítsd 0-ra. (Lásd. a `send()` man oldalát további flag információért.)

Pár minta kód lehet:

```
char *msg = "Beej was here!";
int len, bytes_sent;
.
.
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
.
.
.
```

A `send()` függvény a kiküldött bájtok számával tér vissza - *ez lehet kevesebb annál, mint amennyit mondtál neki, hogy küldjön!* Néha lehet, hogy egy jó adag adatot akarsz vele elküldetni, és szegény nem tudja lekezelni a kérésedet. Annyi adatot fog elküldeni, amennyit bír, és rádbízza, hogy küldd el a maradékot késõbb. Emlékezz rá, hogy ha a `send()` által visszaadott érték nem egyezik meg a *len* értékével, akkor a te dolgod elküldeni a string maradék részét. A jó hír az, hogy ha a csomag kicsi (kisebb mint pl. 1K), akkor az valószínû egyben el lesz küldve. Mint ahogy mindig, hiba esetén -1 a visszatérési érték, és az *errno* beállítódik a hiba számára.

A `recv()` hívás nagyon sok vonatkozásban egyszerû:

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

A *sockfd* az a socketleíró, ahonnan olvasunk, a *buf* a buffer amibe információkat olvasunk be, a *len* a buffer maximális hossza, a *falgs*-et pedig megint 0-ra állíthatjuk. (Lásd. a `recv()` man oldalát további flag információkért.)

A `recv()` az aktuálisan a bufferbe olvasott bájtok számával tér vissza, vagy -1 értékkel hiba esetén (az *errno* is beállítódik ilyenkor természetesen.)

Várj! A `recv()` 0 értékkel is visszatérhet. Ez csak egy dolgot jelenthet: a távoli site befejezte az irányodban levõ kapcsolatát! A `recv()` ezzel a 0-val jelzi, hogy a kapcsolatbontás bekövetkezett.

Hát ez elég könnyû volt, nemigaz? Most már tudsz adatokat küldeni és fogadni a stream socketen keresztül! Hûha, te már egy Unix hálózati programozó vagy!

4.7. `sendto()` és `recvfrom()` - Beszélj hozzám, DGRAM-stílus

"Ez mind szép és remek," - hallom, ahogy mondod, - "de mire megyek ezzel a kapcsolatnélküli datagram socketeknél?"
Semmi gond amigo. Meg van a megoldás.

Mivel a datagram socketek nem csatlakoznak egy távoli hosthoz, tippelj, melyik darab információra lesz szükségünk, mielõtt elküldjük a csomagot? Így van! A cél címére! Itt a megoldás:

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct
sockaddr *to, int tolen);
```

Ahogy láthatod, ez a hívás alapjában véve ugyanaz, mint a `send()` kiegészítve két apró információval. A *to* egy `sockaddr` struktúrára mutató pointer (amely lehet egy `sockaddr_in` struktúrára mutató is és a legvégén átkasztolod) amely tartalmaz egy IP címet és portot. A *tolen* egyszerûen beállítható a `sizeof(struct sockaddr)` függvényhívással.

Ugyanúgy, mint a `send()` esetében, a `sendto()` is az aktuálisan elküldött bájtok számával térvissza (ami itt is lehet hogy kevesebb, mint amennyit elküdeni szándékoztunk!), vagy -1 hiba esetén.

A `recv()` és `recvfrom()` pedig egyszerûen megegyezik. A `recvfrom()` vázlata:

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct sockaddr
*from, int *fromlen);
```

Ennek is a legtöbb mezõje megegyezik a `recv()` függvényével. A *from* egy `sockaddr` struktúrára (amely az eredeti gép IP címével és portszámával van kitöltve) mutató pointer. A *fromlen* egy helyi int típusra mutató pointer, amit a sizeof(struct sockaddr) függvénnyel kell inicializálnunk. Amikor a függvény visszatér, akkor a *fromlen* fogja tartalmazni a *from*ban tárolt cím hosszát.

A `recvfrom()` függvény a kapott bájtok számát fogja visszaadni, illetve hiba esetén -1 értéket (az *errno* is beállítódik értelemszerûen.)

Emlékezz, ha a `connect()` függvénnyel csatlakozol egy datagram sockettel, akkor egyszerûen használhatod a `send()` és `recv()` függvényeket az összes tranzakciódhoz. A socket önmaga, még mindig egy datagram socket és a csomagok még mindig UDP-t használnak, de a socket interfész automatikusan fogja megadni neked a cél és a forrás információkat.

4.8. `close()` és `shutdown()` - Viszlát!

Húha! Egész nap adatokat küldtél és fogadtál (`send()`/`recv()`) és mindent elintéztél. Készen állsz, hogy befejezd a kapcsolatot a socket leírodban. Ez elég könnyû. Máris használhatod a Unix szokásos fájlleírójának a `close()` függvényét:

```
close(sockfd);
```

Ez megakadályozza socket további írását és olvasását. Ezután ha valaki olvasni vagy írni szeretne a socketre a távoli oldalon, egy hibát fog visszakapni.

Ha egy kicsit jobban szeretnéd szabályozni, hogy a socket hogyan zárjon be, használhatod a `shutdown()` függvényt erre. Ez lehetõséget ad, hogy egy bizonyos irányba szakítsd meg a kommunikációt, vagy mindkettõben (éppen, mint ahogy a `close()` teszi.) Vázlata:

```
int shutdown(int sockfd, int how);
```

A *sockfd* az a socket fájlleíró amit be akarsz zárni, és a *how* a következõk egyike:

- `0` - További fogadások tiltása
- `1` - További küldések tiltása

- `2` - További küldések és fogadások tiltása (mint a `close()` esetében)

A `shutdown()` 0 értéket ad vissza sikeres végrehajtás esetén, és -1 értéket hiba esetén (természetesen az *errno* beállítása sem maradhat el.)

Ha a `shutdown()` függvényt méltóztatsz használni egy nem csatlakozott datagram socketnél, akkor az egyszerûen hozzáférhetetlenné teszi a socketet további küldés (`send()`) és fogadás (`recv()`) híváshoz (emlékezz, hogy ezeket akkor használhatod, ha `connect()` függvénnyel használod a datagram socketet.)

Fontos még megjegyezni, hogy a `shutdown()` igazából nem zárja le a fájlleírót, éppen csak megváltoztatja a használhatóságát. Amennyiben fel akarsz szabadítani egy socketleírót, a `close()` függvényt kell használnod.

Ez van. Kész-passz.

4.9. `getpeername()` - Ki vagy?

Ez a függvény nagyon egyszerû.

Annyira könnyû, hogy majdnem nem is adtam neki külön fejezetet. No de azért mégis itt van.

A `getpeername()` függvény megmondja, hogy ki van a stream socket kapcsolat túloldalán. A használata:

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

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

A *sockfd* a kapcsolatban lévõ stream socket leírója, az *addr* egy `sockaddr`(vagy `sockaddr_in`) struktúrára mutató pointer, amely a kapcsolat túl oldaláról tárol információkat. Végül az *addrlen* egy int típusra mutató pointer, amit a `sizeof(struct sockaddr)` függvénnyel kell inicializálni.

Hiba esetén -1 értéket ad vissza függvény és beállítja az *errno* értékét.

Miután egyszer már megkaptad a címét, használhatod az `inet_ntoa()` vagy a `gethostbyaddr()` függvényeket több információ kiiratásához. Nem, nem. A login nevét nem kaphatod meg. (Jól van, jól van. Ha a másik számítógép egy azonosított daemont futtat, akkor akár ez is lehetséges. Ez, akárhogyis, ennek a domunetumnak a tárgykörén kívül esik. További információért nézz bele az RFC-1413[10] hivatkozásba.)

4.10. `gethostname()` - Ki vagyok?

A `getpeername()` függvénynél már csak a `gethostname()` egyszerûbb. Annak a számítógépnek a nevével tér vissza, amelyiken a programot futtatod. Ez a név használható ezek után a `gethostbyname()` (lásd lejjebb) függvény paramétereként, hogy visszakapd a helyi számítógéped IP címét.

Mi tudna még több örömöt okozni? Hát mondjuk, most eszembe jutott egy-kettõ, de azok nem kapcsolódnak a socket programozáshoz. Minden esetre, a lényeg a következõ:

```
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

A paraméterek egyszerûek: a *hostname* egy karaktertömbre mutató pointer, ami a függvény visszatérése után a hostnevet fogja tartalmazni, a *size* pedig a *hostname* tömbjének a mérete bájtokban.

Hiba esetén -1 ellenben 0, és természetesen hibánál az *errno* is belövi a megfelelõ értéket, mint ahogy az már megszokott.

4.11. DNS

(Amire te azt mondod "whitehouse.gov", én azt mondom "198.137.240.92")

Abban az esetben, ha nem tudnád, hogy mi is az a DNS, nos a "Domain Name Service"-t takarja. Dióhélyban annyi, hogy te megadod az emberileg olvasható címét a sitenak, õ pedig az IP címét fogja neked adni (így használhatod a `bind()`, `connect()`, `sendto()`, stb. függvényekkel, arra amire csak akarod.) Ezúton, ha valaki a következõt írja be:

```
$ telnet whitehouse.gov
```

a **telnet** kitalálja, hogy a "198.137.240.92"-re kell csatlakoznia (`connect()`).

De ez hogyan is mûködik? A `gethostbyname()` függvényt fogod erre használni:

```
#include <netdb.h>

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

Ahogy látod, ez egy `hostent` nevû struktúrára mutató pointert ad vissza. A struct hostent kinézete a következõ:

```
struct hostent {
     char *h_name; char **h_aliases; int h_addrtype; int h_length; char
     **h_addr_list;
};
#define h_addr h_addr_list[0]
```

Most pedig következzen a hostent struktúra mezõinek leírása:

- *`h_name`* - A host hivatalos neve.
- *`h_aliases`* - A host alneveinek egy NULL-végzõdésû tömbje
- *`h_addrtype`* - A visszatérõ cím típusa; általában *`AF_INET`*.
- *`h_length`* - A cím hossza bájtokban.
- *`h_addr_list`* - Egy 0-végzõdésû tömbje a hosthoz tartozó hálózati címeknek. A host címek Network Byte Orderben vannak.
- *`h_addr`* - Az elsõ cím a `h_addr_list`-ben.

A `gethostbyname()` függvény egy kitöltött `hostent` struktúrára mutató pointert ad vissza, vagy pedig hibaesetén NULL értéket. (De az *errno* nem állítódik be, helyett a *h_errno* állítódik be. Lásd `herror()` függvényt lejjebb.)

De ezt hogyan is használják? Néha (ahogyis mi a kézikönyvekben találtuk), az információ odaömlesztése az olvasóhoz nem elég. Ennek a függvénynek a használata könnyebb, mint ahogyan az látszik.

Itt van egy példaprogram[11]:

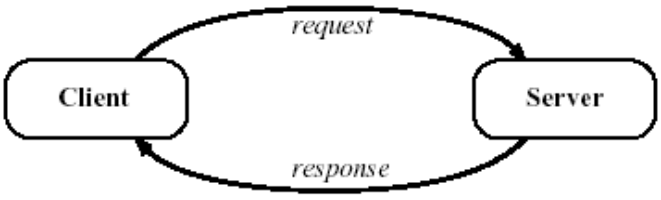
```
/*
** getip.c - egy hostnév keresõ demo
*/

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

int main(int argc, char *argv[])
{
      struct hostent *h;

      if (argc != 2) { // a parancssor hibavizsgálata
            fprintf(stderr,"usage: getip address\n");
            exit(1);
      }

      if ((h=gethostbyname(argv[1])) == NULL) { // a host információt megkapja
            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;
}
```

A `gethostbyname()` függvényhez nem használhatod a `perror()` függvényt a hibaüzenet kiiratásához (mivel az *errno* nincs itt használatban). Helyette hívd meg a `herror()` függvényt.

Ez elég szókimondó. Csak egyszerûen beadod a gép nevét ("whitehouse.gov") tartalmazó információt a `gethostbyname()` függvénynek, és aztán csak markold ki az infot a visszaadott hostent struktúrából.

Az egyedüli lehetséges hátborzongató dolog az IP cím kiiratása lehet. *h->h_addr* egy `char*`, de az `inet_ntoa()` függvény egy `in_addr` struktúrát szeretne kapni. Így én a *h->h_addr*-ot `in_addr*` struktúrára kasztoltam, majd újrahivatkoztam rá, hogy megkapjam az adatot.

2. ábra - Kliens-Szerver kölcsönhatás:



5. Kliens-Szerver háttér

Ez egy kliens-szerver világ öcsi! A hálózati életben szinte minden esetben kliens folyamatok szerver folyamatoknak beszélnek, és fordítva. Vegyük a **telnet**-et példaként. Amikor te a 23-as porton egy távoli hostra telnetelsz (kliens), egy azon a hoston lévõ program (hívd telnetd-nek, a szerver) fakad életre. Ez kezeli

a bejövõ telnet csatlakozásokat, egy login(bejelentkezõ) promptot rak ki neked, stb.

Az információcsere a kliens és a szerver között a 2. ábrán látható.

Megjegyezzük, hogy a kliens-szerver pár `SOCK_STREAM`, `SOCK_DGRAM`, vagy bármi máson tud egymással beszélni (egész addig, amíg hasonló dolgokról beszélnek.) Pár jó példa a kliens-szerver párokra a **telnet/telnetd**, **ftp/ftpd**, vagy a **bootp/bootpd**. Akárhányszor **ftp**-t használsz, a távoli program az **ftpd** lesz, ami kiszolgál téged.

Gyakran csak egy szerver program van egy gépen és ez összetetten kezeli a különbözõ klienseket a `fork()` használatával. Az alap eljárás a következõ: a szerver egy kapcsolatra vár, `accept()` azaz elfogadja azt, aztán az elágaztatásra a `fork()` függvényt használja, ami segítsgével egy gyerek folyamat fogja kezelni a kapcsolatot. Ez az, amit a mi minta szerverünk is tesz a következõ részben.

5.1. Egy egyszerû Stream szerver

Az össz dolog, amit ez a szerver csinál, az az, hogy kiír egy "`Hello, World!\n`" stringet egy stream kapcsolaton keresztül. Az össz dolgod, amit a tesztelés érdekében tenned kell, hogy elindítod ezt a szervert egy ablakban, egy másikban pedig rátelnetezel:

```
$ telnet remotehostname 3490
```

ahol a remotehostname annak a gépnek a neve, amin éppen dolgozol.

A szerver kód[12] a következõ:

```
/*
** server.c - egy stream socket szerver demo
*/

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

#define MYPORT 3490 // a port használói ide fognak csatlakozni

#define BACKLOG 10 // mennyi kezeletlen kapcsolat várakozhat sorban

void sigchld_handler(int s)
{
     while(wait(NULL) > 0);
}

int main(void)
{
     int sockfd, new_fd; // figyelés-hallgatás a sock_fd-n, új kapcsolat a new_fd-n
     struct sockaddr_in my_addr; // a saját címinformációm
     struct sockaddr_in their_addr; // a csatlakozó címinformációja
     int sin_size;
     struct sigaction sa;
     int yes=1;

     if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
          perror("socket");
          exit(1);
```

```
        }

        if (setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
             perror("setsockopt");
             exit(1);
        }

        my_addr.sin_family = AF_INET; // host byte order
        my_addr.sin_port = htons(MYPORT); // short, network byte order
        my_addr.sin_addr.s_addr = INADDR_ANY; // automatikusan kitölti az IP-mel
        memset(&(my_addr.sin_zero), '\0', 8); // nulla a struktúra többi részében

        if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1) {
             perror("bind");
             exit(1);
        }

        if (listen(sockfd, BACKLOG) == -1) {
             perror("listen");
             exit(1);
        }

        sa.sa_handler = sigchld_handler; // levág minden halott folyamatot
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = SA_RESTART;
        if (sigaction(SIGCHLD, &sa, NULL) == -1) {
             perror("sigaction");
             exit(1);
        }

        while(1) { // main accept() loop
             sin_size = sizeof(struct sockaddr_in);
             if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size))
             == -1) {
                  perror("accept");
                  continue;
             }
             printf("server: got connection from %s\n",inet_ntoa(their_addr.sin_addr));
             if (!fork()) { // ez a gyerek folyamat
                  close(sockfd); // a gyereknek nincs szüksége a listázóra
                  if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
                       perror("send");
                  close(new_fd);
                  exit(0);
             }
             close(new_fd); // a szülõnek nincs szüksége erre
        }

        return 0;
   }
```

Ha kíváncsi vagy, (úgyérzem) a mondattani átláthatóság kedvéért van a kód egy nagy `main()` függvényben. Ha úgy érzed, szabdald kisebb függvényekre.

(Ezenfelül ez az egész `sigaction()` dolog új lehet neked - ez rendben is van. A kód, ami a meghallt folyamatok kiírtásáért felel, a `fork()` függvény egy gyerek folyamat távozásának(exit) tûnik. Ha túl sok zombi folyamatot csinálsz, és nem írtod ki õket, akkor a rendszergazdád elég mérgesen fog rádnézni.)

A szerver álltal szolgáltatott adatot a következõ részben bemutatott kliens tudja megkapni.

5.2. Egy egyszerû Stream kliens

Ez a srác sokkal könnyebb eset, mint a szerver haverja. Annyit csinál összesen, hogy kapcsolódik a parancssorban megadott 3490-es portszámon keresztül. Aztán megkapja a sztringet, amit a szerver küldött.

A kliens forráskódja[13]:

```
/*
```

```
** client.c - egy stream socket kliens demo
*/

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

#define PORT 3490 // az a port, ahova a kliens csatlakozni fog

#define MAXDATASIZE 100 // az egyszerre kapható bájtok maximális értéke

int main(int argc, char *argv[])
{
      int sockfd, numbytes;
      char buf[MAXDATASIZE];
      struct hostent *he;
      struct sockaddr_in their_addr; // a csatlakozó címinformációja

      if (argc != 2) {
            fprintf(stderr,"usage: client hostname\n");
            exit(1);
      }

      if ((he=gethostbyname(argv[1])) == NULL) { // megkapja a hostinformációt
            perror("gethostbyname");
            exit(1);
      }

      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
            perror("socket");
            exit(1);
      }

      their_addr.sin_family = AF_INET; // host byte order
      their_addr.sin_port = htons(PORT); // short, network byte order
      their_addr.sin_addr = *((struct in_addr *)he->h_addr);
      memset(&(their_addr.sin_zero), '\0', 8); // kinullázza a struktúra többi részét

      if (connect(sockfd, (struct sockaddr *)&their_addr,sizeof(struct sockaddr)) ==
      -1) {
            perror("connect");
            exit(1);
      }

      if ((numbytes=recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
            perror("recv");
            exit(1);
      }

      buf[numbytes] = '\0';

      printf("Received: %s",buf);

      close(sockfd);

      return 0;
}
```

Jegyezzük meg, hogy ha nem indítod el a szervert a kliens futtatása elõtt, akkor a `connect()` függvény "Connection refused"(csatlakozás elutasítva)-ot ad vissza. Ez nagyon hasznos.

5.3. Datagram socketek

Én tényleg nem szeretnék sokat beszélni itt, így bemutatok egy pár minta programot: `talker.c` és `listener.c`.

A **listner** egy gépen ül és bejövõ csomagokra vár a 4950-es porton. A **talker** egy csomagot küld ugyanarra a portra,a meghatározott gépen, a csomag bármit tartalmazhat, amit a felhasználó beír a parancssorba.

Jöjjön a `listner.c`[14] forráskódja:

```
/*
** listener.c - egy datagram sockets "szerver" demo
*/

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

#define MYPORT 4950 // az a port ahova a felhasználó kapcsolódni fog

#define MAXBUFLEN 100

int main(void)
{
     int sockfd;
     struct sockaddr_in my_addr; // my address information
     struct sockaddr_in their_addr; // connector's address information
     int addr_len, numbytes;
     char buf[MAXBUFLEN];

     if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
          perror("socket");
          exit(1);
     }

     my_addr.sin_family = AF_INET; // host byte order
     my_addr.sin_port = htons(MYPORT); // short, network byte order
     my_addr.sin_addr.s_addr = INADDR_ANY; // automatikusan kitöltõdik a saját
     IP-mel
     memset(&(my_addr.sin_zero), '\0', 8); // kinullázza a struktúra többi részét

     if (bind(sockfd, (struct sockaddr *)&my_addr,sizeof(struct sockaddr)) == -1) {
          perror("bind");
          exit(1);
     }

     addr_len = sizeof(struct sockaddr);
     if ((numbytes=recvfrom(sockfd,buf, MAXBUFLEN-1, 0,(struct sockaddr
     *)&their_addr, &addr_len)) == -1) {
          perror("recvfrom");
          exit(1);
     }

     printf("got packet from %s\n",inet_ntoa(their_addr.sin_addr));
     printf("packet is %d bytes long\n",numbytes);
     buf[numbytes] = '\0';
     printf("packet contains \"%s\"\n",buf);

     close(sockfd);

     return 0;
}
```

Vedd észre, hogy `socket()` hívásunkban végülis `SOCK_DGRAM`-ot használtunk. Azt is vedd észre, hogy nincs szükség a `listen()` és az `accept()` használatára. Ez az egyik mellékes szépsége a kapcsolat nélküli datagram socketnek!

Most pedig jöjjön a `talker.c`[15] forráskódja:

```
/*
** talker.c - egy datagram "kliens" demo
*/

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

#define MYPORT 4950 // a port, ahova a felhasználók csatlakozni fognak

int main(int argc, char *argv[])
{
     int sockfd;
     struct sockaddr_in their_addr; // a csatlakozó címinformációja
     struct hostent *he;
     int numbytes;

     if (argc != 3) {
          fprintf(stderr,"usage: talker hostname message\n");
          exit(1);
     }

     if ((he=gethostbyname(argv[1])) == NULL) { // veszi a hosztinformációt
     perror("gethostbyname");
     exit(1);
}

if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
     perror("socket");
     exit(1);
}

their_addr.sin_family = AF_INET; // host byte order
their_addr.sin_port = htons(MYPORT); // short, network byte order
their_addr.sin_addr = *((struct in_addr *)he->h_addr);
memset(&(their_addr.sin_zero), '\0', 8); // kinullázza a struktúra maradék részét

if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0,(struct sockaddr
*)&their_addr, sizeof(struct sockaddr))) == -1) {
     perror("sendto");
     exit(1);
}

printf("sent %d bytes to %s\n", numbytes,inet_ntoa(their_addr.sin_addr));

close(sockfd);

return 0;

}
```

És ez van, ezt kell szeretni! Futtasd a *listener*t pár gépen, aztán futtasd a *talker*t egy másikon. Figyeld, hogyan kommunikálnak!

Van még egy kicsi részlet, amit már többször említettem a múltban: kapcsolatot is létesítõ datagram socketek. Ezt itt kellene megtárgyalnunk, mert ez a datagram fejezete a dokumentumnak. Mondjuk, hogy a **talker** maghívja a `connect()` függvényt és meghatározza a **listener** címét. Ettõl a ponttól kezdve a **talker** lehet, hogy csak abban az irányban fogadhat és küldhet amelyet a `connect()` meghatározott. Ebbõl az okból kifolyólag nem kell használnod a `sendto()` és `recvfrom()` függvényeket; egyszerûen használhatod a `send()` és `recv()` függvényeket.

6. Kicsit fejlettebb teknikák

Ezek nem is *igazán* fejlettebbek, de mégis kivezetnek bennünket a legtöbb alapabb szintbõl, amiket már lefektettünk. Abban az esetben, ha ilyen távolra jutottál, fel kell ismerned, hogy már becsületesen kímûvelted

magad a Unix hálózati programozásának alapjaiból! Gratulálok!

Így induljunk el a socket programozásról szóló szép új, számunkra még rejtett világok felé.
Lássunk neki!

6.1. Blokkolás

Blokkolás. Már hallottál róla - de mi a bánat is ez? Dióhéjban, a "block" a technika szakmai nyelve a "sleep"(alvás)nak. Már valószínû észrevetted, hogy amikor a **listener** programot futtatod, akkor az addig ott ül, amíg egy csomag érkezik. Ez történt akkor is amikor a `recvfrom()` függvényt meghívtad és nem volt adat, és így a `recvfrom()` függvény addig úgy mondjuk "block"olt (ez a sleep itt) amíg valamilyen adat nem érkezett.

Sok függvény blokkol. Az `accept()` blokkol. A `recv()` függvények minden típusa blokkol. Az ok amiért ezt megtehetik, az az, hogy meg van nekik ez engedve. Amikor elõször megcsináltad a socket leírót a `socket()` függvénnyel, akkor azt a kernel blokkolásra állította. Ha nem szeretnéd, hogy a socket blokkolva legyen, akkor egy `fcntl()` függvényhívást kell intézned:

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

A socketet nem-blokkolásra állítva, hatékonyan "kinyírhatod" a socketet. Ha megpróbálsz olvasni egy nem-blokkolt socketrõl ami nem kap adatot, mivel nincs neki megengedve a blokkolás - akkor -1 értékkel tér vissza és az *errno* `EWOULDBLOCK` értékre állítódik.

Általában azt mondják, hogy a kinyírás e-típusa nem egy túl jó ötlet. Ha a programodat a socketen lévõ adatok szorgalmas figylésére utasítod, akkor egy halom CPU idõt elpazarolsz vele és ez már ugye emiatt is stílustalan megoldás. Egy sokkal elegánsabb megoldás az olvasásra váró adatok figyelésére a `select()` függvény, amirõl a következõ részben olvashatsz.

6.2. *select()* - Szinkron I/O Multiplexelés

Ez a függvény kissé furcsa, de nagyon hasznos. Vegyük a következõ szituációt: te egy szerver vagy és bejövõ kapcsolódásokra olyannyira szeretnél figyelni, mint a meglévõ kapcsolatokból való olvasásra.

Semmi gond, mondod, csak egy `accept()` és egy halom `recv()` függvény és kész. Nem elég gyors kisfiam! Mi van akkor, ha egy `accept()` híváson blokkolsz? Mégis hogyan fogod a `recv()` függvényt használni adatok fogadására ugyanabban az idõben? "Használjunk nem-blokkolt socketeket!" Semmi esetre sem! Nem szeretnénk a CPUval kicseszni. Mi legyen akkor?

A `select()` adja neked az erõt, hogy felügyelj több socketet ugyanabban az idõben. Megmondja neked, hogy melyikek állnak készen olvasásra, melyikek írásra, és melyikek dobtak kivételt, - mindent amit csak tudni szeretnél.

Minden további nélkül jöjjön a `select()` felvázolása:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
```

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

A függvény felügyelõk fájlleírók halmazai; nevezetesen *readfds*, *writefds* és *exceptfds*. Ha azt szeretnéd látni, hogy tudsz-e olvasni a standard bemenetrõl és pár socket leíróból,*sockfd*, csak addj a fájlleíróhoz 0-át és a *readfds* halmazhoz a *sockfd*t. A *numfds* paramétert a legmagasabb fájlleíró értéke + 1 -re kell állítani. Ebben a példában, *sockfd+1*re kell állítani, amiatt mert ez kétségtelenül nagyobb mint a standard bemenet (0) értéke.

Amikor a `select()` függvény visszatér, a *readfds* módosítva lesz, hogy megmutassa, hogy az általad kiválasztott fájlleírók közül melyek állnak készen olvasásra. Az `FD_ISSET()` makróval le is tudod tesztelni, - alább.

Mielõtt jobban tovább mennék, megmondom hogy hogyan manipulálhatod ezeket a halmazokat. Mindegyik halmaznak `fd_set` a típusa. A következõ makrók ezen a típuson mûködnek:

- `FD_ZERO(fd_set *set)` - egy fájlleíró tömböt töröl
- `FD_SET(int fd, fd_set *set)` - hozzáadja *fd*-t a halmazhoz
- `FD_CLR(int fd, fd_set *set)` - eltávolítja az *fd*-t a halmazból
- `FD_ISSET(int fd, fd_set *set)` - letesztelhetjük, hogy az *fd* a halmazban van-e

Végezetül mi ez a különös `struct timeval`? Nos, néha nem akarsz örökké várni, hogy valaki küldjön neked valami adatot. Mondjuk minden 96-odik másodpercben szeretnéd kiiratni a terminálra, hogy "még megyek...", annak ellenére, hogy semmi sem történt. Ez a time struktúra lehetõséget ad idõlejárati periódusok megszabására. Amikor az idõ letelt és a `select()` még mindig nem talált egy készen álló fájlleírót sem, vissza fog térni hogy folytathasd a feldolgozást.

A `struct timeval` struktúra a következõ mezõkkel rendelkezik:

```
struct timeval {
     int tv_sec; // másodpercek
     int tv_usec; // mikromásodpercek
};
```

Csak állítsuk a *tv_sec* értékét a várakozásra szánt másodpercek számára, és a *tv_usec* értékét a várakozásra szánt mikromásodpercek számára. Igen, az mikromásodperc, nem pedig ezredmásodperc. 1,000 micormásodperc van egy ezredmásodpercben, és 1,000 ezredmásodperc egy másodpercben. Ebbõl következõleg 1,000,000 mikromásodperc van egy percben. Akkor miért "usec" az elnevezése? Az "u" ugy néz ki, mint a görög mû betû, amit a "micro" jelölésére használunk. Mindenesetre, amikor a függvény visszatér, a *timeout* *megmutathatja* a visszalévõ idõt. Ez attól függ, hogy milyen Unixot használsz.

Jajj! Van egy micromásodpercekre bontható idõzítõnk! Nos, ne alapozz erre. A szabványos Unix idõegység 100 ezredmásodperc körül van, így legalább addig kell várnod, eltekintve attól, hogy milyen ici-picire is állítottad a `timeval` struktúrád értékét.

Másik lényeges dolog: ha a `struct timeval` mezõinek értékét `0`-ra állítod, akkor a `select()` azonnal lejár, ebbõl kifolyólag kinyírja a halmazaidban lévõ fájlleírókat. Ha a *timeout* paraméterét NULLra állítod, akkor sosem fog lejárni az idõ, és egészen addig vár, amíg az elsõ fájlleíró készen nem áll. Végezetül, ha nem törõdsz azzal, hogy egy meghatározott ideig várjon, akkor a `select()` hívásban közvetlenül NULLra állíthatod az értéket.

A következõ kódrészlet[16] 2.5 másodpercig vár, hogy a standard bemeneten valami történjen:

```
/*
** select.c - a select() demo
*/

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define STDIN 0 // fájlleíró a szabványos bemenethez

int main(void)
{
     struct timeval tv;

     fd_set readfds;

     tv.tv_sec = 2;
     tv.tv_usec = 500000;

     FD_ZERO(&readfds);
     FD_SET(STDIN, &readfds);

     // ne foglalkozz a writefds és a exceptfdssel:
     select(STDIN+1, &readfds, NULL, NULL, &tv);

     if (FD_ISSET(STDIN, &readfds))
          printf("A key was pressed!\n"); // egy billetyû le lett nyomva
     else
          printf("Timed out.\n"); // lejárt az idõ

     return 0;
}
```

Ha egy sorbufferelt terminálon dolgozol, akkor csak a RETURN billentyûre lesz érzékeny, különben minden másra megy tovább a lejárati idõ.

Most amire talán gondolsz, hogy ez egy jó módszer adatok várására datagram socketen - és igazad van: az *lehet*. Pár Unix tudja a selectet erre használni, pár pedig nem. Meg kellene nézned, hogy mit ír a helyi man oldalad, abban az esetben, ha arra akarod használni.

Pár Unix változtatja az idõt a `struct timeval` struktúrában, hogy megmutassa a lejárat elõtt visszalévõ idõt. De vannak amelyek ezzel nem rendelkeznek. Ne támaszkodj erre a módszerre, ha portablilis programot szeretnél írni. (Használd a `gettimeofday()` függvényt, ha nyomon akarod követni az idõ telését. Ez elég húzós, tudom, de ez a módja.)

Mi történik akkor, amikor a socket az írás beállításakor bezárja a kapcsolatot? Nos, ebben az esetben a `select()` azzal a socket leíróval tér vissza amely "ready to read"-re van állítva. Amikor ebben az idõben csinálsz `recv()` hívást arról, a `recv()` 0-val tér vissza. Ebbõl tudod, hogy a kliens bezárta a kapcsolatot. (??????)

Még egy érdekes megjegyzés a `select()` függvényrõl: ha egy hallgató(`listen()`) socketed van, akkor ezt használhatod az új kapcsolatok figyelésére, úgy hogy annak a socketnek a fájlleíróját belerakod a *`readfds`* halmazba.

És ez kedves barátom, csak egy gyors áttekintés volt a mindenható `select()` függvényrõl.

De közkívánatra itt van egy nagyon részletes példa. Sajnos lényeges a különbség ez és a fentebbi piszkosan egyszerû példa között. Hogy lásd, olvashatsz egy leírást, ami követi a példát.

Ez a program[17] úgy mûködik, mint egy multi-felhasználós chat szerver. Indítsd el egy ablakban, majd **telnet**elj rá("**telnet hostname 9034**") többszörösen más ablakokból. Amikor beírsz valamit egy **telnet** ablakban, akkor az kiiródik a többin.

```
/*
** selectserver.c - a cheezy multiperson chat server
*/

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

#define PORT 9034 // a port, amin figyelünk

int main(void)
{
     fd_set master; // master fájlleíró lista
     fd_set read_fds; // átmeneti(temp) fájlleíró lista a select()-hez
     struct sockaddr_in myaddr; // szerver cím
     struct sockaddr_in remoteaddr; // kliens cím
     int fdmax; // maximum száma a fájlleíróknak
     int listener; // a hallgatózó socket leíró
     int newfd; // frissen elfogadott (accept()ed) socket leíró
     char buf[256]; // buffer a kliens adatoknak
     int nbytes;
     int yes=1; // a setsockopt()-hoz SO_REUSEADDR, alább
     int addrlen;
     int i, j;

     FD_ZERO(&master); // törli a master és átmeneti(temp) halmazokat
     FD_ZERO(&read_fds);

     // vegyük a listener-t
     if ((listener = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
          perror("socket");
          exit(1);
     }

     // szabaduljunk meg a bosszantó "address already in use(cím már használatben)"
     hiba üzenettõl
     if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes,sizeof(int)) == -1)
     {
          perror("setsockopt");
          exit(1);
     }

     // bind
     myaddr.sin_family = AF_INET;
     myaddr.sin_addr.s_addr = INADDR_ANY;
     myaddr.sin_port = htons(PORT);
     memset(&(myaddr.sin_zero), '\0', 8);
     if (bind(listener, (struct sockaddr *)&myaddr, sizeof(myaddr)) == -1) {
          perror("bind");
          exit(1);
     }

     // listen
     if (listen(listener, 10) == -1) {
          perror("listen");
          exit(1);
     }

     // addjuk hozzá a hallgatózót(listener) a master halmazhoz
     FD_SET(listener, &master);

     // tartsuk szem elõtt a legnagyobb fájlleírót
     fdmax = listener; // ez idáig ez az egy

     // main loop
     for(;;) {
          read_fds = master; // másold ezt
          if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
               perror("select");
               exit(1);
          }
```

```
            // fussunk keresztül a meglévõ csatlakozásainkon megnézni, hogy van-e
            olvasható adat
            for(i = 0; i <= fdmax; i++) {
                 if (FD_ISSET(i, &read_fds)) { // van egy!!
                      if (i == listener) {
                           // kezeljük az új csatlakozást
                           addrlen = sizeof(remoteaddr);
                           if ((newfd = accept(listener, (struct sockaddr
                           *)&remoteaddr,&addrlen)) == -1) {
                                perror("accept");
                           } else {
                                FD_SET(newfd, &master); // adjuk hozzá a master
                                halmazhoz
                                if (newfd > fdmax) { // ne veszítsük szem elõl a
                                maximumot
                                     fdmax = newfd;
                                }
                                printf("selectserver: new connection from %s on "
                                "socket %d\n", inet_ntoa(remoteaddr.sin_addr), newfd);
                           }
                      } else {
                           // kezeljük a kliens adatait
                           if ((nbytes = recv(i, buf, sizeof(buf), 0)) <= 0) {
                                // hibát kaptunk, vagy a kliens bezárta a kapcsolatot
                                if (nbytes == 0) {
                                     // kapcsolat bezárva
                                     printf("selectserver: socket %d hung up\n", i);
                                } else {
                                     perror("recv");
                                }
                                close(i); // viszlát!
                                FD_CLR(i, &master); // távolítsuk el a master
                                halmazból
                           } else {
                                // van egy kis adatunk a klienstõl
                                for(j = 0; j <= fdmax; j++) {
                                     // küldd el mindenkinek!
                                     if (FD_ISSET(j, &master)) {
                                          // kivétel a hallgatózónak és saját magának
                                          if (j != listener && j != i) {
                                               if (send(j, buf, nbytes, 0) == -1) {
                                                    perror("send");
                                               }
                                          }
                                     }
                                }
                           }
                      } // ez nagyon UNDORITO!
                 }
            }
       }

       return 0;
  }
```

Megjegyezzük, hogy két fájlleíróm van a kódban: *master* és *read_fds*. Az elsõ, a *master* tartalmazza a már kapcsolódott socketek leíróit, természetesen annak a socketnek is, amely az újabb kapcsolódásokhoz hallgat kifele.

Az ok, amiért van *master* halmazom, az az, hogy a `select()` idõszerûen *megváltoztatja* a halmazt azzal amit beletettél, hogy mutassa, hogy melyik socket áll készen olvasásra. Amellett nekem nyomon kell követnem a kapcsolatokat az egyik *select()* hívástól a másikig, és ezeket biztonságosan el kell tárolnom valahol. A legvégén bemásolom a *master*-t a *read_fds*be és azután meghívom a `select()` függvényt.

De ez nem azt jelenti, hogy mindig amikor egy új kapcsolatom lesz, akkor azt a *master* halmazhoz kell hozzáadnom? Naná! És minden alkalommal, amikor egy kapcsolat megbomlik, el kell azt távolítani a *master* halmazból? De igen, azt.

Vedd észre, hogy vizsgálódtam, hogy mikor áll készen a *listener* socket írásra. Amikor készen áll, az azt jelenti, hogy van egy új kapcsolati kérelmem, amelyet elfogadok (accept()) és hozzáadok a *master*

halmazhoz. Hasonlóan, amikor egy kliens kapcsolat készen áll az olvasásra és a `recv() 0`-val tér vissza tudom, hogy a kliens megbontotta a kapcsolatot és ezt el kell távolítani a *`master`* halmazból.

Ha a kliens `recv()` függvény nem nullával tér vissza, tudom, hogy adatok érkeztek. Így átveszem ezeket és keresztül futok a *`master`* listán és elküldöm az adatot a többi csatlakozásban levõ kliensnek.

És ez, kedves barátom, csak egy kevesebb-mint-egyszerû áttekintése volt a mindenható `select()` függvénynek.

6.3. Parciális `send()`-ek kezelése

Emlékezz vissza arra a fejezet részre, amikor a `send()` függvényrõl írtam (feljebb), amikor azt mondtam, hogy a `send()` lehet, hogy nem küldi el az összes bájt adatot, amit kérsz tõle?! Például lehet, hogy 512 bytot akarsz elküldeni, de õ csak 412-vel tér vissza. Mi lett a maradék 100 bájttal?

Nos, azok még mindig a kicsi bufferedben várakoznak, hogy ki legyenek küldve. A körülményektõl függõen az akaratodon kívül a kernel úgy döntött, hogy nem küldi ki az adatokat egyszerre, és most, kedves barátom a te feladatod, hogy kiküldd a maradék adatot.

Erre a célra írhatsz egy olyan függvényt, mint ez itt:

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

int sendall(int s, char *buf, int *len)
{
      int total = 0; // mennyi bájtot küldtünk már el
      int bytesleft = *len; // mennyi maradt még elküldetlenül
      int n;

      while(total < *len)
            n = send(s, buf+total, bytesleft, 0);
            if (n == -1) { break; }
            total += n;
            bytesleft -= n;
      }

      *len = total; // vissza adja azt a számot, amit itt aktuálisan küldtünk el

      return n==-1?-1:0; // hiba esetén -1, siker esetén pedig 0 a visszatérõ érték
}
```

Ebben a példában az *`s`* a socket amivel az adatokat ki akarod küldetni, a *`buf`* az a buffer, ami az adataidat tartalmazza, a *`len`* pedig egy `int` típusra mutató pointer, ahol az `int` a bufferben lévõ bájtok számát tartalmazza.

A függvény -1 értékkel tér vissza hiba esetén (az *`errno`* értékét még mindig a `send()` hívás állítja be.) Az aktuálisan elküldött bájtok számát a *`len`* adja vissza. Ennek értéke majd annyi lesz, mint azoknak a bájtoknak a száma, amiket kiküldesz, ellenkezõ esetben hiba lépett fel. A `sendall()` függvény a legjobb esetben "huffing and puffing", hogy kiküldje az adatokat, de ha hiba történik, akkor természetesen visszaadja azt neked.

A teljesség kedvéért itt van a függvény egy minta hívása:

```
char buf[10] = "Beej!";
int len;

len = strlen(buf);
if (sendall(s, buf, &len) == -1) {
      perror("sendall");
      printf("We only sent %d bytes because of the error!\n", len);
```

```
}
```

Mi történik a fogadó oldalon, amikor egy csomag része érkezik meg? Ha a csomagok változó hosszúságúak, honnan tudja a fogadó, hogy hol van egy csomag vége, és hol kezdõdik egy másik? Hát igen, ez egy elég kérdéses probléma.
Neked valószínû be kell *tokozódtatnod* (emlékszel rá az adat beágyazásról szóló fejezet részbõl még az elején?) További részletekért olvass hozzá valahol!

6.4. Az adatbeágyazás leszármazottja

Mit is jelent valójában beágyazni adatot? A legegyszerûbb esetben, azt jelenti, hogy hozzácsapsz egy headert amely valamennyi azonosító információt vagy a csomag hosszát tartalmazza, vagy mindkettõt.

Hogyan kell kinéznie a headernek? Nos, ez csak valamennyi bináris adat, ami szükséges a project sikeres végrehajtásához.

Wow. Ez elég homályos.

O.K. Bevezetés képpen mondjuk, hogy neked van egy multi-felhasználós chat programod, ami `SOCK_STREAM`eket használ. Amikor egy felhasználó beír ("mond") valamit, két információ részt kell elküldeni a szervernek: mit mondott és ki mondta.

Így távlatokban érthetõ? "Akkor mi a probléma?" kérdezheted.

A probléma az, hogy az üzenetek különbözõ hosszúságuak lehetnek. Egy felhasználó, aki mondjuk legyen "tom", azt mondhatja, hogy "Hi", egy másik felhaszánló, aki legyen "Benjamin", pedig azt mondhatja, hogy "Hey guys what is up?"

Ezek után te kiküldöd (`send()`) a cuccost a klienseknek, úgy, ahogy beérkezett. A kimenõ adat adatfolyamod így fog kinézni:

```
t o m H i B e n j a m i n H e y g u y s w h a t i s u p ?
```

És így tovább. Honnan tudja a kliens, hogy egy üzenet hol kezdõdik és hol végzõdik? Megteheted, ha akarod, hogy az összes üzenetet azonos hosszúságúra csinálod, és ekkor használhatod a fentebb már bemutatott `sendall()` hívást. De ez sávszélességet pazarol! Mi nem szeretnénk 1024 bájtot kiküldeni (`send()`) csak a "tom" azt mondja hogy "Hi"-hoz.

Így mi *beskatulyázzuk* az adatot egy kicsi header és csomag szerkezetbe. Mind a kliens és a szerver is tudja, hogy hogyan csomagolja be és ki ezt az adatot. Még nem látod, de egy *protokol*t kezdünk definiálni ami a szerver és a kliens kommunikációját írja le!

Ebben az esetben tegyük fel, hogy a felhasználó név 8 karakternyi kötött hosszúságú, kiegészítve '\0'-kal, ha szükséges. Aztán tegyük fel, hogy az adat változó hosszúságú, de maximum 128 karakter. Lássuk ezt a csomag szerkezet mintát, amit mi ebben a helyzetben használhatunk:

1. `len` (1 bájt, elõjelnélküli(unsigned)) - A csomag teljes hossza, beleszámolva a 8 bájtos felhasználó nevet és a chat adatot.
2. `name` (8 bájt) - A felhasználó neve, NUL-kiegészítéssel, ha szükséges.
3. `chatdata` (n-bájtos) - Maga az adat, 128 bájtnál nem több. A csomag hosszát ennek az adatnak a hossza + 8(a név hossza, fent) -ra kell kalkulálni.

Miért választottam a 8 bájt + 128 bájt határokat a mezõknek? A hasamra csapva adtam õket, feltéve hogy elég hosszúak lesznek. Ha úgy gondolod, hogy 8 bájt túl szûk a te szükségleteidhez, használhatsz 30-bájtos név mezõt is, vagy akármekkorát. A választás a tiéd.

A fenti csomagdefiníciót használva az elsõ csomag a következõ információkból állhat (hex és ASCIIben):

```
   0A      74 6f 6d 00 00 00 00 00    48 69

(hossz)    T o m + (kiegészítés)      H i
```

A második hasonlóan:

```
  14     42 65 6E 6A 61 6D 69 6E   48 65 79 20 67 75 79 73 20 77

(hossz)      B e n j a m i n          H e y g u y s w . . .
```

(A hossz Network Byte Orderben van tárolva természetesen. Ebben az esetben nem számít, mert csak egy bájtnyi, de általánosan mondjuk az összes bináris egész számot Network Byte Orderben szertnéd tárolni a csomagjaidban.)

Amikor ezt az adatot elküldöd, biztonságosnak kell lenned és a `sendall()` parancsot kell használnod (fent), így tudod, hogy az összes adat el lesz küldve, még akkor is, ha ez a `send()` többszörös hívását igényli, hogy kiküldd õket.

Hasonló képpen, amikor megkapod ezt az adatot, egy kis extra munkát kell végezned. Hogy biztosra menj, fel kell tenned, hogy lehet, hogy csak egy csomag részt kaptál (mint ahogy lehet, hogy mi az "`00 14 42 65 6E`"-t kapjuk Benjamintól(fent), ez az össz, amit a `recv()` hívásával kaptunk.) Amíg meg nem kapjuk az egész csomagot, újra és újra kell a `recv()` függvényt meghívni.

De hogyan? Nos, mi tudjuk a bájtok teljes számát, amiknek be kellene érkezni, amíg a csomag teljesen át nem jön, mivel ez a szám a csomag elejére lett biggyesztve. Ezenkívül tudjuk a maximális csomagméretet, ami 1+8+128, vagy 137 bájt (mivel így definiáltuk a csomagot.)

Amit tenni tudsz, hogy definiálsz egy két csomag részére elég nagy tömb típust. Ez lesz a munka tömböd, ahol újra építed a csomagokat, ahogy beérkeznek.

Mindig, amikor adatokak fogadsz(`recv()`), meg fogod etetni a munka bufferrel, és közben megvizsgálod, hogy a csomag elkészült-e. Ez akkor van, ha bájtok száma a bufferben nagyobb vagy egyenlõ azzal a hosszal, amit a header elején specifikáltál (+1, mert a hossz nem tartalmazza azt a bájtot ami a hosszt tartalmazza, azaz sajátmagát.) Ha a bájtok száma a bufferben kisebb, mint 1, a csomag a bufferban nyílvánvalóan nincs készen. Ezt egy speciális esetként kell kezelni, mivel az elsõ bájt szemét és nem számíthatsz rá a csomag korrekt hosszával kapcsolatban.

Amikor a csomag elkészült, azt csinálhatsz vele, amit akarsz. Használhatod, vagy éppen eltávolíthatod a munka bufferedbõl.

Húha! Sikerült belevarázsolni a fejedbe ezt? Nos, akkor itt van a második felvonása az én kis elõadásomnak: olvashattad az egyik csomag végét a múltban és a `recv()`vel hívsz a következõ felé. Ez az, hogy már van egy elkészült csomagod és egy hiányos csomag rész a munka bufferben! Véres játszma. (De pont ezért lett a munka buffer olyan nagyra csinálva, így el tud tárolni két csomagot is - mint ahogy ebben az esetben is történt!)

Mivel a headerbõl tudod az elsõ csomag hosszát, és lefoglalsz területet a bájtok száma szerint a munka bufferben, így le tudod vonni és ki tudod számolni, hogy a munka bufferben mennyi bájt tartozik a második (hiányos) csomaghoz. Amikor az elsõt lekezelted, akkor azt kitörölheted a munka bufferbõl és belrakhatod a hiányos második csomagot a buffer elejébe, így már készen is állsz a következõ `recv()` híváshoz.

(Kedves olvasóim, közületek páran megjegyezhetik, hogy a hiányos csomagrész mozgatása a buffer elejébe idõt igényel, és a programot meg lehet úgy írni, hogy erre ne legyen szükség, még pedig körfolytonos buffer segítségével. Sajnálatotokra a körfolytonos buffer megtárgyalása ennek a leírásnak a keretein kívül mozog. Ha még mindig kíváncsi vagy, ragadj meg egy adat struktúrákról szóló könyvet és ott utánna olvashatsz.)

Sosem mondtam, hogy ez egyszerû volt. O.K., mondtam, hogy egyszerû. És az is; csak egy kis gyakorlatot kell szerezned, és akkor nemsokára számodra is természetes lesz. Ezt megígérem!

7. További referencia

Eléggé belemélyültél és még többet szeretnél tudni!? Hol a frászban tanulhatsz még többet errõl az anyagról?

7.1. *man* oldalak

Kezdetnek próbáld meg a következõ man oldalakat:

- `htonl()`[18]
- `htons()`[19]
- `ntohl()`[20]
- `ntohs()`[21]
- `inet_aton()`[22]
- `inet_addr()`[23]
- `inet_ntoa()`[24]
- `socket()`[25]
- `socket options`[26]
- `bind()`[27]
- `connect()`[28]
- `listen()`[29]
- `accept()`[30]
- `send()`[31]
- `recv()`[32]
- `sendto()`[33]
- `recvfrom()`[34]
- `close()`[35]
- `shutdown()`[36]

- `getpeername()`[37]
- `getsockname()`[38]
- `gethostbyname()`[39]
- `gethostbyaddr()`[40]
- `getprotobyname()`[41]
- `fcntl()`[42]
- `select()`[43]
- `perror()`[44]
- `gettimeofday()`[45]

7.2. Könyvek

A régi iskolák aktualitásáért tartsd-a-kezedben típusú rostpapírú könyvek, próbálj ki párat a következõ kíválló útmutatók közül. Vedd észre a szembetûnõ Amazon.com logót. Amit ez a szemtelen reklám jelent, az az hogy én alapjába véve kapok egy seggberúgást (Amazon.com store credit), amiért ezeket a könyveket ezen az útmutatón keresztül árulom. Így ha meg fogod rendelni valamelyik könyvet ezek közül, miért nem küldesz egy köszit, hogy a murid egy alábbi linkkel kezdõdhet.

Emellett, több könyvvel még több segítséghez vezetlek titeket. ;-)

Unix Network Programming, volumes 1-2 by W. Richard Stevens. Published by Prentice Hall. ISBNs for volumes 1-2: 013490012X[47], 0130810819[48].

Internetworking with TCP/IP, volumes I-III by Douglas E. Comer and David L. Stevens. Published by Prentice Hall. ISBNs for volumes I, II, and III: 0130183806[49], 0139738436[50], 0138487146[51].

TCP/IP Illustrated, volumes 1-3 by W. Richard Stevens and Gary R. Wright. Published by AddisonWesley. ISBNs for volumes 1, 2, and 3: 0201633469[52], 020163354X[53], 0201634953[54].

TCP/IP Network Administration by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 1565923227[55].

Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by AddisonWesley. ISBN 0201563177[56].
Using C on the UNIX System by David A. Curry. Published by O'Reilly & Associates, Inc. ISBN 0937175234. Out of print.

7.3. Webes referenciák

A Weben:

BSD Socketek: Egy gyors és undorító alapozó[57] (más Unix rendszerekhez való programozási infó is van!)

A Unix Socket FAQ[58]

Kliens-Szerver programozás[59]

Bevezetés a TCP/IP-be[60] (gopher)

Internet Protocol - leggyakrabban feltett kérdések (FAQ)[61]

A Winsock FAQ[62]

7.4. RFC-k

RFCs[63] – az igazi ócsmányság:

RFC-768[64] – A felhasználói Datagram Protocol (UDP)

RFC-791[65] – Az Internet Protocol (IP)

RFC-793[66] – A Transmission Control Protocol (TCP)

RFC-854[67] – A Telnet Protocol

RFC-951[68] – A Bootstrap Protocol (BOOTP)

RFC-1350[69] – A Trivial File Transfer Protocol (TFTP)

8. Általános kérdések

K: Honnan tudom megszerezni azokat a header fájlokat?

V: Ha már nincsenek meg a rendszeredben, akkor lehet, hogy nincs is rájuk szükség. Nézz utánna a platformod használati útmutatójában. Ha windows alatt dolgozol, akkor csak a `#include <winsock.h>`-ra van szükséged.

K: Mit csináljak akkor, amikor a `bind()` azt mondja, hogy "A cím már használt"?

V: Akkor a `setsockopt()`ot kell használnod az `SO_REUSEADDR` opcióval a hallgatózó socketen. Példákért lásd a `bind()` és a `select()` részeket feljebb.

K: Hogyan tudom kilistáztatni a rendszeremen lévõ nyitott socketetket?

V: Használd a netstat programot. Részletesebb leírásért nézz utánna a man oldalán, de még a következõ beírásával is pár jó infót szerzhetsz:

```
$ netstat
```

Az egyedüli trükk meghatározni, hogy melyik socketet melyik program használja. :-)

K: Hogyan tudom megnézni a routing táblát?

V: Futtasd a route parancsot (a legtöbb Linuxban a `/sbin` könyvtárban található) vagy a netstat -r parancsot.

K: Hogyan tudom egyszerre futtatni a kliens és a szerver programot, ha csak egy számítógépem van? Nincs hálózatra szükségem hálózati programok írásához?

V: Szerencsédre jóformán minden gépbe van egy úgynevezett loopback hálózati "device" implementálva, amely a kernelben van és hálózati kártyának tetteti magát. (Ez az "`lo`"-ként listázott interfész a routing táblában.)

Szimuláljuk azt, hogy a "`goat`" névvel vagy bejelentkezve egy géphez. Futtasd a klienst egy ablakban a szervert egy másikban, vagy indítsd el a szervert a háttérben ("**server &**") és indítsd el a kliens ugyanabban az ablakban. A loopback device eredménye hogy lehetsz egyszerre **kliens goat (client goat)** vagy **kliens localhost (client localhost)** (azáltal, hogy a "`localhost`" valószínûleg már definiált a `/etc/hosts` fájlodban) és így van egy kliensed, ami a szerverrel kommunikál hálózat nélkül!

Röviden, semmi módosítást nem igényelnek a programokódok, hogy egy hálózat nélküli gépen futtasd õket! Hurrá!

K: Hogyan ismerhetem fel hogy a távoli oldal bezárta a kommunikációt?

V: Onnan, hogy a `recv()` `0`-t ad vissza.

K: Hogyna tudok implementálni egy "ping" utilityt? Mi az az ICMP? Hol találok többet a raw socketekrõl és a `SOCK_RAW`ról?

V: Minden raw sockettel kapcsolatos kérdésedre választ ad W. Richard Stevens UNIX Network Programming címû könyve. Lásd ennek az útmutatónak a könyvekkel foglalkozó részét.

K: Hogyan tudok Windows alatt fordítani?

V: Elõször töröld le a Windowst és installáld a Linuxot vagy BSD-t. };-). Nem szükséges, csak nézz utánna a bevezetõben a Windowsról szóló fejezet részben.

K: Hogyan tudok Solaris/SunOS alá fordítani? Egyfolytában linker hibákat kapok, amikor megpróbálok fordítani!

V: A linker hibák azért történnek, mert a Sun rendszerek nem a socket könyvtárban fordítanak automatikusan. Példáért nézz a bevezetõ fejezet "Megjegyzés a Solaris/SunOs programozóknak" címû részében.

K: A `select()` miért lép ki a sorból signál kapása esetén?

V: A signálok blokkolt rendszer hívása esetén megeshet, hogy -1 értékkel térnek vissza, az *errno*-t pedig `EINTR` értékre állítják. Ha a `sigaction()` függvény segítségével beiktatsz egy signal kezelõt, akkor be tudod állítani a flaget `SA_RESTART`ra, amely feltételezhetõen újraindítja a rendszerhívást megszakítás

után.

Természetesen ez nem mindig mûködik.

A kedvenc megoldásom erre, hogy belerakok egy `goto` jegyzéket. Tudod, hogy ez vég nélkül irritálja az elõadóidat, így vessük rá magunkat!

```
select_restart:

     if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
          if (errno == EINTR) {
               // pár signal éppen megszakított minket, így újraindítjuk
               goto select_restart;
          }
          // az igazi hibát itt kezeljük:
          perror("select");
     }
```

Igaz, nem *szükséges* a `goto`-t használni ebben az esetben; más struktúrákat is tudsz használni ennek kezelésére, de én úgy gondoltam, hogy a `goto` sokkal tisztább megoldás.

K: Hogyan tudok idõzítõt implementálni a `recv()` hívásomhoz?

V: Használd a `select()` függvényt! Ez megengedi, hogy egy idõzítõ paramétert specifikálj annak a socket leíród részére, amelyrõl olvasni szeretnél. Vagy, elrejtheted a tényleges funkcionalitását egy olyan függvényben, mint ez:

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

int recvtimeout(int s, char *buf, int len, int timeout)
{

     fd_set fds;
     int n;
     struct timeval tv;

     // állítsuk be a fájlleíró beállításait
     FD_ZERO(&fds);
     FD_SET(s, &fds);

     // állítsuk be a struktúra lejárati idõértékét
     tv.tv_sec = timeout;
     tv.tv_usec = 0;

     // várjon amíg adat nem érkezik, vagy le nem telik az idõ
     n = select(s+1, &fds, NULL, NULL, &tv);
     if (n == 0) return -2; // le telt az idõ!
     if (n == -1) return -1; // hiba

     // az adat itt van, így hajtson végre egy normális recv() hívást
     return recv(s, buf, len, 0);

}

// A recvtimeout() függvény egy minta hívása:

     .
     .
     n = recvtimeout(s, buf, sizeof(buf), 10); // 10 másodperc lejárati idõ

     if (n == -1) {
          // hiba következett be
          perror("recvtimeout");
     }
     else if (n == -2) {
          // lejárt az idõ
     } else {
```

```
        // adatot kaptunk a bufferbe
    }
    .
    .
```

Vedd észre, hogy a `recvtimeout() -2` értéket ad idõlejárás esetében. Miért nem `0`-át? Nos, ha újrahívod, a `recv()` hívása esetén a `0` visszatérõ érték azt jelenti, hogy a távoli oldal bezárta a kapcsolatot. Így ennek a visszatérési értéknek már van jelentése, míg a `-1` ugye a hibát jelenti, így a `-2` értéket választottam a lejárt idõ jelzésére.

K: Hogyan tudom titkosítani(encrypt) vagy betömöríteni az adatokat, mielõtt elküldöm a socketen keresztül?

A: Az titkosítás egy egyszerû módja, hogy SSL-t (secure sockets layer) használsz, de ez ennek a leírásnak a határain kívül mozog.

De tegyük fel, hogy saját tömörítõ vagy titkosító rendszert szeretnél implementálni, meggondolandó, hogy ehhez az adaton lépések sorát kell futtatni a kapcsolat mindkét végén. Minden lépés megváltoztatja az adatot valamilyen módon.

1. szerver adatokat olvas egy fájlból (vagy akárhonnan)
2. a szerver titkosítja az adatot (ezt a részt te adod hozzá)
3. a szerver elküldi a titkosított adatot

Most a másik irány:

1. a kliens megkapja(`recv()`) az titkosított adatot
2. a kliens visszakódolja az adatot (ezt a részt te írod meg)
3. a kliens kiírja az adatokat egy fájlba (vagy máshova)

A tömörítés is hasonlóan megy mint a titkosítás. Vagy megcsinálhatod mind a kettõt! Csak el ne felejts a titkosítás elõtt tömöríteni. :)

Éppen ameddig a kliens visszacsinálja, azt amit a szerver csinált, az adat a legvégén visszaalakul nem törõdve azzal, hogy mennyi lépésen keresztül babrálták.

Az össz dolog, amit tenned kell, hogy használod a kódomat, megtalálva neki a helyet az adat olvasása és a hálózatra való küldése(`send()`) között, ragassz oda egy kis titkosító kódot aztán kész is.

K: Mi az a "`PF_INET`" amit mindenhol látok? Rokon az `AF_INET`-tel?

V: Igen, az. Lásd a `socket()` fejezet részt részletekért.

K: Hogyan tudok olyan szervert írni, amelyik shell parancsokat fogad el a klienstõl és végre is hajtja

õket?

V: Az egyszerûség kedvéért mondjuk, hogy a kliens kapcsolódik (`connect()`), adatot küld (`send()`) és bezárja (`close()`) a kapcsolatot (ez van,nincsen rákövetkezõ rendszerhívás a kliens újabb kapcsolódása nélkül.)

A folyamat, amit a kliens követ, a következõ:

1. kapcsolódás (`connect()`) egy szerverhez
2. `send("/sbin/ls > /tmp/client.out")`
3. bezárja `close()` a kapcsolatot

Eközben a szerver kezeli az adatokat majd végrehajtja azokat:

1. elfogadja `accept()` a kliens irányába a kapcsolatot
2. fogadja `recv(str)` a parancs szöveget
3. bezárja `close()` a kapcsolatot
4. meghívja a `system(str)` hívást, hogy futtassa a kapott parancsot

Vigyázz! Hagyni, hogy a server végrehajtsa a kliens által adott parancsokat, olyan mintha távoli shell hozzáférést adnánk és az emberek a te azonosítóddal tudnának dolgokat csinálni, amikor kapcsolódnak a szerverhez. Bevezetés képpen, a fenti példában, mi van akkor, ha a kliens küld egy "**rm -rf ~**" parancsot? Mindent töröl az azonosítódban!

Kezdesz bölcs lenni, és megakadályozod a klienst, hogy egy pár elõre megadott dologon kívül, amiket tudod, hogy rád nézve biztonságosan használhatóak, mást ne érhessen el. Példa legyen a **foobar** utility:

```
if (!strcmp(str, "foobar")) {
     sprintf(sysstr, "%s > /tmp/server.out", str);
     system(sysstr);
}
```

De még mindig nem vagy biztonságban, sajnos: mi van akkor, ha a kliens a következõt üti be: "**foobars;rm -rf~**"? A legbiztonságosabb dolog, hogy írsz egy kis rutint amely escape("\") karaktert tesz az összes nem alfanumerikus karakter elé (beleérve a space helyeket, ha helyénvaló) a parancs argumentumaiban.

Ahogy láthatod, a biztonság felügyelése egy elég nagy terjedelmû kiadvány, amikor a server elkezdi végrehajtani azokat a dolgokat, amiket a kliens küldött.

K: Elküldtem egy adag adatot, de amikor `recv()` lekértem õket, csak 536 bájtot, vagy 1460 bájtot küldött egy idõben. De ha a helyi gépemen futtattam, akkor megkaptam az összes adatot ugyanabban az idõben. Mi történik?

V: Elérted az MTU-t - a maximum méret, amit a fizikai egység kezelni képes. A helyi gépen a loopback

eszközt használtad amelyiknek 8K vagy még több kezelése nem ügy. De etherneten, ami csak 1500 bájtot tud egyszerre kezelni a headerrel együtt, elérted a határt. Modemen keresztül 576 MTU-val (itt is a headerrel együtt), is elérted az egész alacsony határt.

Biztos akarsz lenni, hogy az összes adat el lett küldve, egyszer és mindekorra. (lásd a `sendall()` függvény implementációját részletekért.) Amikor biztos vagy már benne, a `recv()` függvényt kell használni ciklusban, amíg az összes adat be nem lesz olvasva.

Olvasdd el "Az adatbeágyazódás leszármazottja" címû fejezet részt részletekért az olyan csomagok fogadásával kapcsolatban, amik többszörös `recv()` hívást használnak.

K: Windows alatt dolgozom és nincs `fork()` rendszer hívásom vagy bármilyen fajta `struct sigaction` struktúrám. Mi tegyek?

V: Ha valahol is vannak, akkor a POSIX könyvtárakban lesznek, amiket a fordítóddal kezelhetsz. Mivel nekem nincs Windowsom, tényleg nem tudok ennél többet mondani, de nekem úgy van az emlékeimben, hogy a Microsoftnak van egy POSIX kompatibilis rétege és ez az, ahol a `fork()` lehet. (És talán a `sigaction` is.)

Az MSDNben keress rá a "fork" vagy a "POSIX" kulcsszóval és talán kiad valamit.

Ha ez az ötlet nem mûködik, akkor távolítsd el a `fork()/sigaction` cuccot és helyettesítsd a Win32-s megfelelõjével: `CreateProcess()`. Nem tudom, hogyan kell használni ezt - buzillion sok argumentuma van, de az MSDNben biztosan le van írva.

K: Hogyan tudok adatot küldeni TCP/IP felügyelete mellett, titkosítás használatával?

V: Figyeld meg az OpenSSL projectben[70].

K: Én egy tûzfal mögött vagyok - hogyan tudom megadni a tûzfalon kívüli embereknek az IP címem, hogy a gépemre csatlakozhassanak?

V: Sajnos, a tûzfal célja pont az, hogy megvédje a gépet a tûzfalon kívüli emberek rákapcsolódásától, így engedélyezni nekik ezt, alapjában véve a biztonságosság megsértésének tekinthetõ.

Ez nem azt jelenti, hogy minden el van veszve. Még mindig tudsz csatlakozni (`connect()`) a tûzfalon keresztül, ha az csinál valamilyen fajta álarcosbált(? - :-)), vagy NAT-t, vagy valami ehhez hasonlót. Csak szerkezd úgy a programod, hogy te legyél mindig az egyik alakítója a kapcsolatnak, és akkor rendben leszel.

Ha ez nem kielégítõ, akkor kérd meg a rendszergazdát, hogy ejtsen egy rést a tûzfalban, és úgy már tudnak emberek hozzád kapcsolódni. Tûzfal feléd is lehet egy NAT szoftveren, vagy proxyn vagy valami hasonlón keresztül.

Figyelj arra, hogy egy lyuk a tûzfalban nem vehetõ félválra. Biztosnak kell lenned, hogy nem adsz gonosz embereknek hozzáférést egy bizalmas hálózatba; ha még kezdõ vagy, akkor elég nehéz olyan szoftver felügyelõt készítenei, mint amilyet elképzelhetsz.

Ja, és ha lehet ne mérgesítsd rám a rendszergazdádat. ;-)

9. Helyreigazítás és segítségkérés

Nos, ez a leggyakoribb. Végezetül remélem, hogy az információ nagy része, ami ebben a dokumentumban

van, hiteles és õszintén remélem nincs benne semilyen feltûnõ hiba. Nos, biztos, hogy van.

Így, legyen ez egy figyelmeztetés számodra! Sajnálom, ha bármilyen a dokumentumban lévõ hiba fájdalmat okoz neked, de nem vonhatsz felelõsségre. Hogy értsd, a dokumentum egyetlen szava mögött sem én állok. Az egész cucc akár teljesen rossz is lehet!

De talán mégsem az! Ezen felül, sok-sok órát töltöttem ennek a cuccnak a veszõdésével (én is -> a fordító :-)), és számos TCP/IP hálózati segédszeközt telepítettem munka közben, írtam többjátékos játék motorokat, és így tovább. De én nem vagyok socket Isten; én is csak egy amolyan faszi vagyok, mint a többi.

Minden esetre, ha bárkinek akármilyen építõ(romboló)jellegû kritikája van ezzel a dokumentummal kapcsolatban, kérem írjon egy levelet a <beej@privatehaven.org> címre és meg fogok próbálni erõt kifejteni hogy rekord tisztágúra csináljam.

Abban az esetben, ha azt kérdeznéd miért írtam ezt, nos, a pénzért tettem. Ha! Nem, valójában, azért, mert egy csomó ember fordult hozzám a sockettel kapcsolatos kérdésekkel és én megkérdeztem õket, mit szólnának, ha kiraknám együtt õket egy socket oldalra, erre õk azt mondták: "Király!". Emellett úgy éreztem, hogy ez a nehezen megszerzett tudomány elpocsékolódna, ha nem tudnám másokkal is megosztani. A web pedig erre a legkiválóbb módnak bizonyul. Másokat is bíztatok arra, hogy lássanak el hasonló információkkal, amikor lehetséges.

Na jó, elég ebbõl - gyerünk vissza kódolni! ;-)

Megjegyzések

1. http://www.ecst.csuchico.edu/~beej/guide/net/
2. http://tangentsoft.net/wskfaq/
3. http://www.tuxedo.org/~esr/faqs/smart-questions.html
4. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/send.2.inc
5. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recv.2.inc
6. http://www.rfc-editor.org/rfc/rfc793.txt
7. http://www.rfc-editor.org/rfc/rfc791.txt
8. http://www.rfc-editor.org/rfc/rfc768.txt
9. http://www.rfc-editor.org/rfc/rfc791.txt
10. http://www.rfc-editor.org/rfc/rfc1413.txt
11. http://www.ecst.csuchico.edu/~beej/guide/net/examples/getip.c
12. http://www.ecst.csuchico.edu/~beej/guide/net/examples/server.c
13. http://www.ecst.csuchico.edu/~beej/guide/net/examples/client.c
14. http://www.ecst.csuchico.edu/~beej/guide/net/examples/listener.c
15. http://www.ecst.csuchico.edu/~beej/guide/net/examples/talker.c
16. http://www.ecst.csuchico.edu/~beej/guide/net/examples/select.c
17. http://www.ecst.csuchico.edu/~beej/guide/net/examples/selectserver.c
18. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/htonl.3.inc
19. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/htons.3.inc
20. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/ntohl.3.inc
21. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/ntohs.3.inc

61. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_aton.3.inc
62. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_addr.3.inc
63. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_ntoa.3.inc
64. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/socket.2.inc
65. http://linux.com.hk/man/showman.cgi?manpath=/man/man7/socket.7.inc
66. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/bind.2.inc
67. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/connect.2.inc
68. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/listen.2.inc
69. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/accept.2.inc
70. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/send.2.inc
71. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recv.2.inc
72. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/sendto.2.inc
73. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recvfrom.2.inc
74. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/close.2.inc
75. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/shutdown.2.inc
76. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getpeername.2.inc
77. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getsockname.2.inc
78. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyname.3.inc
79. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyaddr.3.inc
80. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/getprotobyname.3.inc
81. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/fcntl.2.inc
82. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/select.2.inc
83. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/perror.3.inc
84. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/gettimeofday.2.inc
85. http://www.amazon.com/exec/obidos/redirect-home/beejsguides-20
86. http://www.amazon.com/exec/obidos/ASIN/013490012X/beejsguides-20
87. http://www.amazon.com/exec/obidos/ASIN/0130810819/beejsguides-20
88. http://www.amazon.com/exec/obidos/ASIN/0130183806/beejsguides-20
89. http://www.amazon.com/exec/obidos/ASIN/0139738436/beejsguides-20
90. http://www.amazon.com/exec/obidos/ASIN/0138487146/beejsguides-20
91. http://www.amazon.com/exec/obidos/ASIN/0201633469/beejsguides-20
92. http://www.amazon.com/exec/obidos/ASIN/020163354X/beejsguides-20
93. http://www.amazon.com/exec/obidos/ASIN/0201634953/beejsguides-20
94. http://www.amazon.com/exec/obidos/ASIN/1565923227/beejsguides-20
95. http://www.amazon.com/exec/obidos/ASIN/0201563177/beejsguides-20
96. http://www.cs.umn.edu/~bentlema/unix/
97. http://www.ibrado.com/sock-faq/
98. http://pandonia.canberra.edu.au/ClientServer/
99. gopher://gopher-chem.ucdavis.edu/11/Index/Internet_aw/Intro_the_Internet/intro.to.ip/

61. http://www-iso8859-5.stack.net/pages/faqs/tcpip/tcpipfaq.html
62. http://tangentsoft.net/wskfaq/
63. http://www.rfc-editor.org/
64. http://www.rfc-editor.org/rfc/rfc768.txt
65. http://www.rfc-editor.org/rfc/rfc791.txt
66. http://www.rfc-editor.org/rfc/rfc793.txt
67. http://www.rfc-editor.org/rfc/rfc854.txt
68. http://www.rfc-editor.org/rfc/rfc951.txt
69. http://www.rfc-editor.org/rfc/rfc1350.txt
70. http://www.openssl.org/