Beejs Guide Till Nätverksprogrammering
med Internetsockets

Brian ”Beej” Hall

beej@piratehaven.org

Copyright © 1995-2001 by Brian "Beej" Hall

Revisions Historia

Revision Version 1.0.0 augusti, 1995 Reviderad av: beej
Första versionen.
Revision Version 1.5.5 januari 13, 1999 Reviderad av: beej
Senaste HTML-versionen.
Revision Version 2.0.0 mars 6, 2001 Reviderad av: beej
Konverterad till DocBook XML, korrektioner, tillägg.
Revision Version 2.0.1 mars 7, 2001 Reviderad av: beej
Mindre korrektioner.
Revision Version 2.0.2 mars 16, 2001 Reviderad av: beej
inet_ntoa() skulle ha varit inet_aton() på en del stället.
Revision Version 2.0.3 april 2, 2001 Reviderad av: beej
inet_aton()s returnvärde har korregerats , selectserver ändrad, stavfel fixade, lagt till recvtimeout().
Revision Version 2.1.0 maj 3, 2001 Reviderad av: beej
Buffer overruns i client.c och listener.c fixade, gjort server.c stabila “reap zombies”, lagt till mailpolicy.
Revision Version 2.2.0 juni 24, 2001 Reviderad av: beej
Uppdaterad SoF, Windows, talker.c, PF_INET info, accept() demo, inga mer bzero()s, Solaris setsockopt().
Revision Version 2.2.1 juni 25, 2001 Reviderad av: beej
Mindre SoF uppdateringar, lagt till Amazon relaterade saker.

1. Introduktion

Hallå där! Har socket programmering tagit knäcken på dig? Är de här grejerna lite för svåra att lära sig utifrån man-sidorna? Du vill syssla med Internet programmering, men har inte tid att ta dig igenom högar av structs bara för att lista ut om du skall anropa call() före du anropar connect(), etc.

Men du, vet du vad? Jag har redan gjort det värsta, och jag bara *måste* få dela med mig av den informationen med resten av världen! Du har kommit till rätt ställe. Detta dokument borde ge den medel kunnige C programmeraren en bra början till det där nätverks tjafset.

1.1 Läsare

Det här dokumentet har skrivits som ett inlärningsmaterial, och inte som en referens. Det är troligtvis bäst läst av personer som precis har börjat med socket programmering och söker efter fotfäste. Det är absolut ingen *komplett* guide till socketprogrammering.

Förhoppningsvis kommer det att vara tillräckligt för förstå vad det står på de där man-sidorna.

1.2 Plattform och kompilator

Koden som finns i dokumentet var kompilerad på en Linux PC med GNUs **gcc** kompilator. Men det bör fungera på andra plattformar som använder **gcc**. Naturligtvis gäller inte detta om du programmerar på Windows – Se kapitlet för Windows programmering nedan.

1.3 Den officiella hemsidan

Den officiella hemsidan för det här dokumentet finns hos California State University, Chico, under:
http://www.ecst.csuchico.edu/~beej/guide/net/[1]

1.4 Notering för Solaris/SunOS programmerare

När koden kompileras för Solaris eller SunOS, så behövs det några extra parametrar för att länka de rätta biblioteken. Detta görs genom att man lägger till ”-lnsl –lsocket –lresolv” i slutet av kompileringskommandot, så här:

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

Om du fortfarande får fel så pröva med att lägga till ”-lxnet” i slutet av kommandoraden. Jag vet inte exakt vad det gör, men en del tycks behöva detta tillägg.

Ett annat problem som kan uppstå är vid anropet av setsockopt(). Prototypen skiljer sig från den på min Linux burk, så istället för

```
int yes=1;
```

så skriv:

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

Jag har ingen Sunburk, så jag har inte testat den ovanstående informationen – Det är bara vad folk har berättat för mig via email.

1.5 Noteringar för Windowsprogrammerare

Jag ogillar Windows till en viss del, och tycker att du skall prova Linux, BSD eller Unix istället. Det sägs dock att du kan använda detta under Windows.

För det första ignorera i princip allt vad som nämns angående system header-filer. Allt du behöver inkludera är:

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

Vänta lite. DU behöver också anropa WSAStartup() innan du gör något annat med socket biblioteket. Koden som behövs ser ut någon ting så här:

```
#include <winsock.h>
{
    WSADATA wsaData; // Om detta inte fungerar
    //WSAData wsaData; // Försök med detta.

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

Du behöver också säga till din kompilator att länka till Winsock-biblioteket, vanligtvis wsock21.lib eller winsock32.lib eller liknande. Under VC++ kan detta göras genom Project menyn, under Settings... Klicka på "Link" fliken, and titta efter en ruta som heter "Object/library modules". Lägg till "wsock.lib" till listan.

Det är i alla fall vad jag har hört.

När du har gjort det här, så borde exemplen i dokumentet fungera, med vissa undantag. Till exempel kan du inte använda close() för att stänga en socket – utan måste använda closesocket(), istället. Och select() fungerar bara med socketdeskreptorer, inte med fildeskriptorer (som 0 för stdin).

För att få mer information om Winsock, läs Winsock FAQen[2] och utgå från den.

Till sist har jag hört att Windows inte har något systemanrop motsvarande fork(), vilket, oturligt nog, används i några av mina exempel. Möjligtvis kan du länka till något POSIX bibliotek eller nått för att få det att fungera, eller så kan du använda CreateProcess() istället. fork() tar inga argument och CreateProcess() tar ungefär 48 miljarder argument. Välkommen till det underbara världen Win32 programmering.

1.6 Email policy

Jag finns oftast tillgänglig för att svara på frågor via email så det är helt ok att skriva, men jag kan inte garantera ett svar. Jag har ett ganska hektiskt liv och det finns perioder som jag inte

kans svara på dina frågor. När det är så brukar jag bara radera mailet. Det är inget personligt, utan jag har bara inte tid att ge alla de detaljer som ett svar kräver.

Som regel så är det mindre chans att få svar desto mer komplex frågan är. Om du smalnar av frågan före du mailar över den och är noga med att skicka med all information rörande problemet (som plattform, kompilator, felmeddelande du får, och allt annat som du tror kan hjälpa mig att lösa problemet), är det större chans att du får svar.

Om inte, så knacka lite mer, försök att hitta lösningen. Om det fortfarande blir fel, skriv till mig igen och skicka med den nya informationen du fått. Förhoppningsvis är det tillräckligt för mig för att hjälpa.

Nu har jag plågat dig med hur du skall och hur du inte skall skriva till mig. Men du skall veta att jag till fullo uppskattar all den hyllning som denna guide har fått över åren. Det ger kick, och det glädjer mig att den blir till nytta! =) Tack

1.7 Spegling

Du får mer än gärna spegla den här siten, oavsett om det är för privat eller allmänt bruk. Om du vill att din spegling skall länkas från min sida så skicka ett mail till mig på <beej@piratehaven.org>.

1.8 Notering för översättare

Om du vill översätta den här guiden till ett annat språk, så skriv till mig på <beej@piratehaven.org>, så länkar jag till översättningen från min huvudsida.

Det är tillåtet att lägga till ditt namn och email till det översatta dokumentet.

Tyvärr så kan jag inte lägga upp dokumentet själv på grund av platsbrist.

1.9 Upphovsrätt och spridning

Beej's Guide till Näteverksprogrammering är Kopieringsskyddad © 1995-2001 Brian "Beej" Hall.

Denna guide får fritt kopieras i alla typer av medium såvida att innehållet inte ändras, det är presenterat i dess helhet, och att detta avsnitt angående upphovsrätts avsnitt är intakt.

Lärare är uppmuntrade att rekommendera eller sprida kopior av denna guide till sina studenter.

Denna guide får översättas till vilket språk som helst, såvida översättningen är rätt, och guiden är översatt i sin helhet. Översättningen får också inkludera översättarens namn och kontaktinformation till denne.

Källkoden som finns med i dokumentet är härmed allmän egendom.

Kontakta <beej@piratehaven.org> för mer information.

1.10 Översättarens kommentarer

För det första måste jag påpeka att jag inte är jätteduktig på socketprogrammering utan denna översättning är en del i mitt lärande. Jag får tillfälle att tränga djupare in i texten och dess innebörd; jag blir tvungen att ta till mig det som står. Man kan tycka att det är onödigt arbete, men då Du läser detta dokument nu så kan inte allt arbete vara förgäves.

Du kan kontakta mig angående dokumentets översättning, men jag kommer inte att svara på några frågor rörande textens innehåll. Skriv gärna på <asgd@linux.nu>, och snälla, skriv ”socketprogrammering” i ämnesraden.

Nya översättningar av dokumentet kan du hitta på Beejs egna hemsida <http://www.ecst.csuchico.edu/~beej/guide/net/> eller på min sida <w1.520.telia.com/~u52013572/dokument>

Versionen på dokumentet är 2.2.1swe1

2. Var är en socket?

Du hör att folk talar om ”sockets” hela tiden, och kanske har du undrat vad exakt det är för något? Nå, Detta är vad en socket är: Ett sätt att prata med andra program genom att använda Unix standard Fildeskriptor (file descriptor /översättarens anmärkning).

Va?

Ok- Du kanske har hört någon Unixhacker säga ”Jösses, *allt* i Unix är en fil!”. Vad personen troligtvis har pratat om, är det faktum att när ett Unixprogram gör någon sort av I/O, så vad de gör är att läsa eller skriva till en fildeskriptor. En fildeskriptor är helt enkelt en integer som är kopplad till en öppen fil. Men (här kommer haken) den här filen kan vara en nätverks koppling, en FIFO, en pipe, en terminal, en riktig på-disk-fil, eller något helt annat. Allt i Unix *är* en fil. Så när du skall kommunicera med ett annat program över Internet, så kommer det ske genom fildeskriptorer, även om du tror det eller ej.

”Och var kan jag finna den där fildeskriptoren för nätverkskommunikation då, Herr Besserwisser?”, är troligtvis den sista frågan du har nu, men jag kommer svara på den i alla fall: Du kan anropa socket()-systemrutinen. Den returnerar socketdeskriptorn och Du kommunicerar sedan med den genom de specialiserade socketfunktionerna send() och recive() (**man send**[3]**, man recv**[4]).

”Men hörru”, kanske du tänker nu. ”Om det är en fildeskriptor, varför i Neptunus namn kan man inte använda de normala read() and write() anrop för att kommunicera med en socket?”. Det korta svaret är ”Det kan du!”. Det långa svaret är ”Du kan, men send() och recv() erbjuder mycket bättre kontroll över din dataöverföring.”

2.1 Två typer av Internet sockets

Vad nu? Finns det två typer av Internet sockets? Ja. ööh, nej nu ljuger jag. Det finns fler, men jag ville inte skrämma dig. Men jag kommer bara att prata om två typer här. Förutom i denna

mening, där jag kommer att tala om för dig att ”Raw Sockets” är en väldigt kraftfull sockettyp och att du borde kolla upp den.

Ok då. Vilka är de två typerna? En är ”Stream Sockets”; och den andra är ”Datagram Sockets”, vilken härefter kommer refereras till som ”SOCK_STREAM” respektive ”SOCK_DGRAM”. Datagram sockets kallas ibland för ”anslutningslösa sockets” (Men de kan connect();a om du verkligen vill. Se connect(), nedan).

Streamsockets är bra och pålitliga tvåvägs anslutnings kommunikations strömmar. Om du skriver ut två saker till socketen i ordningen ”1,2” kommer de att komma fram i samma ordning, ”1,2”, i andra änden. De kommer också vara felfria. De fel som du möter på är fantasier, skapande av din egen vrickade hjärna och kommer inte att diskuteras här.

Vad använder streamsockets? Tja, du kanske har hört talas om Telnet applikationen, eller? Den använder stream sockets. Alla tecken som du skriver ner kommer fram i samma ordning som du skrev ner dem i. En Webbläsare använder HTTP protokollet som använder stream sockets för att hämta sidor. Om du telnetar till en webbsida på port 80 och skriver ”GET /”, kommer det att hämta HTML koden tillbaks till dig.

Hur gör streamsocket för att nå den höga kvalitén på dataöverföring? Det använder ett protokoll som kallas för ”The Transmission Control Protocol”, annat känt som TCP (se RFC-793[5] för extremt detaljerad information om TCP). TCP ser till att din data kommer fram sekventiellt och felfritt. Du kanske har hört ”TCP” före dess bättre hälft av ”TCP/IP” där ”IP” står för ”Internet Protocol” (se RFC-791[6]). IP tar hand om Internetrouting och är inte ansvarig för datatillståndet (data integrity //övers. anm.)

figur 1. Datainkapsling

Häftigt! Och Datagramsockets? Varför kallas de ”anslutningslösa”? Vad är det frågan om, egentligen? Varför är inte de pålitliga? Tja, så här är det; Om du sänder ett datagram *kanske* det kommer fram. Det kanske kommer fram i en annan ordning. Men om det kommer fram, så är datan i paketet felfritt.

Datagramsockets använder också IP för routing, men de använder inte TCP, utan ”User Datagram Protocol”, eller ”UDP” (se RFC-768[7]).

Varför är de anslutningslösa? Jo, helt enkelt för att du inte behöver hålla en öppen anslutning som du måste med streamsocket. Du bara bygger ett paket, slänger in en IP-header med destinationsinformation, och skickar iväg det. Ingen anslutning behövs. De används oftast för paket-för-paket överföring. Exempel på applikationer: **tftp, bootp**, etc.

”Ok!”, skriker du nu troligtvis till. ”Hur kan dessa program ens fungera om datagrams kan gå förlorade?!”. Nåja, min mänskliga vän, varje paket har sitt eget protokoll ovanpå UDP. Till exempel, tftp protokollet säger till för varje paket som sänds, att mottagaren måste skicka tillbaka ett packet som säger, ”Jag har det!” (Ett ”ACK” packet), Om sändaren av originalpaketet inte får svar inom, låt oss säga, fem sekunder, kommer sändaren att skicka

paketet en gång till, tills ett ACK paket har mottagits. Denna bekräftelseprocess är mycket viktigt när man använder SOCK_DGRAM applikationer.

2.2 Lågnivånonsens och nätverksteori

Eftersom jag nämnde lager av protokoll är det dags att prata om hur nätverk verkligen fungerar. Det skall också visas hur SOCK_DGRAM-paket byggs. Du kan troligtvis hoppa över det här kapitlet, fast det är bra att känna till bakgrunden.

Hejsan barn, Nu är det dags att lära sig om *datainkapsling*! Detta är mycket viktigt. Det är så viktigt att du till och med skulle ha lärt dig det om du tog en nätverkskurs här på Chico State ;-). Enkel kan man säga att; ett paket föds, paketet blir omslaget (”inkapslat”) i en header (och i enstaka fall en footer), av det första protkollet (låt oss säga TFTP). Sen blir hela paketet (inklusive TFTP headern) är inkapslad igen av nästa protokoll(säg UDP), och sedan igen av nästa (IP). Och tillslut det slutgilltlika protokollet i det fysiska lagret (exempelvis Ethernet).

När den andra datorn tar emot paketet, tar hårdvaran bort Ethernet headern, kärnan tar bort IP och UDP-headern. Sist tar Tftp-programmet bort TFTP-headern för att sedan komma åt datan.

Nu kan vi äntligen tala om den berömda OSI-modellen (Layered Network Model). Den här nätverksmodellen beskriver ett system för nätverksfunktioner och har många fördelar mot andra modeller. Till exempel kan du skriva socketprogram som är exakt lika utan att behöva bry dig om hur datan fysiskt är överförd (serial, tunn Ethernet, AUI, eller vad som helst), därför att det finns andra program på lägre nivåer som gör det åt dig. Det egentliga nätverket och topologin är oväsentligt för socketprogrammeraren.

Utan mer tjafs presenterar jag nu modellen med alla dess lager. Kom ihåg den inför din nästa nätverkstenta:

- Applikation
- Presentation
- Session
- Transport
- Nätverk
- Data-länk
- Fysiskt

Det fysiska lagret är hårdvaran (seriell, Ethernet, etc.). Applikationslagret är så långt ifrån det fysiska lagret man kan komma – Det är där användaren integrerar med nätverket.

Den här modellen är så generell att man kan använda den som en reparationsguide för bilen, om man vill det. En modell mer enhetlig med Unix skulle se ut som följande:

- Applikationslager (*telnet, ftp, etc.*)
- Host-to-host transportlager (TCP, UDP)
- Internetlager (IP and routing)
- Nätverksaccesslager (Ethernet, ATM, eller nått annat)

Nu kan du förhoppningsvis se hur dessa lager förhåller sig till inkapsling av data.

Ser du hur mycket arbete det är att bygga ett enkelt packet? Jösses! Och du måste bygga in packetheader själv genom att använda ”**cat**”! Skojade bara! Allt du behöver göra för stream-sockets är att send()a ut datan. Allt du behöver göra för att datagram sockets är att inkapsla paketet med valfri metod och anropa sendto(). Kärnan bygger på Transportlagret och Internetlagret och hårdvaran bygger på Nätverksaccesslagret. Tack gode Gud för modern teknologi.

Och så slutar vår korta resa in i Nätverksteori. Oh ja, Jag glömde allt det jag ville säga om routing: Inget. Just det, jag kommer inte att gå in på det alls. Routern skalar paketet till IPheadern, konsulterar dess routingtabell, bla bla bla. Titta i IP RFC[8] om du vill veta. Om du aldrig lär dig något om det...tja, du kommer överleva i alla fall.

3. structs och datahantering

Ok, då är vi äntligen här. Det är dags att prata om programmering. Det här kapitlet går igenom olika datatyper som används av socketgränssnittet, så en del av dem kan vara omöjliga att lista ut på egen hand.

Den fösta är enkel: En socketdeskriptor. En socketdeskriptor är av följande typ:

```
int
```

En helt vanlig int.

Sedan blir det värre, så läs bara vidare och lid med mig. Det här bör du känna till: Det finns två olika byteordningar: den mest signifikanta (ibland kallad ”oktett” eller ”octet”) först eller den minst signifikanta först. Den förra kallas för ”Network Byte Order”. En del maskiner lagrar deras nummer internt i Network Byte Order, andra inte. När jag säger att något skall vara i Network Byte Order, så måste du anropa en funktion (så som htons()) för att ändra från ”Host Byte Order”. Om jag inte säger Network Byte Order, så skall du låta värdet vara i Host Byte Order.

(För den vetgirige, ”Network Byte Order” är också känt som ”Big-Endian Byte Order”)

Min Första Struct[tm] – struct sockaddr. Den här strukten innehåller adressinformation för många typer av sockets:

```
struct sockaddr {
    unsigned short    sa_family;    // Adressfamilj, AT_xxx
    char              sa_data[14];      // 14 bytes protokolladress.
};
```

sa_family kan vara en hel massa saker, men i det här dokumentet kommer vi bara att använda AF_INET. *sa_data* innehåller destinationsadress och portnummer för socketen. Detta är ganska klumpigt då det är ganska tröttsamt att peta in adressen för hand till *sa_data*.

För att kunna hantera struct sockaddr skapade programmerare en parallell struct: struct sockaddr_in (”in” för Internet).

```
struct sockaddr_in {
```

```
    short int          sin_family;  // Address familj
    unsigned short int sin_port;    // Port nummer
    struct in_addr     sin_addr;    // Internet adress
    unsigned char      sin_zero[8]; // Samma storlek som struct sockaddr
};
```

Den här strukturen gör det enkelt att referera till socketadressen. Notera att *sin_zero* (vilken följer med för att lägga strukturen till struct sockaddr's längd) bör sättas till nollor med funktionen memset(). Dessutom, och detta är den *viktiga* biten, kan en pekare till en struct sockaddr_in fungera som en pekare till struct sockaddr och vica-versa. Så även om socket() vill ha en struct sockaddr*, så kan du fortfarande använda struct sockaddr_in och omvandla den i sista sekunden! Du bör också notera att *sin_family* motsvarar *sa_family* i en struct_sockaddr och bör bli satt till "AF_INET". Slutligen måste *sin_port* och *sin_addr* vara i *Network Byte Order*.

"Men, " säger du, "hur kan hela strukturen, struct in_addr sin addr, vara i Network Byte Order". Denna fråga kräver en mer noggrann undersökning av strukturen struct in_addr, en av de värsta union:erna som funnits:

```
// Internet address (En struktur av historiska skäl)
struct in_addr {
    unsigned long s_addr;
};
```

Nåja, det *brukade* vara en union, men det ser ut som om så inte längre är fallet. Bra! Så om du deklarerar *ina* som en struct sockaddr_in, så refererar *ina.sin_addr.s_addr* till en 4-bytes IP adress (i Network Byte Order). Notera om ditt system fortfarande använder den Gudsavskydde unionen för *struct in_addr*, så kan du fortfarande referera till 4-bytes adressen precis som jag gjorde ovan (detta tack vare #define).

3.1 Konvertera det inhemska

Vi har i och med det kommit till nästa kapitel. Det har varit mycket snack om det där Network to Host Byte Order konvertering – Nu är det dags för lite action.

Ok, Det finns två typer som du kan konvertera: short (två bytes) och long (fyra bytes). De här funktionerna fungerar lika bra med unsigned varianterna. Låt oss säga att du vill konvertera short från Host Byte Order till Network Byte Order. Börja med ett "h" för "host", lägg till ett "to", sedan ett "n" för "network" och till sist ett "s" för "short": h-to-n-s, eller htons() (läs: "Host to network Short").

Det är nästan för enkelt…

Du kan använda alla kombinationer av "n", "h", "s" och "l", om man inte räknar de riktigt dumma Till exempel finns det INTE en stolh() ("Short to Long Host") funktion – inte på den här festen åtminstone. Men det finns:

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

Nu tror du har hajat allt och tänker "Med om jag behöver byta ordning på en char?". Sen kanske du tänker "Ööh, glöm det". Du kanske också tänker att då redan 68´000 maskiner använder Network Byte Order, behöver du inte anropa htonl() för din IP-adress. Då har du rätt, MEN om du försöker porta ditt program till en maskin som använder omvända Byte Order, så kommer du att misslyckas. Var flexibel! Det här är en Unix-värld! (Även om Bill Gates gärna vill tro något annat). Komihåg: Sätt dina bytes i Network Byte Order före du släpper ut dem på nätverket.

En sista punkt: Varför behöver *sin_addr* och *sin_port* vara i Network Byte Order i en struct *sockaddr_in*, men inte *sin_family*? Svar: *Sin_addr* och *sin_port* blir inkapslade i IP respektive UDP lagren. Dessa måste vara i Network Byte Order. Men *sin_family* fältet används endast av kärnan för att ta reda på vad för typ av adress som strukturen innehåller, så den måste vara i Host Byte Order. Alltså, eftersom *sin_family inte* skickas ut på närverket, kan den vara i Host Byte Order.

3.2 IP adresser och hur man hanterar dem

Som tur är finns det en mängd funktioner som manipulerar IPadresser. Det finns därför ingen anledning att sitta och för hand trycka in dem i en long med <<-operatorn

För det första, låt oss säga att du har en struct sockaddr_in ina, och att du villl lagra IPadressen "10.12.110.57" i den. Funktionen du ska använda, inet_addr(), konvererar en IP adress från nummer och punkter till en unsigned long. Tilldelningen ser ut som följande:

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

Notera att inet_addr() returnerar adressen i Network Byte Order – Du behöver inte därför anropa htonl(). Alla tiders!

Ok, kodsnutten ovan är inte speciellt stabil, eftersom den saknar felkontroll. Du ser, inet_addr() returnerar ett –1 vid fel. Kommer du ihåg de binära talen? (unsigned –1) råkar bara motsvara IP-adressen 255.255.255.255! Det är ju broadcast-adressen. Kom ihåg att göra din felkoll.

Faktiskt så finns det ett renare gränsitt som du kan använda istället för inet_addr(): Det kallas för inet_aton() ("aton" står för "ascii to network"):

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

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

Här är ett exempel på användning, där ett struct sockaddr_in paketeras (Du kommer förstå det här exemplet bättre när du kommer till kapitlen bind() och connect()).

```
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); // Nollställ resten av structen
```

inet_aton(), *olikt praktiskt taget resten av alla andra socket-relaterade funktioner*, returnerar icke-noll när det lyckas och noll när det misslyckas (Om någon vet varför, så säg till mig då). Och adressen returneras som *inp*.

Tyvärr så finns inte inet_aton() på alla plattformar, så trotts att inet_aton är mer fördelaktig, så kommer den äldre, mer vanliga, inet_addr att användas i den här guiden.

Ok, nu kan du konvertera en IP-adress av typen sträng till dess binära motsvarighet. Men om man vill göra tvärt om? Om du har en struct in_addr och vill skriva ut den i nummer- och punktform? I det här fallet använder du funktionen inet_ntoa() (”ntoa” står för ”network to ascii”), så här:

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

Detta kommer att skriva ut IP-adressen. Notera att inet_ntoa() tar en struct in_addr som argument och inte en long. Notera också att den returnerar en pekare till en char. Denna pekar till en statisk char vektor i inet_ntoa(), så varje gång du anropar inet_ntoa() kommer den senast IP-adressen skrivas över. Till exempel:

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

kommer att skriva ut:

```
adress 1: 10.12.110.57
adress 2: 10.12.110.57
```

Om du behöver spara adressen, strcpy():a den till din egen teckenvektor.

Det var allt om detta för tillfället. Senare kommer du dock att få lära dig att konvertera strängar som ”whitehouse.gov” till en motsvarande IP-adress (se DNS nedan).

4. System anrop eller ”Bust”

I det här kapitlet kommer jag att gå igenom de systemanrop som ger till tillgång till nätverksfunktionerna i Unix. När du gör ett systemanrop tar kärnan över och gör allt jobb automatiskt.

Det som de flesta har svårt för är i vilken ordning som man skall göra anropen. Med det problemet kan inte **man**-sidorna hjälpa dig, som du troligen har upptäckt. För att hjälpa dig i den här fruktansvärda situationen har jag försökt att presentera systemanropen i *exakt* (nästan) den ordningen de skall anropas.

Det, tillsammans med lite kodexempel här och där, lite mjölk och kakor (som du tyvärr får stå för själv), samt lite stake och mod, kommer du att skicka data omkring på Internet som en vettvilling.

4.1 socket() – Hämta fildeskriptorn

Nu tror jag inte att jag kan hålla tillbaka det längre – Jag måste bara prata om socket() anropet. Här är ett utdrag:

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

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

Men vad betyder argumenten? För det första, *domain* bör sättas till ”AF_INET”, precis som i struct sockaddr_in (se oven). Sedan talar *type* argumentet om för kärnan om vilken typ av socket det rör sig om: SOCK_STREAM eller SOCK_DGRAM. Till sist, ange bara *protcol* till ”0”. När *protocol* är satt till 0 väljer socket() själv rätt protocol baserad på ineehållet i *type*. (Notera: Det finns många fler *domain* än de jag har listat här. Det finns också fler *types* än jag listat. Se socket()s mansida. Det finns också ett “bättre” sätt att få *protcol*typ. Se man-sidan för getprotobyname()).

socket() returnerar helt enkelt en socketdeskriptor som du senare kan använda i ditt system. Vid fil returneras–1. Den globala variabeln *errno* sätts också till *error*s värde. (see perrors mansida).

I viss dokumentation kan en mystisk “PF_INET” nämnas. Detta är ett underligt, overkligt monster, som sällan ses i naturligt tillstånd, men det är lika bra att jag förklarar det här protokollet. För länge sedan var det tänkt att varje adressfamilj (det är vad ”AF” i AF_INET står för) skulle ha stöd för ett flertal protokoll, som var relaterad till deras protokollfamilj (PF som i PF_INET). Så blev det inte. Okej, så rätt sak att göra är att använda AF_INET i din struct sockaddr_in och PF_INET när du anropar socket(). Fast i praktiken kan du använda AF_INET överallt. Och eftersom det är vad W. Richard Stevens gör i sin bok, kommer vi att göra det här också.

Ja,ja,ja, men hur bra är socket()? Svaret är att den är totalt oanvändbar när den används för sig själv. För att det skall få användning av din nyvunna kunskap är du tvungen läsa vidare.

4.2 bind() – Vilken port befinner jag mig på?

När du väl har en socket måste du koppla ihop den med en port på din lokala maskin. (Detta görs vanligtvis när du skall lystna efter inkommande förbindelse på en bestämd port – MUDar gör det när de talar om för dig att ”telneta till x.y.z port 6969”). Portnumret används av kärnan för att matcha ett inkommande paket med en viss process´s socketdeskriptor. Om du bara kommer att connect()a är detta inte nödvändigt. Men läs det gärna ändå, för att få en kick.

Här är en översikt av systemanropet bind():

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

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

sockfd är socket-fildeskriptorn som returnerades av socket(). *my_addr* är en pekare till en struct sockaddr som innehåller information om din adress, nämligen port och IP-adress. *addrlen* kan sättas till sizeof (struct sockaddr).

Pust, Det var mycket att ta till sig på en gång! Vi behöver ett exempel:

```
#include <string.h>
#include <sys/types.h>
#include <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); // gör lite felkoll här

    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); // Nollor i resten av stuct:en

    // Glöm inte att felkolla bind()
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
    .
    .
    .
```

Det är några saker som man bör notera här: *my_addr.sin_port* är i Network Byte Order. Det är även *my_addr.sin_addr.s_addr*. En annan sak man bör se upp för är att headerfilerna kan skilja sig från system till system. Var noga att kontrollera med dina lokala **man**-sidor.

Till sist, under ämnet bind(), bör jag nämna att vissa delar i skaffandet av den egna IP-adressen och/eller porten kan automatiseras.:

```
my_addr.sin_port = 0; // slumpvis välj en ledig port.
my_addr.sin_addr.s_addr = INADDR_ANY; // Använd min adress.
```

Genom att sätta my_addr.sin_port till noll, säger du till bind() att välja en port åt dig. Likaså, genom att sätta *my_addr.sin_addr.s_addr* till INADDR_ANY, säger du att den skall automatiskt fylla i IP-adressen för maskinen processen körs på.

Om du har läggningen att lägga märke till småsaker, så kanske du lade märke till att jag inte angav INADDR_ANY i Network Byte Order. Dumma mig! Men jag har lite insideinformation. INADDR_ANY är faktiskt noll! Noll har fortfarande noll bitar även om du omorganiserar dem. Hur som helst kommer purinister att påpeka att det kan finnas ett parallellt universum där INADDR_ANY är, låt oss säga, tolv och att min kod inte fungerar där. Det är ok med mig:

```
my_addr.sin_port = htons(0); // slumpvis välj en ledig port.
```

```
my_addr.sin_addr.s_addr = htonl(INADDR_ANY); // Använd min adress.
```

Nu är vi så portabla att du inte kan tro det. Jag måste dock påpeka att den mesta koden du kommer att påträffa kör inte INADDR_ANY genom htonl().

bind() returnerar –1 vid fel och sätter errno till errors värde.

En annan sak som du bör se upp med när du anropar bind(): Välj inte ett för lågt nummer till din port. Alla portar under 1024 är RESERVERADE (såvida du inte är superuser)! Du kan använda alla portnummer större än 1024, ända upp till 65535 (såvida de inte redan används av ett annat program).

Ibland, som du kanske har märkt, när du försöker köra bind() en andra gång så misslyckas du med motivationen: ”Adressen används redan” (”Address already in use”). Vad betyder det? Jo, en bit av socketen som var ansluten hänger fortfarande kvar i kärnan och håller kvar vid porten som den blivit tilldelad. Du kan antingen vänta tills porten blir ledig (en minut eller något sådant) eller lägga till kod i ditt program som gör att du kan återanvända en port, så här:

```
int yes=1;
                //char yes='1'; // Solarisfolk använder detta

// Ta bort "Address already in use" felmedelandet
if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
    perror("setsockopt");
    exit(1);
}
```

En liten extra sista notering angående bind() : Det finns tillfällen som du absolut inte måste anropa den. Om du anropar en maskin och inte bryr dig om vad din lokala port är (som i fallet **telnet** där du bara bryr dig om porten på den andra maskinen) kan du helt enkelt anropa connect(). Den kollar om socketen är obunden och binder den till en ledig port om det blir nödvändigt.

4.3 connect() – Hej där!

Låt oss låtsas för ett tag att du är en Telnet applikation. Din användare ger dig instruktioner (precis om i filmen TRON) att hämta en socket-fildeskriptor. Du lyder och anropar socket(). Sen talar din användare om för dig att skall ansluta till ”10.12.110.57” på port ”23” (Telnets standard port). Hups! Vad gör du nu?

Tur för dig att du nu läser igenom kapitlet som handlar om connect() – hur du ansluter till en värddator. Så läs genast vidare. Ingen tid att förlora!

connect()-anropen är som följande:

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

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

sockfd är vår kära vän socket-fildeskriptor, som returneras av socket()-anropet. *serv_addr* är en struct sockaddr som innehåller destinationsadress och portnummer och *addrlen* kan sättas till sizeof(struct sockaddr).

Är det svårt? Vi behöver ett exempel:

```
#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;   // Denna besitter destinationsadressen

    sockfd = socket(AF_INET, SOCK_STREAM, 0); // Felkolla!

    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);  // Nollor i resten av structen

    // Glöm inte att felkolla connect()
    connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
.
.
.
```

Och glöm inte att kontrollera returnvärdet från connect() – det är –1 vid fel och sätter variabeln *errno*.

Notera också att vi inte anropade bind(). Detta då vi inte bryr oss om vårt lokala portnummer utan endast portnumret på den värddatorn. Kärnan väljer en lokal port åt oss och maskinen som vi ansluter till får automatiskt reda på vilken port det blev. Inga bekymmer.

4.4 listen() – Kan ingen kontakta mig?

Ok, dags att byta tempo. Men om vi inte vill göra en fjärranslutning då?. Låt oss säga, bara för att skojs skull, att vi bara skulle vilja vänta på inkommande anslutningar och hantera dem på något sätt. Den processen delas upp i två steg: först måste du listen() och sedan accept() (se nedan)

Anropet till listen() är ganska enkelt, men behövs ändå förklaras:

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

sockfd är som vanligt socket-fildeskriptorn som kommer från socket()-anropet. *backlog* är det antal anslutningar som är tillåtna i anslutningskön. Det är så att inkommande anslutningar måste vänta tills du accept()erar dem (se nedan) och detta är gränsen för hur många

anslutningar som kan ställas i kö. De flesta system sätter denna gräns till ungefär 20, men du kommer troligtvis undan med fem eller tio.

Igen, som vanligt, returnerar listen() – 1 och sätter errno vid fel.

Som du kanske förstår, så måste vi anropa bind() före vi kan anropa listen(). Annars kommer kärnan att lyssna på en slumpmässigt vald port. Blä! Om du skall lyssna efter inkommande anslutningar, så måste du göra anropen i denna ordning:

```
socket();
bind();
listen();
/* här kommer accept() */
```

Jag kommer inte att förklara koden ovan eftesom jag tycker att det är gjorts tillräckligt nu. (Koden i accept()-kapitlet är mer komplett). Det riktigt problematiska är att få hela schabraket att anropa accept().

4.5 accept() – ”Tack för att du ansluter till port 3490”

Var beredd – accept() anropet är lite underligt! Detta är vad som kommer att hända: Någon långt långt borta kommer att försöka connect()a till din maskin på den port som du har kopplat listen() till. Deras anslutning kommer att köas och väntar på att accpet()eras. Du anropar accept() och säger till den att ta hand om den aktuella anslutningen. Denna kommer att returnera en helt ny socket-fildeskriptor enbart för denna anslutning! Just det, helt plötsligt har du *två socket-fildeskriptorer* för priset av en! Den ursprungliga lyssnar fortfarande på din port och den nya är tillslut klar för att använda send() och recv(). Nu är vi framme!

Anropet ser ut som följande:

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

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

sockfd är den lyssnande socketdeskriptorn. Enkelt! *addr* kommer vanligtvis vara en pekare till en lokal struct sockaddr_in. Detta är var informationen om den inkommande anslutningen kommer att hamna (och med den kan du ta reda på vad för maskin som anropar vilken port). *addrlen* är en lokal variabel som bör sättas till sizeof(struct sockaddr_in) innan den förs vidare till accept(). *accept()* kommer inte att stoppa in fler än *sizeof* många bytes i *addr*. Om *accept()* stoppar in färre bytes, så ändras storleken för att matcha antalet bytes.

Och gissa vad! accept() returnerar –1 och sätter errno om ett fel påträffas. Det kunde du inte list ut!

Som tidigare blev det mycket att ta till sig på en gång, så här kommer ett litet exempel som du kan ögna igenom:

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

```
#define MYPORT 3490    // Porten som användare ansluter till

#define BACKLOG 10     // Hur många väntande anslutningar åt gången

main()
{
    int sockfd, new_fd;  // syssna på sock_fd, ny anslutning på new_fd
    struct sockaddr_in my_addr;    // min adressinformation
    struct sockaddr_in their_addr; // anslutarens adressinformation
    int sin_size;

    sockfd = socket(AF_INET, SOCK_STREAM, 0); // Lägg till felkontroll

    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; // Fyll i min IP automatiskt
    memset(&(my_addr.sin_zero), '\0', 8); // Nollor i resten av strukten

    // Glöm inte din felkontroll
    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);
    .
    .
    .
```

Notera att vi använder socketdeskriptorn *new_fd* för alla send() och recv() anrop. Om du enbart kommer att få en anslutning kan du använda close() för att stänga *sockfd* för att hindra fler inkommande anslutningar på samma port.

4.6 send() och recv() – Tala med mig, sötnos!

Dessa två funktioner är till för att kommunicera över streamsocket eller en ansluten datagramsocket. Om du vill använda vanliga oanslutna datagramsockets, behöver du läsa kapitlen sendto() och recvfrom() nedan.

send() anropet:

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

sockfd är socketdeskriptorn som du vill skicka data till (oavsett om det är den som returnerades av socket() eller den du fick från accept()). *msg* är en pekare till den data som du vill skicka och *len* är längden på den data i bytes. Sätt *flags* till 0 (Se send()-mansida för mer information angående flags).

Exempelkoden skulle kunna se ut så här:

```
char *msg = "Beej was here!";
int len, bytes_sent;
.
.
```

```
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
.
.
.
```

send() returnerar det antal bytes som faktiskt har sänds – *detta kan vara mindre än det du sa skulle sändas.* Du ser, ibland talar du om för den att skicka ut hela högar av data som inte kan hanteras. send() kommer att skicka ut all den data den kan och lita på att du skickar ut resten senare. Komihåg det, att om returvärdet från send() inte är lika stort som värdet *len*, så är det upp till dig att skicka resten av strängen. De goda nyheterna är att om paketet är litet (mindre än ett K eller så) kommer det *troligtvis* att skickas iväg på en gång. Och ännu en gång så returneras –1 vid fel och *errno* sätts till felnumret.

recv() är lik i många avseenden.

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

sockfd är socketdeskriptorn som skall läsas ifrån, *buf* är bufferten där den lästa informationen lagras. *len* är maxlängden på bufferten och *flags* kan återigen sättas till 0 (se recv()-mansidor för flag information).

recv() returnerar det antal bytes som har blivit lästa, eller returnerar –1 (med tillhörande *errno*).

Vänta! recv() kan även returnera 0. Detta kan bara betyda en sak: maskinen du försökt ansluta till har stängt anslutningen för dig! När du har fått ett returvärde på 0 vet du att detta har hänt.

Sedär, det var enkelt, eller hur? Du kan nu skicka fram och tillbaka data med hjälp av streamsocket. Häftigt, Du är en Nätverksprogrammerare i Unix.

4.7 sendto() och recvfrom() – Tala med mig, DGRAM-viset

Nu säger du troligtvis att ”Det var ju trevligt, men jag har inte lärt mig något om oanslutna datagramsocket!”. Inga problem, amigo. Jag har allt du behöver

Då datagramsockets inte är anslutna till något fjärran dator, så gissa vilken information vi måste ange innan vi skickar ett paket? Just det! Destinationsadressen! Så här ser den ut:

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

Som du ser är detta precis som att anropa send(), förutom de två sista argumenten. *to* är en pekare till struct sockaddr (som du troligtvis har som en struct sockaddr_in och omvandlar den). vilken innehåller destinationens IP-adress och port. *tolen* sätter du helt enkelt till sizeof(struct sockaddr).

Precis som send() så returnerar sendto() antalet bytes som har sänts (som även här kan vara mindre än vad du har angett!), eller –1 vid fel.

Samma sak gäller för recv() och recvfrom(). Ett utdrag från recvfrom():

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

Detta är precis som recv() med undantaget de två extra argumenten. *from* är en pekare till en lokal struct sockaddr som kommer bli ifylld med IP-adressen och port från den ursprungliga maskinen. *fromlen* är en pekare till en lokal int som bör vara initierad som en sizeof (struct sockaddr). När funktionen returnerar, kommer *fromlen* att innehålla längden på adressen som är lagrad i *from*.

recvfrom() returnerar antalet bytes som den mottagit, eller –1 vid fel (med tillhörande *errno*)

Komihåg att om du connect()ar till en datagramsocket så kan du helt enkelt använda send() respektive recv() för alla överföringar. Socketen är fortfarande en datagramsocket och paketet kommer fortfarande att använda sig av UDP, men socketgränssnittet kommer automatiskt att lägga till destination- och källinformation åt dig.

4.8 close() och shutdown() – Stick och brinn

Pust! Nu har du skickat och tagit emot data hela dagen lång, och börjar faktiskt bli ganska trött på det. Du är redo för att stänga socketdeskriptorn. Det är enkelt. Du behöver bara använda den vanliga close()-fildeskriptorn som finns i Unix.

```
close(sockfd);
```

Detta gör att det varken går att läsa eller skriva till socketen. Alla försök till att skriva eller läsa till socketen kommer att misslyckas.

Bara i fall att du vill ha lite mer kontroll över hur du stänger av socketen, kan du använda shutdown(). Den möjliggör att du kan stänga av all trafik i en viss riktning, eller åt båda hållen (som close() gör). Ett litet utdrag:

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

sockfd är socket-fildeskriptorn som du vill stänga av, och *how* är följade:

- 0 – Inga fler mottagningar är tillåtna
- 1 – Inga fler sändningar är tillåtna
- 2 – Varken sändningar eller mottagningar är tillåtna (som close())

shutdown() returnerar 0 vid lyckad stängning och –1 vid fel (med tillhörande *errno*).

Om du behagar använda shutdown() på oanslutna datagramsockets, så kommer socketet bli otillgänglig för alla vidare send() och recv() anrop (komihåg att du använder dessa när du använder connect() med datagramsockets).

Det är viktigt att veta att shutdown() faktiskt inte stänger fildeskriptorn – Det ändrar bara dess användningsmöjlighet. För att frigöra en socketdeskriptor måste du använda close().

Inget mer att tillägga!

4.9 getpeername() – Vem är du?

Den här funktionen är näsan för enkel!

Den är så enkel att det var nära att den inte fick ett eget kapitel. Men här är det i alla fall.

Funktionen getpeername() talar om för dig vem som är på andra sidan på en ansluten streamsocket. Utdrag:

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

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

sockfd är en deskriptor för en ansluten streamsocket. *addr* är en pekare till en struct sockaddr(eller en struct sockaddr_in) som innehåller information om anslutningens andra ände. Tillsist är *addrlen* en pekare till en int, som bör vara initierad till en sizeof(struct sockaddr).

Funktionen returnerar –1 vid fel och sätter *errno*, precis som det skall vara.

När du väl har deras adress kan du använda inet_ntoa() eller gethostbyaddr() för att skriva ut eller få mer information. Och Nej! Du kan inte se deras inloggningsnamn. (Ok,ok. Om den andra datorn kör en identdeamon, är det möjligt. Detta ligger dock utanför detta dokument. (Se RFC-1413[9] för mer information).

4.10 gethostname() – Vem är jag?

Ännu enklare än getpeername() är gethostname(). Den returnerar namnet på den dator som ditt program körs på. Detta namn kan användas av gethostbyname() (nedan) för att ta reda på din lokala maskins IP-adress.

Finns det något som är roligare? Jag kan möjligtvis tänkta mig ett par saker, men de har inte med socketprogrammering att göra. I vilket fall som helst, Här är ett utdrag:

```
#include <unistd.h>

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

Argumentet är lätta: *hostname* är en pekare till en vektor av typen char som kommer att tilldelas datorns namn av funktionen. *size* är längden på hostname-vektorn.

4.11 DNS – Du säger ”whitehouse.gov”, Jag säger ”198.137.240.92”

Ifall du inte vet vad DNS är, så står det för ”Domain Name Service”. Enkelt kan man säga att du ger DNSen den människovänliga webbadressen, och du kommer att få IP-adressen (så att du kan använda bind(), connect(), sendto() eller vad du nu har tänkt). När någon skriver in:

```
$ telnet whitehose.gov
```

så kan Telnet hämta det som behövs för att connect()a till ”198.137.240.92”.

Men hur fungerar det? Du kommer att använda funktionen gethostnyname():

```
#include <netdb.h>

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

Som du ser returnerar den en pekare till en struct hostent, som ser ut så här:

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

Och här är förklaringen för de fält som struct hostent innehåller:

- h_name – Värdens officiella namn
- h_aliases – En NULL-ställd vektor med ett alternativt namn på värden
- h_addrtype – Typen på adressen som returneras, vanligtvis AF_INET
- h_lenght – Längden då adressen i bytes
- h_addr_list – En nollad vektor för värdens nätverksadress. Adressen är i Network Byte Order
- h_addr – Den första adressen i *h_addr_list*

gethostbyname() returnerar en pekare till den ifyllda struct hostent, eller NULL vid fel (Men *errno* är inte tilldelad – *h_errno* används här. Se herror() nedan).

Men hur används detta? Ibland (som vi kan läsa i datormanualer) räcker det inte att bara spy ut information till läsaren. Den här funktionen är betydligt lättare än vad den verkar.

Här är ett exempelprogram[10]:

```
/*
** getip.c – Demo för att kolla up hostname
*/

#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) {  // Felkontroll
        fprintf(stderr,"usage: getip address\n");
        exit(1);
    }
```

```
    if ((h=gethostbyname(argv[1])) == NULL) {  // Hämta information om värddatorn
        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;
}
```

Med gethostbyname() kan du inte använda perror() för att skriva ut felmedelanden (då *errno* inte används) utan du måste anropa herror().

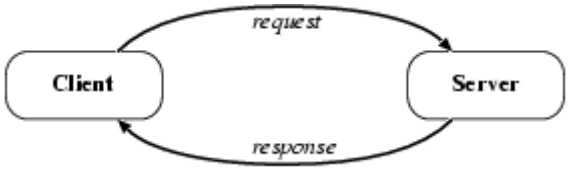
Det här är ganska klart! Du skickar helt enkelt en sträng, som innehåller maskinens namn (”whitehouse.gov”), till gethostbyname() och sen hämtar ut informationen från den struct hostent som returneras.

Den enda underliga saken är möjligtvis i utskrivandet av IP-adressen. I koden ovan så är *h->h_addr* en *char**, men inet_ntoa() vill ha en struct in_addr. Så jag omvandlar *h->h_addr* till en *struct in_addr** och sedan omrefererar den för att komma åt datan.

5. Bakgrund till Klient-Server

Vi lever i en klient-servervärld, sötnos. Allt som har med nätverk att göra består av en klient-process som talar med en server-process. Ta **Telnet**, till exempel. När du ansluter till en värddator på port 23 med Telnet (klienten), så vaknar ett program (Telnetd, servern) till liv. Detta program hanterar den inkommande anslutningen, visar loginprompten, etc.

Utbytandet av information mellan klient och server summeras i Figur 2.



Figur 2. Klient-Server interaktion

Notera att klient-serverparet kan tala SOCK_STREAM, SOCK_DGRAM, eller vad som helst (så länge ta talar samma sak). Bra exempel på klient-serverpar är **telnet/telnetd, ftp/ftpd** eller **bootp/bootpd**. Varje gång som du använder **ftp**, så kommer serverprogrammet **ftpd**, att betjäna dig.

Vanligast är att det bara finns en server på en maskin och att servern hanterar flertalet klienter med hjälp av fork(). Så här fungerar det: Servern väntar på en anslutning, accept()erar den, och fork()ar en barnprocess för att hantera den. Det är vad vår exempelserver gör i nästa kapitel.

5.1 En enkel streamserver

Allt denna server gör är att skicka strängen ”Hello World!\n” över en streamanslutning. Allt du behöver göra för att testa servern är att köra den i ett fönster och telneta till den från ett annat med:

$telnet remotehostname 3490

där remotehostname är namnet på den maskin där du kör servern.

Serverkoden[11]:

```
/*
** server.c – Ett streamsocket-server 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    // porten som användaren ansluter till

#define BACKLOG 10     // Hur många väntande anslutningar servern kan ha

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

int main(void)
{
   int sockfd, new_fd;  // lyssna på sock_fd, nya anslutningar på new_fd
   struct sockaddr_in my_addr;    // min adressinformation
   struct sockaddr_in their_addr; // anslutningens adress information
   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; // Fyller automatiskt i min IP-adress
    memset(&(my_addr.sin_zero), '\0', 8); // Nollor i resten av structen

    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; // ta bort alla döda processer
    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()) { // Detta är barnprocessen
            close(sockfd); // Barnet behöver ingen lystnare
            if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
                perror("send");
            close(new_fd);
            exit(0);
        }
        close(new_fd);  // Föräldern behöver inte detta.
    }

    return 0;
}
```

Om du är nyfiken så har jag koden i en enda stor main()-sats för att få bättre syntaktisk översikt över koden.

(Det här med sigaction() kanske är nytt för dig – men det är ok. Den koden är till för att ta bort ”zombi”-processer som uppstår efter det att den fork()ade barnprocessen avslutas. Om du får en massa zombies och inte tar bort dem kommer din systemadministratör att bli irriterad).

Du kan få datan från denna servern genom att använda klienten som finns i nästa kapitel.

5.2 En enkel streamklient

Klienten är betydligt enklare än vad servern var. Allt den här klienten gör är att koppla upp sig mot den värd som du specificerade; på port 3490. Den får tillbaka den sträng som servern skickar.

Klient koden[12]:

```
/*
** client.c – En streamsocket-klient
*/

#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 // porten som klienten ansluter till

#define MAXDATASIZE 100 // Maxantalet bytes klienten kan ta emot på en gång

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

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

    if ((he=gethostbyname(argv[1])) == NULL) {  // Skaffa information om värden
        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), 8);  // Nollor i resten av structen

    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;
}
```

Notera att om du kör klienten innan du kört servern så får du ”Connection refused”. Mycket användbart.

5.3 Datagramsockets

Jag har egentligen inte mycket mer att ta upp angående detta så jag kommer endast att visa att par exempelprogram: talker.c och listner.c:

listener körs på en maskin, och väntar på inkommande paket på port 4950. **talker** skickar paket på den porten, till den specificerade maskinen, och väntar på att användaren skall skriva in något på kommandoraden.

Här är koden för listener.c[13]:

```
/*
** listener.c – En datagramsocket-server
*/

#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    //porten som användaren ansluter till

#define MAXBUFLEN 100

int main(void)
{
    int sockfd;
    struct sockaddr_in my_addr;    // min adressinformation
    struct sockaddr_in their_addr; // Anslutningens adressinformation
    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; // Fyller automatiskt i min IP-adress
    memset(&(my_addr.sin_zero), '\0', 8); // Nollor i resten av structen

    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;
}
```

Notera att i anropet till socket(), så använder vi äntligen SOCK_DGRAM. Notera också att det finns inget behov av listen() eller accept(). Det är en av fördelarna när man använder anslutningslösa datagramsockets.

Här kommer koden till talker.c[14]:

```
/*
** talker.c – Ett exempel på en datagramsocket-klient
*/

#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    // porten som användaren ansluter till

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; // Anslutningens adressinformation
```

```
    struct hostent *he;
    int numbytes;

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

    if ((he=gethostbyname(argv[1])) == NULL) {  // Hämta värdinformation
        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); // Nollor i resten av structen

    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;
}
```

Det är allt! Kör nu **listener** på en maskin och kör sedan **talker** på en annan. Se hur de kommunicerar. Roligt barntillåtet superskoj för hela den lilla kärnfamiljen!

Förutom en liten detalj som jag har nämnt många gånger tidigare: anslutna datagramsockets. Jag måste nog ta upp det här, då du nu läser kapitlet om datagramsocket. Låt oss säga att **talker** anropar connect() och anger **listener**s adress. Från den punkten kan **talker** bara sända och ta emot från den specifika adressen som angets till connect(). Av den anledningen behöver du inte använda sendto() och recvfrom(); du kan helt enkelt använda send() och recv().

6. Lite mer avancerade tekniker

De kanske inte är *så* avancerade, men de är mer avancerade än det grundläggande som vi redan har gått igenom. Faktum är att om du har gått så här långt så kan du anse dig hyggligt kunnig i grundläggande nätvärkprogrammering i Unix!
Grattis!

Så nu går vi in i en ny värld där du kommer att lära dig mer mystiska saker om sockets.
På det bara!

6.1 Blockering

Blockering. Du har hört talas om det men – vad i helsike är det? Förenklat, är ”block” teknikerslang för ”sleep”. Du noterade kanske att när du körde **listener** förut, så väntade det bara på att ett paket skulle komma. Vad som hände var att det recvfrom()anropades, som fann att det inte fanns någon data. Så recvfrom() blev tillsagt att ”blocka” (som innebär sov (sleep) där) till data kommer.

Flera av funktionerna blockerar. accept() blockerar. Alla recv()funktioner blockerar. Anledningen är att de kan göra det är att de är tillåtna att göra det. När du i först skapar en socketdeskriptor med socket(), sätter kärnan den på blockering. Om du inte vill att en socket skall blockera måste du anropa fcntl():

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

Genom att ange en socket till icke-blockering kan du effektivt fråga socketen efter information. Om du föröker läsa från en icke-blockerad socket och det inte finns någon data där, är det inte tillåtet att blockera – Så det kommer att returnera –1 och *errno* kommer att bli satt till EWOULDBLOCK.

Generellt är denna typ av utfrågning en ganska dålig idé. Om du sätter ditt program i ett ”upptaget- vänta, söker efter data på socketen”- läge kommer du att suga CPU-kraft utan dess like. En mer elegant lösning för att se om det finns data som väntas på att läsas kan du läsa om i nästa kapitel om select().

6.2 select() – Flera synkroniserade I/O

Den här funktionen är lite underlig, men mycket användbar. Ta följande situation: Du är en server och du vill lyssna efter inkommande anslutningar samtidigt som du vill läsa data från de anslutningar du redan har.

Inga problem, säger du, bara en accept() och några recv(). Inte så fort, grabben! Vad händer om du blockerar på ett accept() anrop? Hur skall du kunna ta emot data samtidigt? ”Använd icke-blockarande sockets!” – Nej du! Du vill inte ta hela CPUns kraft. Och nu då?

select() ger dig möjlighet att övervaka flera sockets samtidigt. Den här funktionen talar om för dig vilka sockets som är klara att läsas ifrån, och vilka som kan skrivas till. Om du vill så kan den även tala om vilka sockets som har kastat undantag.

Utan vidare fördröjning, presenterar jag ett utdrag ur select():

```
#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);
```

Den här funktionen övervakar ”fildeskriptor-listor”; i synnerlighet *readfds, writefds* och *exceptfds*. Om du vill se om du kan läsa från standardinput eller någon socketdeskriptor, *sockfd*, lägg till fildeskriptorerna 0 och *sockfd* till setet *readfs*. Parametern numfds bör vara satt till värdet av den högsta fildeskriptorn plus ett. I detta exempel skulle den vara satt till *sockfd+1*, då det är med säkerhet större än standardinput (0).
När select() returnerar, blir *readfds* modifierad för att reflektera vilken av fildeskriptorerna som du valde som är klara att läsas. Du kan testa dem med makrot FD_ISSET(), nedan.

Före jag fortsätter, måste jag ta upp hur man manipulerar dessa set. Varje set är av typen *fd_set*. Följande makron opererar på denna typ:

- FD_ZERO(fd_set *set) – rensar alla deskriptorset.
- FD_SET(int fd, fd_set *set) – lägger till fd till set
- FD_CLR(int fd, fd_set *set) – tar bort fd från set
- FD_ISSET(int fd, fd_set *set) – testar om fd finns med i set

Tillsist; vad är det där för underlig struct timeval? Ibland vill du inte vänta evigheter på att någon skall skicka data till dig. Du kanske vill skriva ut ”Håller fortfarande på…” till terminalen även om inget har hänt. Den här strukturen ger dig möjligheter att specificera timeoutperioder. Om den specificerade tiden har passerat och select() inte har funnit någon klar fildeskriptor, kommer den att returnera så att du kan fortsätta processen.

struct timeval har följande fält:

```
struct timeval{
        int tv_sec;  //sekunder
        int tv_usec; // mikrosekunder
};
```

Det är nu bara att sätta *tv_sec* till det antal sekunder som programmet skall vänta, och *tv_usec* till antal mikrosekunder, inte millisekunder. Det går 1’000 mikrosekunder på en millisekund och 1’000 millisekunder på en sekund. Så det går 1’000’000 mikrosekunder på en sekund. Men varför heter det ”usec”? ”u”:et skall likna den grekiska bokstaven μ (Mu) som står för ”mikro”. Och när funktionen returnerar så *kanske* timout uppdateras för att visa den återstående tiden. Det beror på vilken Unixsmak du använder.

Tjoho! Vi har en räknare som använder mikrosekunder! Nåja, räkna inte med det. Unix standard tidsindelning är omkring på 100 millisekunder, så du måste troligen vänta den tiden även om du ställer in stuct timeval på mindre.

Andra saker som kanske är intressanta: Om du sätter fälten i struct timeval till 0, så kommer select() att avbryta omedelbart och fråga ut alla fildeskriptorer i dina sets. Om du sätter parametern timeout till NULL så kommer den aldrig att bryta, utan kommer att vänta tills den första fildeskriptorn är klar. Slutligen; Om du inte bryr dig om att vänta på ett särskilt set, så kan du sätta det till NULL i anropet till select().

Följande kodavsnitt 15 så väntar programmet 2,5 sekunder på att något skall hända på standardinputen:

```
/*
** select.c -- Ett select() exempel
*/

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

#define STDIN 0  // Fildeskriptor för standardinput

int main(void)
{
    struct timeval tv;
    fd_set readfds;

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

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

    // Bryr mig inte om writefds eller exceptfds:
    select(STDIN+1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds))
        printf("A key was pressed!\n");
    else
        printf("Timed out.\n");

    return 0;
}
```

Om du befinner dig på en radbuffrad terminal så är det troligtvis RETURN du skall trycka på eller så kommer det att bli timeout i alla fall.

Nu tänker några av er att detta kanske är ett bra sätt att vänta på data med datagramsocket – och ni har rätt: de *kanske* är bra. Vissa Unixversioner kan använda select() på det här sättet, andra kan inte. Du borde läsa dina lokala man-sidor och se vad de har att säga på den här punkten, om du tänkte dig att försöka dig på det.

Vissa Unixversioner uppdaterar tiden i struct timeval så att de visar den kvarvarande tiden innan ett timeout. Men andra gör inte detta. Du bör inte räkna med att det är så om du vill vara portabel. (Använd gettimeofday() om du vill se hur mycket tid som är kvar. Det är inte speciellt bra, men så här är det!).

Vad händer om en socket i lässetet avslutar anslutningen? Om så är fallet, så returnerar select() det socketdeskriptorsetet som en ”klar att läsa”. När du tar emot data kommer recv() att returnera 0. Det är så du vet att klienten har stängt anslutningen.

En annan intressant notering angående select(): Om du har en socket i listening(), så kan du kontrollera om det finns en ny anslutning genom att lägga till den socketens fildeskriptor i *readfds* setet.

Och det, min vän, var en snabb genomgång av den allsmäktiga select()funktionen.

Men här kommer, på allmänhetens begäran, ett mer djupgående exempel. Tyvärr så skiljer sig detta exempel drastiskt i jämförelse med ovanstående exempel. Men läs igenom det noga och se till att läsa förklaringen som kommer efter.

Detta program[16] fungerar som ett enkelt multi-user chattprogram. Starta programmet i ett fönster och sedan **telnet**a till det (”**telnet hostname 9034**”) från ett flertal andra fönster. När du skriver in något i en **telnet** session, så bör det komma upp i alla de andra.

```
/*
** selectserver.c – en häftig multiuser-chatserver
*/

#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   // porten vi lyssnar på

int main(void)
{
    fd_set master;   // huvudlistan för fildeskriptorerna
    fd_set read_fds; //temporär lista för fildeskriptorer, används med select()
    struct sockaddr_in myaddr;     // serverns adress
    struct sockaddr_in remoteaddr; // klientens adress
    int fdmax;        // max antal fildeskriptorer
    int listener;     // lyssnande socketdeskriptor
    int newfd;        // Nyss accept()erad socketdeskriptor
    char buf[256];    // buffert för klientdata
    int nbytes;
    int yes=1;        // för setsockopt() SO_REUSEADDR, nedan
    int addrlen;
    int i, j;

    FD_ZERO(&master);    // rensa huvud och temporära listan
    FD_ZERO(&read_fds);

    // hämta lyssnaren
    if ((listener = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    // dumpa det dåliga  "address already in use" felmedelandet
    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);
}

// Lägg till lyssnaren till huvudlistan
FD_SET(listener, &master);

// hållreda på den största fildeskriptorn
fdmax = listener; // än så länge finns det bara en

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

    // Gå igenom befintliga anslutningar efter data att läsa
    for(i = 0; i <= fdmax; i++) {
        if (FD_ISSET(i, &read_fds)) { // vi har en
            if (i == listener) {
                // hantera nya anslutningar
                addrlen = sizeof(remoteaddr);
                if ((newfd = accept(listener, (struct sockaddr *)&remoteaddr,
                                    &addrlen)) == -1) {
                    perror("accept");
                } else {
                    FD_SET(newfd, &master); // lägg till newfd till huvudlistan
                    if (newfd > fdmax) {    // Håll reda på maxantal
                        fdmax = newfd;
                    }
                    printf("selectserver: new connection from %s on "
                        "socket %d\n", inet_ntoa(remoteaddr.sin_addr), newfd);
                }
            } else {
                // Hantera data från klient
                if ((nbytes = recv(i, buf, sizeof(buf), 0)) <= 0) {
                    // Fel har uppstått eller klienten har stängt anslutningen
                    if (nbytes == 0) {
                        // Anslutningen stängd
                        printf("selectserver: socket %d hung up\n", i);
                    } else {
                        perror("recv");
```

```
                    }
                    close(i); // Hejdå
                    FD_CLR(i, &master); // Ta bort från masterlistan
                } else {
                    // Vi har fått data från klienten
                    for(j = 0; j <= fdmax; j++) {
                        // Skicka till alla
                        if (FD_ISSET(j, &master)) {
                            // Utom till lystanaren och mig själv
                            if (j != listener && j != i) {
                                if (send(j, buf, nbytes, 0) == -1) {
                                    perror("send");
                                }
                            }
                        }
                    }
                }
            } // Det här är fult!
        }
    }

    return 0;
}
```

Notera att jag har två fildeskriptorlistan i koden: *master* och *read_fds*. Den första, *master*, innehåller alla socketdeskriptorer som för närvarande är anslutna, samt den socketdeskriptorn som lyssnar efter nya anslutningar.

Anledningen till att jag har *master*listan är att select() *ändrar* listan du skickar med för att visa vilka sockets som är redo att läsas ifrån. För att jag ska hålla reda på anslutningarna, även efter jag anropat select(), så måste jag spara undan dem någonstans. I sista sekunden kopierar jag *master* till *reads_fds*, för att sedan anropa select().

Men betyder inte det att varje gång jag får en ny anslutning, så måste jag lägga till denna till *master*? Jo! Och varje gång som en anslutning stängs så måste jag ta bort den ur *master*listan? Ja, så är det!

Notera också att jag får se efter när lyssnaren är redo att läsa. När det är så, betyder det att jag har en ny anslutning som väntar. Jag accept()erar denna och lägger till den till *master*. Samma gäller när klientanslutning är redo att läsa och recv() returnerar 0, så vet jag att klienten har stängt anslutningen och att jag måste ta bort den ur *master*setet.

Om en klients recv() returnerar ett icke-noll, så vet jag att data har tagits emot. Så jag tar dessa data och genom *master*listan skickar jag datan till resten av de anslutande klienterna.

Och detta, min vän, är en allt annat än enkel översikt över select()funktionen.

6.3 Hantera ofullständiga send()s

Kommer du ihåg att i kapitlet om send() så sa jag att det var inte säkert att send() skickade iväg alla bytes som du sa åt den att skicka? Om du ville skicka 512 bytes, men den returnerade bara 412. Vad hände med de andra 100 byten?

Tja, de ligger fortfarande i din lilla buffer i väntan på att bli skickade. På grund av omständigheter, som du inte rår över, beslutade kärnan att inte skicka alla data på en gång, och nu, min vän, är det upp till dig att se till att få iväg dina data.

Du kan skriva en funktionen som den här för att göra det:

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

int sendall(int s, char *buf, int *len)
{
    int total = 0;        // hur många bytes vi ha skickat
    int bytesleft = *len; // Hur många bytes vi har kvar
    int n;

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

    *len = total; // Returnerar det antal som faktiskt har skickats.

    return n==-1?-1:0; // returnerar -1 vid fel och 0 om det lyckades
}
```

I detta exemplet så är *s* socketen som du vill skicka data till och *buf* är bufferten som innehåller datan. *len* är en pekare till en int som innehåller antalet bytes i bufferten.

Funktionen returnerar –1 vid fel (och *errno* blir satt vid anropet till send()). Alltså är det faktiska antalet bytes som skickades returnerade i *len*. Det skall vara det exakta antalet bytes som du sa till skulle sändas, såvida det inte blev något fel.

sendall() kommer göra sitt bästa, frusta och pusta, för att skicka ut datan, men om det blir något fel kommer den tillbaka till dig likväl.

För att vara komplett, så kommer här ett exempelanrop till funktionen:

```
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);
}
```

Vad händer på mottagarens sida när ett paket kommer? Om paket är olika långa, hur vet då mottagaren när ett paket slutar och den andra börjar? Ja, verkligheten kan ibland vara riktig svår. Du behöver troligtvis inkapsla (kommer du ihåg det från kapitlet om datainkapsling i början av dokumentet?). Läs vidare för detaljer.

6.4 Datainkapsling…

Vad menas det med att inkapsla data egentligen? För att förklara det enklast möjligt, betyder det att du stoppar på en header med information angående identitet eller längd, eller kanske båda delarna.

Men hur skulle denna header se ut? Tja, vilken binär data som helst som du tycker behövs för att färdigställa ditt projekt.

Hops! Det var inte någon ingående förklaring, precis.

Ok, till exempel, säg att du har ett chattprogram för flera simultana användare som använder SOCK_STREAMs. När användarens skriver in (”säger”) något behövs det sändas information om två saker till servern: Vad som sades och vem som sa det.

Än så länge hänger jag med! ”Var finns problemet?”, frågar du.

Problemet är att meddelandet kan variera i längd. En person som heter ”tom” kanske säger ”Hi” och en annan som heter ”Benjamin” kanske säger ”Hey guys what is up?”.

Så du skickar allt till klienterna allt eftersom det kommer in. Din utgående ström ser då ut som följande:

```
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 ?
```

Och så vidare. Hur vet klienten när ett meddelande börjar och när det slutar? Du kan, om du vill, göra alla meddelanden till samma längd och sedan använda funktionen sendall() som vi visade ovan. Men det är att slösa bandbredd! Vi vill inte skicka 1024 bytes bara för att ”tom” skall kunna säga ”Hi”.

Så vi *inkapslar* datan i små header och i paketstrukturer. Både klienten och servern vet hur man skall packa och packa upp (ibland kallas detta ”marshal” och ”unmarshal”) datan. Bli inte rädd nu men vi har börjat att definiera ett protokoll som beskriver hur servern och klienten kommunicerar!

I detta fall antar vi att användarnamnet är fixerat till åtta tecken med ett avslutande ’\0’. Och sedan antar vi att meddelandet är av varierad längd upp till 128 tecken. Vi kanske skall ta ett exempelpaket som vi kan använda oss av:

1. len (1 byte, unsigned) – Den totala längden på paketet, inräknat användarnamn och meddelandet.
2. namn (8 bytes) – Användarnamnet, fyllda med NULL om nödvändigt.
3. chatdata (n-bytes) – Datan själv, inte mer än 128 bytes. Länged på paketet kan räknas ut genom längden på datan plus åtta (längden på namnet ovan).

Varför valde jag åtta och 128 bytes som den maximala längden på fälten? Jag tog dem ur tomma luften och hoppades på att de skulle vara tillräckligt långa. Kanske är åtta bitar lite för restriktivt för dina behov. Du kan utan vidare ha ett namnfält på 30 bytes eller vad du vill. Det är upp till dig.

Enligt ovanstående paketdefinition så skall det första paketet innehålla följande information (i Hex och ASCII):

```
    0A     74 6F 6D 00 00 00 00 00      48 69
(length) T  o  m  (padding)             H  i
```

And det andra:

```
   14     42 65 6E 6A 61 6D 69 6E      48 65 79 20 67 75 79 73 20 77 ...
(length) B  e  n  j  a  m  i  n        H  e  y     g  u  y  s     w ...
```

(Självklart är längden lagrad I Network Byte Order. I detta fall finns det bara en byte så det spelar ingen roll, men generellt så vill du att alla dina binära integers skall vara lagrade i Network Byte Order i dina paket.)

Om du skickar dessa data, så bör du ta det säkra framför det osäkra och använda en funktion som liknar sendall(), som nämns ovan, så att du vet att all data blir sänt. Detta även om det tar flera anrop till send() för att få iväg allt.

Likaså blir det lite extra arbete när du tar emot data. För att ta det säkra före det osäkra, även här, så bör du ta förgivet att det är ett ofullständigt paket som du tar emot (som att få ”00 14 32 65 6E” från Benjamin, ovan, och dessutom är det allt vi får). Vi behöver anropa recv() om och om igen tills vi har fått hela paketet.

Men hur? Vi vet antalet bytes som vi behöver ta emot för att paketet skall vara komplett, då det skickas i början av paketet. Vi vet också att det maximala antalet bytes som vi kan ta emot är 1+8+128, eller 137 bytes (därför att det så vi har definierat paketet…)

Vad du kan göra är att deklarera en vektor tillräckligt stor nog för två paket. Denna är din arbetsvektor där du kan rekonstruera paketen som de kommer.

Varje gång du tar emot data, matar du in den i arbetsbufferten och kontrollerar om paketet är komplett. Alltså om antalet bytes i bufferten är samma antal som eller större än den längd som är specificerad i headern (+1 då headern inte räknar med det byte som definierar längden själv). Om antalet bytes i bufferten är mindre en ett är inte paketet komplett (naturligtvis). Du behöver göra ett specialfall för detta, då den första byten är skräp och du kan inte lita på att det är en korrekt paketlängd.

När paketet sedan är klart, kan du göra precis vad du vill med det. Använda det eller ta bort det ur bufferten.

Pust! Snurrar det runt i huvudet på dig än? Nåja, här är den andra delen av denna två-delade operation: Du kan ha läst förbi slutet av ett paket in i nästa med ett enda anrop till recv(). Du har alltså en arbetsbuffer med ett komplett paket och en okomplett del av nästa paket. Satan också! (men det är på grund av detta som du gjorde din buffer tillräkligt stor för att innehålla två paket).

Då du vet längden på det första paketet ifrån headern och eftersom du har hållit koll på hur många bytes det finns i arbetsbufferten, så kan du ta bort och beräkna hur många av byten som tillhör det andra (okompletta) bufferten. När du har tagit hand om det första paketet, så kan du ta bort det ur bufferten och flytta det andra, ofullständiga paketet först i paketet, så är det redo att fortsätta läsa med hjälp av recv().

(Några av er läsare noterade säkert att lösningen med att flytta det ofullständiga paketet är tidsödslande, och att programmet kan kodas om att använda en cirkulär vektor. Tyvärr för resten av er så kommer inte cirkulära vektorer tas upp i detta dokument. Om du är nyfiken så skaffa en bok om datastrukturer och läs mer där.).

Ja sa aldrig att det skulle vara enkelt. Ok, jag sa faktiskt att det skulle vara enkelt. Och det är det; Du behöver bara träna lite och snart kommer det att vara helt naturligt. Jag svär vid Excalibur!

7. Fler referenser

När du kommit så här långt så bara måste du ha mer!? Var kan Du lära dig mer om de här sakerna?

7.1 man Sidorna

Leta bland följande mansidor, till att börja med:

- htonl()[17]
- htons()[18]
- ntohl()[19]
- ntohs()[20]
- inet_aton()[21]
- inet_addr()[22]
- inet_ntoa()[23]
- socket()[24]
- socket options[25]
- bind()[26]
- connect()[27]
- listen()[28]
- accept()[29]
- send()[30]
- recv()[31]
- sendto()[32]
- recvfrom()[33]
- close()[34]
- shutdown()[35]
- getpeername()[36]

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

7.2 Böcker

För traditionell-skol-att-hålla-i-handen-böcker, kolla upp några av dessa excellenta guiderna. Notera den framstående Amazon.com logon. Vad denna skamlösa kommersialism betyder är att jag får något tillbaka (presentkort hos Amazon.com) genom att sälja dessa böcker genom denna guide. Så om du ändå skall beställa någon av dessa böcker kan du väl passa på att skicka mig ett tack genom att börja ditt köp genom denna länk.

Dessutom, mer böcker för mig kommer utan tvekan leda till fler guider till dig. ;-)

Unix Network Programming, volumes 1-2 by W. Richard Stevens. Published by Prentice Hall. ISBNs for volumes 1-2: 013490012X[46], 0130810819[47].

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[48], 0139738436[49], 0138487146[50].

TCP/IP Illustrated, volumes 1-3 by W. Richard Stevens and Gary R. Wright. Published by Addison Wesley. ISBNs for volumes 1, 2, and 3: 0201633469[51], 020163354X[52], 0201634953[53].

TCP/IP Network Administration by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 1565923227[54].

Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by Addison Wesley. ISBN 0201563177[55].

Using C on the UNIX System by David A. Curry. Published by O'Reilly & Associates, Inc. ISBN 0937175234. *utgången ur förlaget.*

7.3 Webbreferenser

På webben:

BSD Sockets: A Quick And Dirty Primer[56] (Har annan bra information om Unix systemprogrammering)

The Unix Socket FAQ[57]

Client-Server Computing[58]

Intro to TCP/IP (gopher)[59]

Internet Protocol Frequently Asked Questions[60]

The Winsock FAQ[61]

7.4 RFCs

RFCs[62]– Den riktiga skiten:

RFC-768[63] – User Datagram Protocol (UDP)

RFC-791[64]– Internet Protocol (IP)

RFC-793[65]– Transmission Control Protocol (TCP)

RFC-854[66]– Telnet Protocol

RFC-951[67]– Bootstrap Protocol (BOOTP)

RFC-1350[68]– Trivial File Transfer Protocol (TFTP)

8. Vanliga frågor

F: Varifrån kan jag skaffa headerfilerna?

S: Om du inte har dem på ditt system redan kommer du troligtvis inte att behöva dem. Kolla i manualen för din specifika plattform. Om du utvecklar i Windows så behöver du bara `#include <winsock.h>`**.**

F: Vad gör jag om `bind()` säger att ”Address already in use”?

S: Du måste använda `setsockopt()` med alternativet `SO_REUSEADDR` på den lyssnande socketen. Det finns exempel på detta i kapitlen om `bind()` och `select()`.

F: Hur får jag en lista på öppna sockets på ett system?

S: Använd netstat. Läs man-sidan för mer detaljer, men du kan få ganska bra information bara genom att skriva :

```
$ netstat
```

Det enda problemet är att ta reda på vilken socket som tillhör vilket program. :-)

F: Hur kan jag se routing-tabellen?

S: Kör kommandot route (i /sbin på de flesta linuxar) eller kommandot netstat –r.

F: Hur kan jag köra klient- och serverprogrammen om jag bara har en dator? Behöver jag inte ett nätverk för att kunna skriva nätverksprogram?

S: Som tur är, för dig, så finns det en virtuell loopbacknätverkenhet som sitter i kärnan och låtsas att det är ett nätverkskort. (Detta är gränssnittet som listas som ”lo” i routingtabellen).

Låtsas att du är påloggad på en maskin med namnet ”goat”. Kör servern i ett fönster och klienten i ett annat. Eller kör servern i bakgrunden (”**server &**”) och kör klienten i samma fönster. Resultatet av loopbackenheten är att du kan antingen köra **klient goat** eller **klient localhost** (då localhost sannolikt definierad i din /etc/hosts-fil) och så har du din klient som talar med din server utan att du har ett nätverk!

Kort och gott, inga ändringar behövs göras för att du skall få koden att fungera på en ensam, icke-nätverksansluten maskin! Huzzah!

F: Hur vet jag om den andra sidan har stängt sin anslutning?

S: Det vet du genom att recv() returnerar 0.

F: Hur implementerar jag ett ”ping”? Vad är ICMP? Var kan jag hitta mer angående rawsockets och SOCK_RAW?

S: Alla dina rawsocketsfrågor är besvarade i W. Richard Stevens’ Unix Network Programming böcker. Se kapitlet om böcker i den här guiden.

F: Hur utvecklar jag för Windows?

S: För det första, radera Windows och installera Linux eller BSD. };-). Nej seriöst, se kapitlet om Windows i introduktionen.

F: Hur utvecklar jag i Solaris/SunOS? Jag får hela tiden länkfel när jag försöker kompilera!

S: Länkfelen beror på att Suns maskiner inte automatiskt kompilerar in socketbiblioteken. Se kapitlet om utveckling i Solaris/SunOS i introduktionen för att se hur du gör det.

F: Varför blir det fel i select() på en signal?

S: Signaler har en tendens att blocka när systemanrop skall returnera –1 med *errno* satt till EINTR. När du sätter upp en signalhanterare med sigaction() så kan du sätta flaggan till SA_RESTART, vilket gör att systemanrop blir körda igen om det blir avbrutet.

Naturligvis fungerar detta inte alltid.

Min favorit lösning involverar ett goto-uttryck. Du vet att detta irriterar dina lärare till vansinne, så kör bara!

```
    select_restart:
       if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
          if (errno == EINTR) {
                  // Vissa signaler avbryter oss bara, så vi startar om…
                  goto select_restart;
           }
    // Här hanteras de riktiga felen:
    perror("select");
}
```

Visst, Du *behöver* inte använda goto i det här fallet; Du kan använda andra strukturer för att kontrollera det. Men jag tycker att goto-uttrycket faktiskt är renare.

F: Hur implementerar jag en timeout i ett anrop till recv()?

S: Använd select()! Den tillåter dig att ange timeoutparameter för socketdeskriptorer som du ser om du kan läsa ifrån. Eller så kan du vira in det hela i en enda funktion, så här:

```
#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;

        // implementera fildeskriptorsetet
        FD_ZERO(&fds);
        FD_SET(s, &fds);

        // implementera en struct timeval för timeouten
        tv.tv_sec = timeout;
        tv.tv_usec = 0;

        // Vänta på timeout eller tills data tas emot
        n = select(s+1, &fds, NULL, NULL, &tv);
        if (n == 0) return -2; // timeout!
        if (n == -1) return -1; // error


        // Datan skall vara här, så gör en normal recv()
        return recv(s, buf, len, 0);
        }
```

```
        // Exmepelanrop till recvtimeout():
        .
        .
        n = recvtimeout(s, buf, sizeof(buf), 10); // 10 sekunders timeout
        if (n == -1) {
                // Fel uppstod
                perror("recvtimeout");
        }
        else if (n == -2) {
        // det blev timeout
        } else {
        // Har data i bufferten
        }
.
.
```

Notera att recvtimeout() returnerar –2 vid timeout. Varför inte returnera 0? Tja, om du har glömt det så returnerar recv() 0 om den andra sidan har stängt sin anslutning. Så det returnvärdet talar för sig själv, och –1 betyder ”fel”, så jag valde –2 för till att representera timeout.

F: Hur krypterar jag eller komprimerar data innan jag skickar det genom en socket?

S. Ett enkelt sätt att kryptera är att använda SSL (secure sockets layer), med det går utanför den här guiden.

Men antag att du vill implementera en av dina egna kompression- eller krypteringssystem. Det är bara en fråga om att låta din data gå igenom vissa steg i båda ändarna. Varje steg ändrar datan på något sätt.

1. Servern läser data från en fil (eller nått)
2. Servern krypterar datan. (detta är din del)
3. Servern skickar iväg datan.

Och sedan precis det mottsatta.

4. Klienten tar emot data
5. Klienten dekrypterar datan (det får du göra)
6. Klienten skriver datan till en fil (eller nått)

Du kan också komprimera istället för att kryptera i listan ovan. Eller du kan göra båda! Kom bara ihåg att komprimera innan du krypterar. :)

Bara så länge du gör serverns ändringar ogjorda i klienten så kommer datan att vara som den var, oberoende hur många steg som du har förändrat den i.

Så allt du behöver göra i min kod är att hitta det ställe mellan datan är läst och datan skickas (när send() används) ut i nätverket, och stoppa in lite kod där för krypteringen.

F: Vad är det här ”PF_INET” som jag ideligen ser? Är det besläktat med AF_INET?

S: Ja, det är det. Se kapitlet om socket() för detaljer.

9. Dementi och rop på hjälp

Jaha, det var allt det. Förhoppningsvis så är i alla fall lite av den informationen i dokumentet är korrekt och att det inte är allt för uppenbara fel. Tja, Det är det väl i och för sig alltid.

Låt det vara en varning till dig! Jag är ledsen om något fel har resulterat i gråt och tandagnisslande, men du kan inte hålla mig ansvarig. Du vet, jag står inte bakom ett enda ord av det som står i dokumentet, ut juridisk synvinkel. Hela dokumentet kunde lika gärna vara helt och hållet fel!

Men det är det troligtvis inte. När allt kommer omkring så har jag lagt många timmar på de här grejerna och implementerat TCP/IP-nätverksverktyg på jobbet, skrivit motorer till fleranvändarspel, och så vidare. Men jag är inte någon socketgud, bara en vanlig kille.

Om du har någon konstruktiv (eller destruktiv) kritik angående det här dokumentet, så skicka ett mail till <beej@piratehaven.org> så skall jag försöka fixa vad som är fel.

Om du undrar varför jag gjorde detta, så gjorde jag det för pengarna. Ha! Nej, faktum är att jag gjorde det för att många personer frågar mig socket-relaterade frågor hela tiden och jag har då svarat dem att jag har funderat på att skriva en socket-sida. ”Häftigt!”, har de då svarat. Dessutom tycker jag att denna hårt förtjänade kunskapen skulle gå förlorad om jag inte fick dela den med andra. Webben råkade bara vara ett perfekt redskap. Jag uppmuntrar andra att dela med sig med liknande kunskaper närs om helst är möjligt.

Nog om det här – Tillbaka till kodningen! ;-)

Noter

1. http://www.ecst.csuchico.edu/~beej/guide/net/
2. http://www.cyberport.com/~tangent/programming/winsock/
3. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/send.2.inc
4. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recv.2.inc
5. http://www.rfc-editor.org/rfc/rfc793.txt
6. http://www.rfc-editor.org/rfc/rfc791.txt
7. http://www.rfc-editor.org/rfc/rfc768.txt
8. http://www.rfc-editor.org/rfc/rfc791.txt
9. http://www.rfc-editor.org/rfc/rfc1413.txt
10. http://www.ecst.csuchico.edu/~beej/guide/net/examples/getip.c

11. http://www.ecst.csuchico.edu/~beej/guide/net/examples/server.c
12. http://www.ecst.csuchico.edu/~beej/guide/net/examples/client.c
13. http://www.ecst.csuchico.edu/~beej/guide/net/examples/listener.c
14. http://www.ecst.csuchico.edu/~beej/guide/net/examples/talker.c
15. http://www.ecst.csuchico.edu/~beej/guide/net/examples/select.c
16. http://www.ecst.csuchico.edu/~beej/guide/net/examples/selectserver.c
17. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/htonl.3.inc
18. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/htons.3.inc
19. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/ntohl.3.inc
20. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/ntohs.3.inc
21. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_aton.3.inc
22. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_addr.3.inc
23. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_ntoa.3.inc
24. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/socket.2.inc
25. http://linux.com.hk/man/showman.cgi?manpath=/man/man7/socket.7.inc
26. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/bind.2.inc
27. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/connect.2.inc
28. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/listen.2.inc
29. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/accept.2.inc
30. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/send.2.inc
31. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recv.2.inc
32. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/sendto.2.inc
33. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recvfrom.2.inc
34. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/close.2.inc
35. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/shutdown.2.inc
36. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getpeername.2.inc
37. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getsockname.2.inc
38. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyname.3.inc
39. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyaddr.3.inc
40. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/getprotobyname.3.inc
41. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/fcntl.2.inc
42. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/select.2.inc
43. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/perror.3.inc
44. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/gettimeofday.2.inc

45. http://www.amazon.com/exec/obidos/redirect-home/beejsguides-20
46. http://www.amazon.com/exec/obidos/ASIN/013490012X/beejsguides-20
47. http://www.amazon.com/exec/obidos/ASIN/0130810819/beejsguides-20
48. http://www.amazon.com/exec/obidos/ASIN/0130183806/beejsguides-20
49. http://www.amazon.com/exec/obidos/ASIN/0139738436/beejsguides-20
50. http://www.amazon.com/exec/obidos/ASIN/0138487146/beejsguides-20
51. http://www.amazon.com/exec/obidos/ASIN/0201633469/beejsguides-20
52. http://www.amazon.com/exec/obidos/ASIN/020163354X/beejsguides-20
53. http://www.amazon.com/exec/obidos/ASIN/0201634953/beejsguides-20
54. http://www.amazon.com/exec/obidos/ASIN/1565923227/beejsguides-20
55. http://www.amazon.com/exec/obidos/ASIN/0201563177/beejsguides-20
56. http://www.cs.umn.edu/~bentlema/unix/
57. http://www.ibrado.com/sock-faq/
58. http://pandonia.canberra.edu.au/ClientServer/
59. gopher://gopher-chem.ucdavis.edu/11/Index/Internet_aw/Intro_the_Internet/intro.to.ip/
60. http://www-iso8859-5.stack.net/pages/faqs/tcpip/tcpipfaq.html
61. http://www.cyberport.com/~tangent/programming/winsock/
62. http://www.rfc-editor.org/
63. http://www.rfc-editor.org/rfc/rfc768.txt
64. http://www.rfc-editor.org/rfc/rfc791.txt
65. http://www.rfc-editor.org/rfc/rfc793.txt
66. http://www.rfc-editor.org/rfc/rfc854.txt
67. http://www.rfc-editor.org/rfc/rfc951.txt
68. http://www.rfc-editor.org/rfc/rfc1350.txt