

Beej's Handleiding voor Unix Netwerk Programmeren

Het Gebruik van Internet Sockets

Brian "Beej" Hall

beej@piratehaven.org

Copyright © 1995-2001 door Brian "Beej" Hall

Wijzigingen

Herziening Versie 1.0.0 Augustus 1995 Herzien door: beej

Eerste versie.

Herziening Versie 1.5.5 13 Januari 1999 Herzien door: beej

Nieuwste HTML versie.

Herziening Versie 2.0.0 6 Maart 2001 Herzien door: beej

Geconverteerd naar DocBook XML, correcties, toevoegingen.

Herziening Versie 2.3.1 8 Oktober 2001 Herzien door: beej

Typefouten gecorrigeerd, syntax fout in client.c, een en ander aan V&A onderdeel toegevoegd.

Inhoudsopgave

1. Introductie	3
1.1. Publiek	3
1.2. Platform en Compiler	3
1.3. Officiële Homepage	3
1.4. Opmerking voor Solaris/SunOS Programmeurs	3
1.5. Opmerking voor Windows Programmeurs	4
1.6. Email Beleid	5
1.7. Spiegelen (mirroring)	5
1.8. Opmerking voor Vertalers	6
1.9. Opmerking bij de Nederlandse Vertaling	6
1.10. Copyright en Distributie	6
2. Wat is een socket?	6
2.1. Twee Soorten Internet Sockets	7
2.2. Low level Gedoe en Netwerk Theorie	8

3. structs en Data Handling	10
3.1. De Inboorlingen Converteren!.....	12
3.2. IP Adressen en Hoe Ermee Om te Gaan	13
4. Systeemaanroepen of Barsten	14
4.1. <code>socket()</code> –Grijp Die File Descriptor!	15
4.2. <code>bind()</code> –Op Welke Poort Zit Ik?.....	15
4.3. <code>connect()</code> –Hé jij daar!	18
4.4. <code>listen()</code> –Wil iemand me alsjeblieft bellen?.....	19
4.5. <code>accept()</code> –"Bedankt voor het bellen met poort 3490."	20
4.6. <code>send()</code> en <code>recv()</code> –Praat tegen me, schatje!	21
4.7. <code>sendto()</code> and <code>recvfrom()</code> –Zeg wat tegen me, op z'n DGRAM's	22
4.8. <code>close()</code> en <code>shutdown()</code> –Oprotten!.....	23
4.9. <code>getpeername()</code> –Wie ben jij?	24
4.10. <code>gethostname()</code> –Wie ben ik?.....	25
4.11. DNS–Jij zegt "www.overheid.nl", ik zeg "193.78.30.5"	25
5. Client-Server Achtergrond	27
5.1. Een eenvoudige Stream Server	28
5.2. Een Eenvoudige Stream Client.....	31
5.3. Datagram Sockets.....	32
6. Enigszins Geavanceerde Technieken	36
6.1. Blocking	36
6.2. <code>select()</code> –Synchrone I/O Multiplexing	37
6.3. Omgaan met partiële <code>send()</code> s.....	43
6.4. Data Inkapsulatie.....	44
7. Meer Referenties	46
7.1. man Pagina's.....	47
7.2. Boeken.....	48
7.3. Web Referenties	48
7.4. RFCs.....	49
8. Gebruikelijke Vragen	49
9. Disclaimer en een Hulpoproep	56

1. Introductie

Hee! De moed verloren met socket programmeren? Is deze zooi net iets te ingewikkeld om via de **man** pages uit te vogelen? Wil je gaaf Internet programmeren, maar heb je geen tijd om door een stapel `structs` heen te waden om uit te zoeken of je nou moet `bind()` en voor je `connect()`, enz, enz.?

Nou, raad eens! Dat akelige karwei heb ik al gedaan, en ik sta te springen om die kennis met iedereen te delen! Je bent op de goeie plek. Dit dokument zou de gemiddelde bekwame C programmeur net het nodige duwtje in de rug moeten geven die hij/zij nodig heeft om een beetje vat te krijgen op dit hele netwerkgebeuren.

1.1. Publiek

Dit dokument is geschreven als een leerboek (tutorial), niet als een referentiewerk. Het werkt waarschijnlijk het beste als het gelezen wordt door personen die net beginnen met socket programmeren en op zoek zijn naar een beetje houvast. Hoe dan ook, dit is zeker niet de *complete* handleiding over socket programmeren.

Hopelijk, echter, is het net genoeg om al die man pages een beetje te kunnen ontcijferen... :-)

1.2. Platform en Compiler

De code in dit dokument werd gecompileerd op een Linux PC met GNU's **gcc** compiler. Het zou echter moeten compileren op zo ongeveer elk ander platform dat **gcc** gebruikt. Dit is uiteraard niet van toepassing als je in/voor Windows programmeert—zie dan het onderdeel over Windows programmeren, verderop.

1.3. Officiële Homepage

De officiële locatie van dit dokument is op de Universiteit van de staat Californië, Chico, op <http://www.ecst.csuchico.edu/~beej/guide/net/>¹.

1.4. Opmerking voor Solaris/SunOS Programmeurs

Wanneer je aan het compileren bent onder Solaris of SunOS, moet je een aantal extra commandoregel parameters meegeven om de juiste bibliotheken (libraries) te linken. Om dit voor elkaar te krijgen voeg je simpelweg "`-lnsl -lsocket -lresolv`" toe aan het einde van het compileercommando, zo dus:

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

Als je nog steeds fouten krijgt zou je nog kunnen proberen om "-lXnet" aan het einde van die commandoregel toe te voegen. Ik weet niet precies wat dat doet, maar sommige mensen schijnen het nodig te hebben.

Een andere plek waar je problemen tegen zou kunnen komen is in de aanroep naar `setsockopt()`. Het prototype verschilt van die op m'n Linux doos, dus in plaats van:

```
int yes=1;
```

moet je dit invoeren:

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

Aangezien ik geen Sun doos heb, heb ik van de bovenstaande informatie niets kunnen testen—dit is alleen wat mensen me via email verteld hebben.

1.5. Opmerking voor Windows Programmeurs

Ik heb een bijzondere afkeer voor Windows, en raad je aan om in plaats daarvan Linux, BSD of Unix te proberen. Na dat gezegd te hebben kan ik je vertellen dat je deze zooi ook onder Windows kunt gebruiken.

Ten eerste kun je zo'n beetje alle systeem header files die ik in dit dokument opnoem vergeten. Het enige wat je moet invoegen is:

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

Wacht! Je moet ook een aanroep maken naar `WSAStartup()` voordat je ook maar iets met de sockets bibliotheek doet. De code om dat te doen ziet er ongeveer zo uit:

```
#include <winsock.h>

{
    WSADATA wsaData;    // als dit niet werkt,
//WSADATA wsaData; // probeer dan dit.

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

Ook moet je je compiler vertellen om de Winsock library te linken, meestal `wsock32.lib` genaamd of `winsock32.lib` of iets dergelijks. Onder VC++ kan dit gedaan worden via het `Project` menu, onder

Instellingen... Klik op het `Link` tabblad, en zoek naar de knop "Object/Bibliotheek modules". Voeg "wsock32.lib" toe aan die lijst.

Zo is me ten minste verteld.

Tenslotte dien je `WSACleanup()` aan te roepen wanneer je klaar bent met de sockets library. Zie je online help voor details.

Als je dat eenmaal gedaan hebt zou de rest van de voorbeelden in dit leerboek in het algemeen toepasbaar moeten zijn, met een paar uitzonderingen. Om te beginnen, je kunt niet de `close()` functie gebruiken om een socket te sluiten—daar moet je `closesocket()` voor gebruiken. Verder werkt de functie `select()` alleen met socket descriptors, niet met file descriptors (zoals 0 voor `stdin`).

Er is ook een socket klasse die je kan gebruiken, `CSocket`. Check je compiler's help pagina's voor meer informatie.

Ga voor meer informatie over Winsock naar de Winsock FAQ².

Tenslotte, het is me ter ore gekomen dat Windows geen `fork()` systeemaanroep heeft welke, helaas, in een aantal van m'n voorbeelden gebruikt wordt. Misschien moet je iets van een POSIX library of zo gebruiken om het aan de praat te krijgen, anders kun je misschien `CreateProcess()` gebruiken. `fork()` heeft geen argumenten nodig, terwijl `CreateProcess()` ongeveer 48 miljard argumenten wil. Als je daar niet tegen opgewassen bent, de `CreateThread()` is wat makkelijker te verteren... Helaas valt een discussie over multithreading buiten het bestek van dit document. Ik kan het maar over zoveel dingen hebben, weet je!

1.6. Email Beleid

Over het algemeen ben ik gewoon bereikbaar om via email vragen te beantwoorden, dus voel je vrij om te schrijven, maar ik kan geen antwoord garanderen. Ik leid een vrij druk leven en er zijn tijden wanneer ik gewoon geen antwoord op je vraag kan geven. Wanneer dat het geval is verwijder ik het mailtje meestal gewoon. Da's niet persoonlijk bedoeld; ik heb gewoon niet eeuwig de tijd om je het gedetailleerde antwoord te geven dat je nodig hebt.

Als vuistregel, des te complexer je vraag, des te kleiner de kans dat ik antwoord geef. Als je je vraag wat specifiek kan maken voor je hem verstuurt (zoals platform, compiler, foutmeldingen, en al het andere waarvan je denkt dat ik het probleem er sneller mee zou kunnen oplossen) is de kans dat je antwoord krijgt een stuk groter. Voor meer tips zou je ESR's document, *How To Ask Questions The Smart Way*³ (Hoe stel ik vragen op een slimme manier).

Als je geen antwoord krijgt, pruts er dan nog wat meer mee in de hoop op een oplossing, en als het dan nog steeds niet lukt, mail me dan nog een keer met de nieuwe informatie die je hebt gevonden en hopelijk is het genoeg voor me om je te helpen.

Nou ik je eenmaal heb lastig gevallen over hoe je me moet mailen en hoe niet, wil ik je even laten weten dat ik alle lof die m'n handleiding in de afgelopen jaren heeft gekregen *zeer* waardeer. Het geeft me een flinke oppepper, en het doet me plezier te horen dat het voor het goede wordt gebruikt! :-) Bedankt!

1.7. Spiegelen (mirroring)

Je bent van harte welkom om deze site te spiegelen, hetzij publiek of privé. Als je de site publiek spiegelt en wil dat ik ernaar link vanaf de hoofdpagina, laat het me dan even weten op <beej@piratehaven.org>.

1.8. Opmerking voor Vertalers

Als je de handleiding wil vertalen naar een andere taal, meld dat dan even op <beej@piratehaven.org>, dan maak ik een link naar je vertaling op de hoofdpagina.

Voeg gerust je naam en emailadres toe aan de vertaling.

Sorry, maar door ruimtegebrek kan ik de vertalingen niet zelf hosten.

1.9. Opmerking bij de Nederlandse Vertaling

Deze handleiding is voor het eerst vertaald naar het Nederlands op 5 april 2001, door Kars Meyboom. Mocht je opmerkingen of suggesties hebben of typefouten tegenkomen, mail me dan even op <kars@kde.nl>.

Zowel de Nederlandse vertaling als een mirror van de originele handleiding zijn beschikbaar vanaf <http://analyser.oli.tudelft.nl/beej/>⁴.

1.10. Copyright en Distributie

Beej's Handleiding voor Netwerk Programmeren is Copyright © 1995-2001 Brian "Beej" Hall.

Deze handleiding mag vrij herdrukt worden in elk medium mits de inhoud niet is aangepast, het in zijn geheel wordt aangeboden en deze copyright mededeling intact blijft.

Docenten worden in het bijzonder aangemoedigd om deze handleiding aan hun studenten aan te bevelen of hen kopiëren aan te reiken.

Deze handleiding mag vrij worden vertaald naar elke taal, mits de vertaling accuraat is, en de handleiding in zijn geheel wordt aangeboden. De vertaling mag eveneens de naam en contactinformatie van de vertaler bevatten.

De C broncode die in dit document gepresenteerd wordt is bij deze vrijgegeven in het public domain.

Je kunt contact opnemen via <beej@piratehaven.org> voor meer informatie.

2. Wat is een socket?

Je hoort de hele tijd mensen praten over "sockets", en misschien vraag je je af wat dat nou eigenlijk zijn. Nou, sockets zijn: een manier om met andere programma's te praten d.m.v. standaard Unix file descriptors.

Huh?

Ok—misschien heb je een Unix hacker wel eens horen zeggen, "Tjemig, *alles* in Unix is een bestand!" Waar die persoon het dan over had is het feit dat wanneer Unix programma's ook maar enig soort van I/O doen, ze dat doen door te lezen van en te schrijven naar een file descriptor. Een file descriptor is simpelweg een integer, een geheel getal, dat geassocieerd wordt met een open bestand. Maar (en hier zit 'm de kneep), dat bestand kan een netwerkverbinding, een FIFO, een pipe, een terminal, een echt bestand op een schijf zijn, of zo'n beetje wat dan ook. In Unix *is* alles een bestand! Dus als je over het internet met een ander programma wil communiceren, dan doe je dat door een file descriptor, geloof me nou maar.

"Okee wijsneus, waar krijg ik deze file descriptor voor netwerkcommunicatie dan?" is waarschijnlijk de laatste vraag die je nou zou willen stellen, maar ik beantwoord hem toch maar: je roept de `socket()` systeemroutine aan. Het geeft een socket descriptor als waarde terug, en je praat erdoor door de gespecialiseerde `send()` en `recv()` (**man send**⁵, **man recv**⁶) functies aan te roepen.

"Ja maar, hee!" hoor ik je nou roepen, "Als het een file descriptor is, waarom, in naam van Neptunus, kan ik dan niet gewoon de normale `read()` en `write()` systeemaanroepen gebruiken om door m'n socket te praten?" Het korte antwoord is, "Dat kan!" Het lange antwoord is, "Het kan, maar `send()` en `recv()` bieden je veel meer controle over je data-overdracht.

Okee, en nu? Wat dacht je hiervan: er zijn allerlei soorten sockets. Er zijn DARPA Internet adressen (Internet Sockets), padnamen op een locale machine (Unix Sockets), CCITT X.25 adressen (X.25 Sockets die je veilig kunt vergeten) en waarschijnlijk nog veel meer, afhankelijk van het soort Unix dat je draait. Dit document heeft alleen betrekking op de eerste: Internet Sockets.

2.1. Twee Soorten Internet Sockets

Wat krijgen we nou? Zijn er twee soorten Internet sockets? Ja. Nou ja, eigenlijk niet. Ik lieg. Het zijn er meer, maar ik wil je niet bang maken. Ik ga het maar over twee soorten hebben. Behalve in deze zin, waarin ik je vertel dat "Kale Sockets" (Raw Sockets) ook erg krachtig zijn en dat je er wat over zou moeten lezen.

Okee, Okee... Wat zijn die twee soorten? De ene soort is de "Stream Socket", de andere de "Datagram Socket", naar welke vanaf nu als `"SOCK_STREAM"`, respectievelijk `"SOCK_DGRAM"` gerefereerd zal worden. Datagram sockets worden soms "verbindingloze (connectionless) sockets" genoemd. (Hoewel je ze ook kunt `connect()` en als je dat echt wilt. Zie `connect()`, verderop.)

Stream sockets zorgen voor betrouwbare twee-weg communicatiestromen. Als je twee dingen in de socket stopt in de volgorde "1, 2", dan komen ze aan de andere kant ook aan in de volgorde "1, 2". Ze zullen ook vrij van fouten zijn.

Fouten die je wel tegenkomt zijn hersenspinsels van je eigen verwarde geest, en daar zullen we het hier dus niet over hebben.

Wie of wat gebruikt stream sockets? Nou, misschien heb je wel eens van het **telnet** programma gehoord. Dat gebruikt stream sockets. Alle karakters die je typt moeten in dezelfde volgorde aankomen zoals je ze typt, nietwaar? Verder gebruiken web browsers het HTTP protocol, welke stream sockets gebruikt om pagina's op te halen. Inderdaad, als je telnet naar een web site op poort 80, en "GET /" intypt, rolt de HTML pagina over je scherm!

Hoe realiseren stream sockets zo'n hoog kwaliteitsniveau wat de datatransmissie betreft? Ze gebruiken een protocol genaamd "Het Transmissie Controle Protocol (The Transmission Control Protocol)", ook wel "TCP" genaamd (zie RFC-793⁷ voor extreem gedetailleerde informatie over TCP). TCP zorgt ervoor dat je data sequentiëel (dus in volgorde) en foutvrij aankomt. Misschien heb je "TCP" wel eens eerder gehoord in de term "TCP/IP" waarin "IP" staat voor "Internet Protocol" (zie RFC-791⁸). IP houdt zich voornamelijk bezig met internet routing (het zoeken van een weg van een computer naar een andere over een internet) en is over het algemeen niet verantwoordelijk voor data-integriteit.

Gaaf. En hoe zit het met datagram sockets? Waarom worden ze verbindingloos genoemd? Hoe zit dat nou eigenlijk? Waarom zijn ze onbetrouwbaar? Nou, hier zijn wat feiten: als je een datagram stuurt, zou het aan kunnen komen. Het zou in een andere volgorde aan kunnen komen. Als het aankomt, dan is de data in het pakket vrij van fouten.

Datagram sockets gebruiken ook IP voor hun routing, maar ze gebruiken geen TCP; ze gebruiken het "Gebruiker Datagram Protocol (User Datagram Protocol)", of "UDP" (zie RFC-768⁹.)

Waarom zijn ze verbindingloos? Nou, simpelweg omdat je geen verbinding open hoeft te houden zoals je dat met stream sockets doet. Je bouwt gewoon een pakketje, mept er een IP header op met een bestemmingsadres, en stuurt 't de deur uit. Geen verbinding nodig. Ze worden over het algemeen gebruikt voor per-pakket verzendingen van informatie. Voorbeeldtoepassingen: **ftp**, **bootp**, etc.

"Genoeg!" hoor ik je roepen. "Hoe werken deze programma's nou eigenlijk als datagrammen verloren kunnen gaan?!" Nou, m'n menselijke vriend, elk van die programma's heeft zijn eigen protocol boven UDP. Bijvoorbeeld, het tftp protocol zegt dat voor elk pakket dat verstuurd wordt, de ontvanger een pakketje moet terug sturen dat zegt, "Ik heb 'em!" (een "ACK" pakket). Als de verzender van het originele pakket geen antwoord krijgt in, zeg, vijf seconden, dan stuurt hij het pakket opnieuw tot hij een ACK krijgt. Deze bevestigingsprocedure (ACKnowledgement procedure) is erg belangrijk bij het implementeren van `SOCK_DGRAM` programma's.

2.2. Low level Gedoe en Netwerk Theorie

Aangezien ik het net heb gehad over het stapelen (layering) van protocollen, is het tijd om het te hebben over hoe netwerken echt werken, en wat voorbeelden te laten zien over hoe `SOCK_DGRAM` pakketten worden gebouwd. Praktisch gezien zou je dit hoofdstuk over kunnen slaan, maar het bevat wel goede achtergrondinformatie.

Hey jongens, het is tijd om wat te leren over *Data Inkapsulatie (Data Encapsulation)*! Dit is heel erg belangrijk.

Figuur 1. Data Inkapsulatie.



Zelfs zo belangrijk dat je het misschien wel zou leren als je de netwerkcursus doet, hier in Chico State :-). Waar het om draait is het volgende: een pakketje wordt geboren, wordt in een kop (header) verpakt ("ingekapseld") (en zelden in een staart (footer)) door het eerste protocol (zeg, het TFTP protocol). Daarna wordt het hele zaakje (inclusief de TFTP kop) opnieuw ingepakt door het volgende protocol (zeg, UDP), daarna nog een keer door het volgende protocol (IP), dan nog een keer door het laatste protocol van de hardware (fysieke) laag (zeg, ethernet).

Wanneer een andere computer het pakket ontvangt, haalt de hardware de ethernet kop eraf, de kernel verwijdert de IP en UDP koppen, het TFTP verwijdert de TFTP kop, en verkrijgt zo de data.

Nou kan ik het eindelijk hebben over het notoire *Gelaagde Netwerk Model (Layered Network Model)*. Dit netwerk model beschrijft een systeem van netwerkfunctionaliteit dat vele voordelen heeft over andere modellen. Bijvoorbeeld, je kunt socket programma's schrijven die exact hetzelfde zijn zonder je zorgen te maken over hoe de data fysiek wordt verzonden (seriële kabel, ethernet, AUI, maakt niet uit) omdat programma's op lagere niveaus dat voor je doen. De feitelijke netwerk hardware en topologie is transparant voor de socket programmeur.

Zonder verder gedraal presenteer ik de lagen van het hele model. Onthou dit voor je netwerkcursus examens:

- Applicatie
- Presentatie
- Sessie
- Transport
- Netwerk
- Data Link
- Fysiek

De Fysieke Laag is de hardware (serieel, ethernet, enz.). De Applicatie Laag is zo'n beetje het verst van de fysieke laag als je je maar kan voorstellen—dat is waar gebruikers de interactie met het netwerk aangaan.

Nou, dit model is zo algemeen dat je het waarschijnlijk zou kunnen gebruiken als een autoreparatiehandleiding als je het echt zou willen. Een gelaagd model dat meer overeenkomst vertoont met Unix zou kunnen zijn:

- Applicatielaag (*telnet, ftp, etc.*)
- Host-naar-Host Transportlaag (*TCP, UDP*)
- Internetlaag (*IP en routing*)
- Netwerk Toegangslaag (*Ethernet, ATM, of wat dan ook*)

Vanaf dit moment zie je waarschijnlijk wel in hoe deze lagen corresponderen met de inkapsulatie van de originele data.

Zie je hoeveel werk er nodig is om een simpel pakketje te bouwen? Jemig! En je moet de koppen van de pakketten zelf intypen met "**cat**"! Geintje. Het enige wat je moet doen voor stream sockets, is de data `send()` en. Het enige wat je moet doen voor datagram sockets, is je pakketje naar eigen inzicht inpakken, en met `sendto()` naar buiten sturen. De kernel bouwt de Transportlaag en Internetlaag voor je en de hardware doet de Netwerktogangslaag. Ahh, moderne technologie.

Hiermee eindigt ons korte uitstapje in de netwerk theorie. O ja, ik vergat je alles wat ik over routing te zeggen heb te vertellen: niks! Inderdaad, daar ga ik het niet over hebben. De router rukt de IP kop van het pakketje, kijkt in z'n routing tabel, bla bla bla. Kijk maar in de IP RFC¹⁰ als je het echt wilt weten. Als je er nooit wat over leert, ach, dat overleef je wel.

3. structs en Data Handling

Zo, we zijn er eindelijk. Het is tijd om het over programmeren te hebben. In dit hoofdstuk ga ik het hebben over verscheidene datatypes die gebruikt worden door de sockets interface, omdat het uitpluizen van sommige ervan een flinke kluit is.

Eerst de makkelijkste: een socket descriptor. Een socket descriptor is het volgende type:

```
int
```

Gewoon een doorsnee `int`.

Vanaf hier wordt het wat vaag, dus lees maar gewoon door en heb een beetje vertrouwen in me. Onthoud dit: er zijn twee byte volgorden: meest significante byte (most significant byte, soms een "octet" genoemd) eerst, of minst significante byte eerst. De eerste wordt de "Netwerk Byte Volgorde" (Network Byte Order) genoemd. Sommige machines slaan hun data intern in netwerk byte volgorde op, andere niet. Wanneer ik zeg dat iets in netwerk byte volgorde moet staan, moet je een functie (zoals `htons()`) aanroepen om het te veranderen van "Host Byte Volgorde"

naar "Netwerk Byte Volgorde". Als ik niet zeg dat het in "Netwerk Byte Volgorde" moet staan, dan moet het gewoon in Host Byte Volgorde staan.

Voor de nieuwsgierigen, "Netwerk Byte Volgorde" staat ook bekend als "Big-Endian Byte Order", wat zoveel betekent als "Grootste Byte Eerst".

Mijn Eerste Struct™—`struct sockaddr`. Deze structuur bevat socket adresinformatie voor vele sockettypen:

```
struct sockaddr {
    unsigned short    sa_family;    // adres familie, AF_XXX
    char              sa_data[14]; // 14 bytes voor protocol adres
};
```

`sa_family` kan een aantal dingen betekenen, maar in dit document zal het altijd de waarde `AF_INET` hebben. `sa_data` bevat een bestemmingsadres en poortnummer voor de socket. Dit is nogal onpraktisch aangezien je niet tot in den treure het adres met de hand in `sa_data` wil verpakken.

Om met `struct sockaddr` overweg te kunnen hebben programmeurs een parallelle structuur gemaakt: `struct sockaddr_in` ("in" als in "internet").

```
struct sockaddr_in {
    short int         sin_family;   // Adres familie
    unsigned short int sin_port;    // Poort nummer
    struct in_addr    sin_addr;     // Internet adres
    unsigned char     sin_zero[8]; // Zelfde grootte als struct sockaddr
};
```

Deze structuur maakt het makkelijk om te refereren naar elementen van het socket adres. Merk op dat `sin_zero` (welke is ingevoegd om de structuur op te vullen tot dezelfde grootte als een `struct sockaddr`) met `memset()` met allemaal nullen gevuld moet worden. Ook, en dit is het *belangrijke* stuk, een pointer naar een `struct sockaddr_in` kan gecast worden naar een pointer naar een `struct sockaddr` en vice versa. Dus hoewel `socket()` een `struct sockaddr*` wil, kun je gewoon een `struct sockaddr_in` gebruiken en op het laatste moment een cast uitvoeren! Merk verder op dat `sin_family` correspondeert met `sa_family` in een `struct sockaddr` en dat deze op "AF_INET" gezet moet worden. Tenslotte moeten `sin_port` en `sin_addr` in Netwerk Byte Volgorde staan!

"Maar", sputter je tegen, "hoe kan die hele structuur, `struct in_addr sin_addr`, nou in Netwerk Byte Volgorde staan?" Deze vraag vereist een nauwkeurig onderzoek van de `struct in_addr` structuur, een van de ergste verenigingen (unions) in de wereld:

```
// Internet adres (een structuur om historische redenen)
struct in_addr {
    unsigned long s_addr; // da's een 32-bit long, oftewel 4 bytes
};
```

Nou ja, *vroeger* was het een union, maar tegenwoordig lijkt het erop dat 'ie weg is. De groeten. Dus als je *ina* hebt declareerd als van het type `struct sockaddr_in`, dan refereert `ina.sin_addr.s_addr` naar het 4-bytes lange IP adres (in Netwerk Byte Volgorde). Let er wel op dat zelfs als je systeem nog steeds die afgrijselijke union gebruikt voor `struct in_addr`, je nog steeds naar het 4-bytes IP adres kunt refereren op precies dezelfde manier als ik hierboven heb gedaan (dit dankzij `#defines`.)

3.1. De Inboorlingen Converteren!

En zo zijn we netjes bij het volgende hoofdstuk aangeland. We hebben het al te lang over deze Netwerk naar Host Byte Volgorde conversie gehad—nou is het tijd voor actie!

Okie dokie. Er zijn twee typen die je kan converteren: `short` (twee bytes) en `long` (vier bytes). Deze functies werken ook voor de `unsigned` varianten. Stel dat je een `short` wil converteren van Host Byte Volgorde naar Netwerk Byte Volgorde. Begin met "h" voor "host", plak er "to" (als in "naar") aanvast, daarna "n" voor "netwerk" en "s" voor "short": h-to-n-s, oftewel `htons()` (lees: "Host to Network Short").

Dit is bijna te makkelijk...

Je kunt elke combinatie van "n", "h", "s" en "l" gebruiken die je maar wil, de echt stomme niet meegerekend. Zo is er bijvoorbeeld GEEN `stolh()` ("Short to Long Host") functie—niet op dit feestje, in elk geval. Wel zijn er:

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

Nou denk je misschien bij de hand te zijn door te vragen, "Wat moet ik doen om de byte volgorde van een `char` te veranderen?" Daarna denk je hopelijk, "Uh, laat maar." Je denkt misschien ook dat aangezien je machine een 68000 toch al netwerk byte volgorde gebruikt, je `htonl()` niet nodig hebt voor je IP adressen. Je zou gelijk hebben, *MAAR*, als je zou proberen je programma te porteren naar een andere machine die omgekeerde netwerk byte volgorde gebruikt, doet je programma het niet. Wees porteerbaar! Dit is een Unix wereld! (Hoewel Bill Gates je graag anders zou doen geloven.) Denk erom: zet je bytes in Netwerk Byte Volgorde voor je ze op het netwerk zet.

Nog een laatste puntje; waarom moeten `sin_addr` en `sin_port` in Netwerk Byte Volgorde staan in een `struct sockaddr_in`, maar `sin_family` niet? Het antwoord: `sin_addr` en `sin_port` worden ingekapseld in het pakket in de IP en UDP lagen, respectievelijk. Dus moeten ze in Netwerk Byte Volgorde staan. Het `sin_family` veld daarentegen wordt alleen door de kernel gebruikt om te bepalen wat voor type adres de structuur bevat, dus

moet het in Host Byte Volgorde staan. Tevens, aangezien *sin_family* niet over het netwerk wordt verstuurd, kan het gewoon in Host Byte Volgorde staan.

3.2. IP Adressen en Hoe Ermee Om te Gaan

Gelukkig voor jou zijn er een aantal functies die je in staat stellen om IP adressen te manipuleren. Nergens voor nodig om ze zelf te ontleden en met de hand in een `long` te proppen met de `<` operator.

Ten eerste, stel dat je een `struct sockaddr_in` `ina` hebt, en een IP adres "10.12.110.57" die je erin wil opslaan. De functie die je dan wil gebruiken, `inet_addr()`, converteert een IP adres in getallen-en-punten notatie naar een `unsigned long`. De toewijzing kan als volgt worden gedaan:

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

Merk op dat `inet_addr()` het adres al in Netwerk Byte Volgorde teruggeeft—je hoeft niet zelf `htonl()` aan te roepen. Vet!

Nou, het bovenstaande stukje code is niet erg robust omdat er geen foutencontrole plaats vindt. Zie je, `inet_addr()` geeft `-1` terug bij een fout. Ken je de binaire getallen nog een beetje? `(unsigned)-1` komt toevallig overeen met het IP adres `255.255.255.255`! Dat is het broadcastadres! Oepsie. Denk erom dat je wel je foutafhandeling goed voor elkaar hebt.

Eigenlijk is er een nettere interface die je kunt gebruiken in plaats van `inet_addr()`: het heet `inet_aton()` ("aton" betekent "ascii to network"):

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

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

En hier is een gebruiksvoorbeeld, tijdens het inpakken van een `struct sockaddr_in` (dit voorbeeld is beter te begrijpen wanneer je bij de hoofdstukken over `bind()` en `connect()` bent beland).

```
struct sockaddr_in mijn_addr;

mijn_addr.sin_family = AF_INET;           // host byte volgorde
mijn_addr.sin_port = htons(MYPORT);      // short, network byte volgorde
inet_aton("10.12.110.57", &(mijn_addr.sin_addr));
memset(&(mijn_addr.sin_zero), '\0', 8); // de rest van de struct op nul
```

`inet_aton()`, in tegenstelling tot vrijwel elke andere socket-gerelateerde functie, geeft een niet-nul waarde terug bij succes, en nul bij een fout. En het adres wordt teruggegeven in `inp`.

Helaas implementeren niet alle platforms `inet_aton()`, dus, hoewel het gebruik ervan de voorkeur heeft, wordt de oudere, meer bekende `inet_addr()` in deze handleiding gebruikt.

Okee, nou kun je een reeks IP adressen converteren naar hun binaire representaties. Maar andersom? Stel dat je een struct `in_addr` hebt en je wilt het in getallen-en-punten notatie weergeven? In dat geval heb je de functie `inet_ntoa()` nodig ("ntoa" betekent "network to ascii") zoals hier:

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

Dat drukt het IP adres af. Merk op dat `inet_ntoa()` een struct `in_addr` als argument wil, en niet een `long`. Merk ook op dat het een pointer naar char teruggeeft. Deze wijst naar een statisch opgeslagen char array binnen `inet_ntoa()` waardoor bij iedere aanroep naar `inet_ntoa()` het vorige IP adres wordt overschreven. Bijvoorbeeld:

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

drukt af:

```
adres 1: 10.12.110.57
adres 2: 10.12.110.57
```

Als je het adres wil bewaren, `strcpy()` het dan naar je eigen char array.

Dat was het voorlopig wat betreft dit onderwerp. Later leer je hoe je een string zoals "www.overheid.nl" kunt converteren naar het corresponderende IP adres (zie DNS, verderop).

4. Systeemaanroepen of Barsten

Dit is het hoofdstuk waar we in de systeemaanroepen duiken die je in staat stellen de netwerkfunctionaliteit van een Unix doos aan te spreken. Wanneer je één van deze functies aanroept neemt de kernel het over en doet al het werk automatisch voor je.

De plek waar de meeste mensen vast komen te zitten is de volgorde waarin ze deze dingen moeten aanroepen. In dat geval zijn de **man** pages nutteloos, zoals je waarschijnlijk al hebt ontdekt. Nou, om je uit die afgrijselijke situatie te helpen, heb ik geprobeerd om alle aanroepen in de volgende onderdelen *precies* zo uit te leggen (ongeveer) in dezelfde volgorde als je ze in je eigen programma's zou gebruiken.

Dat, samen met wat stukjes voorbeeldcode zo hier en daar, wat melk en koekjes (waar je toch echt zelf voor zult moeten zorgen), een stevig stel kloten en wat moed, en je zult data over het internet stralen als ware je de Zoon van Jon Postel!

4.1. `socket()`—Grijp Die File Descriptor!

Ik vrees dat ik het niet langer uit kan stellen—ik moet het gaan hebben over de `socket()` systeemaanroep. Hier gaat het om:

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

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

Maar wat zijn dat voor argumenten? Ten eerste, *domain* moet op "AF_INET" gezet worden, net zoals in de `struct sockaddr_in` (zie terug). Verder, het *type* argument vertelt de kernel wat voor soort socket dit is: `SOCK_STREAM` of `SOCK_DGRAM`. Tenslotte kun je *protocol* gewoon op "0" zetten om `socket()` op basis van het *type* zelf uit te laten zoeken welk protocol het moet gebruiken. (Opmerkingen: er zijn veel meer *domains* dan ik heb beschreven. Zie de `socket()` man page. Verder is er een "betere" manier om het *protocol* te achterhalen. Zie de `getprotobyname()` man page.)

`socket()` geeft je simpelweg een socket descriptor terug die je kan gebruiken in latere systeemaanroepen, of -1 bij een fout. De globale variabele `errno` krijgt de waarde van de fout (zie de `perror()` man page.)

In bepaalde documentatie is er sprake van een mystieke "PF_INET". Dit is een vreemd etherisch wezen dat zelden in de natuur wordt gesignaleerd, maar ik kan hier wel een beetje opheldering geven. Eens, lang geleden, werd er gedacht dat een adres familie (waar de "AF" in "AF_INET" voor staat) misschien meerdere protocollen zou ondersteunen, aan welke gerefereerd zou worden aan door hun protocol familie (waar "PF" in "PF_INET" voor staat). Dat gebeurde niet. Ook best. Dus de juiste manier is om AF_INET in je `struct sockaddr_in` te gebruiken en PF_INET in je aanroep naar `socket()`. Maar praktisch gezien kun je overal AF_INET gebruiken. En, aangezien W. Richard Stevens dat in zijn boek ook doet, doe ik dat hier ook.

Ja, ja, dat zal wel, maar wat heb ik eraan? Het antwoord is dat je er op zich eigenlijk niks aan hebt, en je verder moet lezen en meer systeemaanroepen moet doen om er wat zinnigs uit te krijgen.

4.2. `bind()`—Op Welke Poort Zit Ik?

Als je eenmaal een socket hebt, moet je die misschien nog associëren met een poort op je locale machine. (Dit is gebruikelijk als je gaat `listen()` en (luisteren) naar inkomende verbindingen op een specifieke poort—MUDs doen dit als ze je vertellen dat je moet "telnetten naar `w.x.y.z` op poort 6969".) Het poortnummer wordt door de kernel gebruikt om een inkomend pakketje met een bepaald proces z'n file descriptor te associëren. Als je alleen maar gaat `connect()` en, is dit misschien onnodig. Lees het toch maar, gewoon voor de kick.

Hier is de beschrijving door de `bind()` systeemaanroep:

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

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

`sockfd` is de socket file descriptor die door `socket()` wordt teruggegeven. `my_addr` is een pointer naar een `struct sockaddr` die informatie bevat over je adres, namelijk poort en IP adres. `addrlen` kun je instellen op `sizeof(struct sockaddr)`.

Pfoeh. Dat was een flinke kluijf. Tijd voor een voorbeeld:

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

#define MIJNPOORT 3490

main()
{
    int sockfd;
    struct sockaddr_in mijn_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0); // doe wat foutafhandeling!

    mijn_addr.sin_family = AF_INET;           // host byte volgorde
    mijn_addr.sin_port = htons(MYPORT);      // short, netwerk byte volgorde
    mijn_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
    memset(&(mijn_addr.sin_zero), '\0', 8); // de rest van de struct op nul

    // vergeet de foutafhandeling voor bind() niet:
    bind(sockfd, (struct sockaddr *)&mijn_addr, sizeof(struct sockaddr));
    .
    .
    .
```

Eer zijn een paar dingen die moeten worden opgemerkt: `mijn_addr.sin_port` staat in Netwerk Byte Volgorde. `mijn_addr.sin_addr.s_addr` ook. Iets anders om op te letten is dat de header files kunnen verschillen van systeem tot systeem. Als je zekerheid wil, zul je in je locale **man** pages moeten kijken.

Tenslotte, nou we het over `bind()` hebben, moet ik je vertellen dat een deel van het proces om je eigen IP adres en/of poort te verkrijgen kan worden geautomatiseerd:

```
mijn_addr.sin_port = 0; // kies willekeurig een ongebruikte poort
mijn_addr.sin_addr.s_addr = INADDR_ANY; // gebruik mijn IP adres
```

Zie je, door `mijn_addr.sin_port` op nul te zetten, geef je `bind()` de opdracht om een poort voor je te kiezen. Vergelijkbaar kun je door `mijn_addr.sin_addr.s_addr` op `INADDR_ANY` te zetten, hem automatisch het IP adres in laten vullen van de machine waar je proces op draait.

Als je op kleine dingetjes let, heb je misschien gezien dat ik `INADDR_ANY` niet in Netwerk Byte Volgorde heb gezet! Stout van me. Maar, ik heb zo m'n voorkennis: `INADDR_ANY` is feitelijk nul! nul blijft nul zelfs als je de bits door elkaar gooit. Maar, puristen zullen vast zeggen dat er een parallelle dimensie zou kunnen zijn waar `INADDR_ANY` misschien wel, zeg, 12 is en dat m'n code daar niet werkt. Dat vind ik wel best:

```
mijn_addr.sin_port = htons(0); // kies willekeurig een ongebruikte poort
mijn_addr.sin_addr.s_addr = htonl(INADDR_ANY); // gebruik mijn IP adres
```

Nou zijn we zo porteerbaar dat het bijna niet meer te geloven is. Ik wou het je maar even zeggen, aangezien de meeste code die je tegenkomt niet de moeite neemt om `INADDR_ANY` door `htonl()` te halen.

`bind()` geeft ook `-1` terug bij een fout en geeft `errno` de foutwaarde.

Nog iets om voor uit te kijken wanneer je `bind()` aanroept: ga niet te laag met je poort nummers. Alle poorten onder 1024 zijn GERESERVEERD (tenzij je de superuser bent)! Je kunt elk poortnummer daarboven krijgen, tot aan 65535 (vooropgesteld dat ze niet al door een ander programma worden gebruikt).

Soms zal je opvallen dat wanneer je een server herstart dat `bind()` faalt met de kreet "Address already in use" (adres is reeds in gebruik). Wat betekent dat? Nou, een deel van een socket dat verbonden was hangt nog rond in de kernel, en houdt de poort bezet. Je kan óf wachten tot 'ie weg is (duurt een minuutje of zo), of je kunt een stukje code aan je programma toevoegen waardoor het de poort opnieuw kan gebruiken, zo:

```
int yes=1;
// char='1'; // Solaris mensen gebruiken dit

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

Nog een kleine, echt allerlaatste opmerking over `bind()`: er zijn tijden dat je hem niet absoluut noodzakelijk hoeft aan te roepen. Als je aan het `connect()` en bent naar een andere machine en het zal je worst wezen wat je locale poort is (zoals met **telnet**, waar het je alleen maar om de bestemmingspoort gaat) kun je simpelweg `connect()` en. Dan zal er gecontroleerd worden of de socket ongebonden is, en het zonnodig `bind()` en aan een ongebruikte locale poort.

4.3. `connect()`—Hé jij daar!

Laten we eens even doen alsof je een telnet applicatie bent. Je gebruiker geeft je de opdracht (net zoals in de film *TRON*) een file descriptor te halen. You doet braaf wat je gezegd wordt en roept `socket()` aan. Vervolgens vertelt de gebruiker je om een verbinding te maken met "10.12.110.57" op poort "23" (de standaard telnet poort). Jeetje! Wat doe je nu?

Gelukkig voor jou, programma, lees je nou net het hoofdstuk over `connect()`—hoe een verbinding te maken met een andere host. Dus lees onverschrokken verder! Geen tijd te verliezen!

De `connect()` aanroep luidt als volgt:

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

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

`sockfd` is onze goede vriend, de socket descriptor, zoals teruggegeven door de `socket()` aanroep, `serv_addr` is een struct `sockaddr` die de bestemmingspoort en IP adres bevat, en `addrlen` kan gesteld worden op `sizeof(struct sockaddr)`.

Begint het al een beetje te dagen? Een voorbeeld:

```
#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;    // voor het bestemmingsadres

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

```

dest_addr.sin_family = AF_INET;           // host byte volgorde
dest_addr.sin_port = htons(DEST_PORT);   // short, netwerk byte volgorde
dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
memset(&(dest_addr.sin_zero), '\0', 8); // de rest van de struct op nul

// niet vergeten om connect() op fouten te controleren!
connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
.
.
.

```

Wederom, denk erom de return waarde van `connect()` te controleren—die geeft `-1` terug bij een fout en wijzigt de `errno` variabele.

Merk ook op dat we `bind()` niet hebben aangeroepen. Eigenlijk maakt ons lokale poortnummer ons geen barst uit; we willen alleen weten waar we naar toe gaan (de bestemmingspoort). De kernel kiest een lokale poort voor ons, en de site waarnaar we verbinding maken krijgt deze informatie allemaal automatisch van ons. Geen zorgen.

4.4. `listen()`—Wil iemand me alsjeblieft bellen?

Ok, even wat anders. Wat nou als je niet geen verbinding met een andere machine wil maken. Zeg maar dat je op inkomende verbindingen wil wachten en er dan wat mee wil doen. Het proces bestaat uit twee delen: eerst `listen()` (luister) je, daarna `accept()` eer je (zie verderop).

De `listen()` aanroep is vrij simpel, maar vereist wat uitleg:

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

`sockfd` is de gebruikelijke socket file descriptor van de `socket()` systeemaanroep. `backlog` is het aantal toegestane verbindingen op de inkomende wachtrij. Wat betekent dat? Nou, inkomende verbindingen zullen in deze rij moeten wachten totdat je ze `accept()` eert, en dit is het maximale aantal dat in de wachtrij mag. De meeste systemen stellen dit stilletjes op ongeveer 20; je komt ervast wel mee weg als je het op 5 of 10 instelt.

Wederom, zoals gewoonlijk, geeft `listen()` de waarde `-1` terug en wijzigt `errno` bij een fout.

Okee, zoals je je waarschijnlijk wel kunt voorstellen, moeten we `bind()` aanroepen voordat we `listen()` aanroepen, anders zorgt de kernel ervoor dat we op een willekeurige poort komen te luisteren. Bleh! Dus als je van plan bent om naar inkomende verbindingen te luisteren, is de volgorde om de systeemaanroepen uit te voeren als volgt:

```
socket();
```

```
bind();
listen();
/* accept() komt hier */
```

Ik laat het hierbij, aangezien het redelijk makkelijk te volgen is. (De code in het `accept()` hoofdstuk hieronder is meer compleet.) Het lastigste deel van deze hele constructie is de aanroep naar `accept()`.

4.5. `accept()` – "Bedankt voor het bellen met poort 3490."

Hou je vast–de `accept()` aanroep is wat eigenaardig! Wat er gaat gebeuren is het volgende: iemand ver weg zal proberen om te `connect()` en met je machine op een poort waarop je aan het `listen()` en bent. Hun verbinding zal in de wachtrij worden geplaatst in de hoop `geaccept()` eerd te worden. Je roept `accept()` aan en geeft opdracht de wachtende verbinding op te halen. Het geeft je vervolgens een *gloednieuwe socket file descriptor* terug voor het gebruik van deze ene verbinding! Dat klopt, opeens heb je *twee socket file descriptors* voor de prijs van één! De originele luistert nog steeds op je poort en de nieuw aangemaakte is eindelijk klaar om te gaan `send()` en `recv()`. We zijn er!

De aanroep is als volgt:

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

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

`sockfd` is de socket descriptor waarop `listen()` d wordt. Simpel. `addr` is meestal een pointer naar een locale `struct sockaddr_in`. Dit is waar de informatie over de inkomende verbinding zal worden opgeslagen (en waarmee je dus kunt bepalen van welke machine en van welke poort de inkomende verbinding komt). `addrlen` is een locale integer variabele die op `sizeof(struct sockaddr_in)` gezet dient te worden voordat z'n adres wordt doorgegeven aan `accept()`. `accept()` zal niet meer dan zoveel bytes in `addr` proberen op te slaan. Als het er minder instopt, zal het de waarde van `addrlen` aanpassen om dat weer te geven.

Raad eens? `accept()` geeft `-1` terug en past `errno` aan als er een fout optreedt. Dat had je vast nooit zelf verzonnen.

Zoals hiervoor is dit een flinke kluit om één keer te verwerken, dus hier is een stukje voorbeeldcode om eens te bestuderen:

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

#define MIJNPOORT 3490 // de poort waar gebruikers verbinding mee maken
```

```
#define BACKLOG 10      // hoeveel openstaande verbindingen de wachtrij mag bevatten

main()
{
    int sockfd, new_fd; // listen() en op sockfd, nieuwe verbinding op new_fd
    struct sockaddr_in mijn_addr; // mijn adresinformatie
    struct sockaddr_in hun_addr; // adresinformatie van de 'beller'
    int sin_size;

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

    mijn_addr.sin_family = AF_INET; // host byte volgorde
    mijn_addr.sin_port = htons(MIJNPOORT); // short, network byte volgorde
    mijn_addr.sin_addr.s_addr = INADDR_ANY; // automatisch mijn IP invullen
    memset(&(mijn_addr.sin_zero), '\0', 8); // de rest van de struct op nul

    // vergeet je foutafhandeling voor de volgende aanroepen niet:
    bind(sockfd, (struct sockaddr *)&mijn_addr, sizeof(struct sockaddr));

    listen(sockfd, BACKLOG);

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

Nogmaals, let erop dat we de socket descriptor *new_fd* gebruiken voor alle `send()` en `recv()` aanroepen. Als je in de loop van je programma maar één hele verbinding krijgt, kan je de luisterende *sockfd* `close()`n om te voorkomen dat er meer verbindingen op dezelfde poort komen te staan, als je dat zou willen.

4.6. `send()` en `recv()`—Praat tegen me, schatje!

Deze twee functies zijn bedoeld om mee te communiceren over stream sockets of verbonden datagram sockets. Als je normale onverbonden (unconnected) datagram sockets wil gebruiken, moet je het hoofdstuk over `sendto()` en `recvfrom()`, verderop lezen.

De `send()` aanroep:

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

`sockfd` is de socket descriptor waar je data naartoe wil sturen (hetzij degene door `socket()` teruggegeven, of degene die je van `accept()` kreeg). `msg` is een pointer naar de data die je wil versturen, en `len` is de lengte van die data in bytes. Zet `flags` maar gewoon op 0. (Zie de `send()` man page voor meer informatie over de vlaggen (flags).)

Wat voorbeeldcode is bijvoorbeeld:

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

`send()` geeft het aantal bytes terug dat daadwerkelijk werd verstuurd—*dit kan minder zijn dat het aantal dat je verteld had dat verstuurd moest worden!* Zie je, soms geef je opdracht om een hele berg data te versturen terwijl 'ie het gewoon niet aankan. In dat geval zal er zoveel mogelijk van de data verstuurd worden, en erop vertrouwd worden dat je de rest later verstuurd. Onthoud, als de door `send()` teruggegeven waarde niet gelijk is aan de waarde van `len`, ben jij verantwoordelijk om de rest van de data te versturen. Het goede nieuws is dit: Als het pakketje klein is (minder dan 1K of zo), zal het *waarschijnlijk* wel lukken om het in z'n geheel te versturen. Wederom wordt bij een fout `-1` teruggegeven, en `errno` gewijzigd naar het foutnummer.

De `recv()` aanroep is in veel opzichten vergelijkbaar:

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

`sockfd` is de socket descriptor waarvan gelezen wordt, `buf` is de buffer waar de data in wordt opgeslagen, `len` is de maximum lengte van de buffer, en `flags` kan wederom op 0 gezet worden. (Zie de `recv()` man page voor vlag (flag) informatie.)

`recv()` geeft het aantal bytes dat daadwerkelijk in de buffer werd opgeslagen terug, of `-1` bij een fout (met `errno` overeenkomstig gewijzigd).

Wacht! `recv()` kan 0 teruggeven. Dit betekent slechts één ding: de andere kant heeft de verbinding verbroken! Een return-waarde van 0 is `recv()`'s manier om je te laten weten dat dit gebeurd is.

Nou, kijk eens aan, dat was makkelijk, nietwaar? Nou kan je data heen en weer sturen over stream sockets! Jippie! Nou ben je een Unix Netwerk Programmeur!

4.7. `sendto()` and `recvfrom()`—Zeg wat tegen me, op z'n DGRAM's

"Dit is allemaal wel leuk en aardig", hoor ik je zeggen, "maar wat moet ik nou aan met onverbonden datagram sockets?" Geen probleem. Ik weet precies wat je nodig hebt.

Aangezien datagram sockets niet verbonden worden aan een andere machine, raad eens wat we op moeten geven voordat we een pakketje verzenden? Precies! Het bestemmingsadres! Zo zit 't:

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

Zoals je kunt zien is deze aanroep in principe hetzelfde als de `send()` aanroep met toevoeging van twee stukjes informatie. `to` is een pointer naar een `struct sockaddr` (welke je waarschijnlijk als een `struct sockaddr_in` hebt declareert en op het laatste moment gecast) welke het bestemmings-IP en -poort bevat. `tolen` kan simpelweg op `sizeof(struct sockaddr)` gezet worden.

Net zoals met `send()` geeft `sendto()` het aantal bytes terug dat daadwerkelijk werd verstuurd (welke, wederom, minder kan zijn dan het aantal bytes dat je had opgegeven om te versturen!), of `-1` bij een fout.

Ook `recv()` en `recvfrom()` lijken op elkaar. De beschrijving van `recvfrom()` is:

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

Ook hier zijn er overeenkomsten met `recv()` met de toevoeging van een aantal extra velden. `from` is een pointer naar een lokale `struct sockaddr` wat opgevuld zal worden met het IP adres en poortnummer van de oorspronkelijke machine. `fromlen` is een pointer naar een lokale `int` die geïnitieerd moet worden op `sizeof(struct sockaddr)`. Wanneer de functie terugkeert zal `fromlen` de lengte van het adres in `from` bevatten.

`recvfrom()` geeft het aantal ontvangen bytes terug, of `-1` bij een fout (met `errno` overeenkomstig gewijzigd).

Onthoud, als je een datagram socket `connect()`, kun je simpelweg `send()` en `recv()` gebruiken voor je transacties. De socket zelf is nog steeds een datagram socket en de pakketjes zullen nog steeds UDP gebruiken, maar de socket interface zal automatisch het bestemmingsadres en broninformatie voor je toevoegen.

4.8. `close()` en `shutdown()`—Oprotten!

Pfoe! De hele dag lang hebben je zitten `send()` en `recv()` en, en nou heb je er genoeg van. Je bent klaar om de verbinding op je socket descriptor te sluiten. Da's makkelijk. Je kunt de gewone Unix file descriptor functie `close()` gebruiken:

```
close(sockfd);
```

Dit voorkomt verdere lees- en schrijfacties van en naar de socket. Iedereen die probeert om te lezen van of te schrijven naar de socket vanaf de andere kant van de verbinding krijgt een foutmelding voor z'n kiezen.

Mocht je nou wat meer controle willen hebben over hoe de socket afsluit, dan kun je de `shutdown()` functie gebruiken. Hiermee kun je de communicatie in een bepaalde richting afbreken, of in allebei de richtingen (zoals `close()` doet). Beschrijving:

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

`sockfd` is de socket file descriptor die je wilt afsluiten, en `how` is een van de volgende:

- 0 – Verdere ontvangsten worden geweigerd
- 1 – Verdere verzendingen worden geweigerd
- 2 – Verdere verzendingen en ontvangsten worden geweigerd (zoals `close()`)

`shutdown()` geeft 0 terug bij succes, en -1 bij een fout (met `errno` onvereenkomstig gewijzigd).

Als je je niet te goed acht om `shutdown()` op onverbonden sockets te gebruiken, zal het de socket simpelweg onbeschikbaar maken voor verdere `send()` en `recv()` aanroepen (denk eraan dat je deze kunt gebruiken als je je datagram socket maar `connect()`).

Het is belangrijk om op te merken dat `shutdown()` de file descriptor niet feitelijk afsluit—het verandert alleen z'n bruikbaarheid. Om een socket descriptor vrij te geven moet je `close()` gebruiken.

Makkelijk zat.

4.9. `getpeername()`—Wie ben jij?

Deze functie is zo makkelijk.

Zo makkelijk, dat ik hem bijna z'n eigen hoofdstuk niet heb gegeven. Maar hier is 'ie toch maar.

De functie `getpeername()` vertelt je wie er aan de andere kant van een verbonden socket hangt. De beschrijving:

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

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

sockfd is de descriptor van de verbonden socket, *addr* is een pointer naar een `struct sockaddr` (of een `struct sockaddr_in`) waar de informatie over de andere kant van de verbinding wordt opgeslagen, en *addrlen* is een pointer naar een `int`, die geïnitieerd wordt op `sizeof(struct sockaddr)`.

Bij een fout geeft de functie `-1` terug en wijzigt *errno* overeenkomstig.

Als je eenmaal hun adres hebt, kun je `inet_ntoa()` of `gethostbyaddr()` gebruiken om het af te drukken of meer informatie te krijgen. Nee, hun login naam krijg je niet. (Ok, ok. Als de andere machine een ident daemon heeft draaien, is dit mogelijk. Maar dat licht buiten het bestek van dit document. Kijk op RFC-1413¹¹ voor meer informatie.)

4.10. `gethostname()` –Wie ben ik?

Nog makkelijker dan `getpeername()` is de `gethostname()` functie. Dit geeft de naam van de computer waar je programma op draait terug. De naam kan dan gebruikt worden met `gethostbyname()`, verderop, om het IP adres van je lokale machine te bepalen.

Wat is er nou leuker? Ik zou wel een paar dingen kunnen bedenken, maar dat heeft niks met socket programmeren te maken. Hoe dan ook, hier is de beschrijving:

```
#include <unistd.h>

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

De argumenten zijn eenvoudig: *hostname* is een pointer naar een array van chars die de hostnaam bevatten na terugkeer van de functie, en *size* is de lengte van de *hostname* array in bytes.

De functie geeft `0` terug bij succes, en `-1` bij een fout, waarbij *errno* zoals gewoonlijk wordt gewijzigd.

4.11. DNS –Jij zegt "www.overheid.nl", ik zeg "193.78.30.5"

Mocht je niet weten wat DNS is, het staat voor "Domain Name Service (Domein Naam Service)". In een notendop, je vertelt het wat het leesbare adres is voor een site, en vervolgens geeft het je het IP adres terug (zodat je het kunt gebruiken met `bind()`, `connect()`, `sendto()` of waar je het ook maar voor nodig hebt). Op deze manier, wanneer iemand intypt:

```
$ telnet www.overheid.nl
```

dan zoekt **telnet** uit dat het moet `connect()` en met "193.78.30.5".

Maar hoe werkt het? Je zult de functie `gethostbyname()` nodig hebben:

```
#include <netdb.h>

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

Zoals je ziet geeft het een pointer naar een `struct hostent` terug, waarvan de layout als volgt is:

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

En hier volgend de beschrijvingen van de velden in de `struct hostent`:

- `h_name` – Officiële naam van de machine.
- `h_aliases` – Een NULL-getermineerde array van alternatieve namen voor de machine.
- `h_addrtype` – Het adrestype dat teruggegeven wordt; meestal `AF_INET`.
- `h_length` – De lengte van het adres in bytes.
- `h_addr_list` – Een nulgetermineerde array van netwerkadressen voor de machine. Machine-adressen staan in Netwerk Byte Volgorde.
- `h_addr` – Het eerste adres in `h_addr_list`.

`gethostbyname()` geeft als return-waarde een pointer naar de gevulde `struct hostent`, of NULL bij een fout. (Maar `errno` wordt *niet* gewijzigd—in plaats daarvan wordt `h_errno` gewijzigd. Zie `herror()`, verderop.)

Maar hoe wordt het gebruikt? Soms (merken we op als we computerhandleidingen lezen) is informatie naar het hoofd van de gebruiker slingeren niet genoeg. Deze functie is zeker makkelijker te gebruiken dan het lijkt.

Hier is een voorbeeldprogramma¹²:

```
/*
** getip.c - een hostnaam 'lookup' demo
*/

#include <stdio.h>
```

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

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

    if (argc != 2) { /* commandoregel op fouten checken */
        fprintf(stderr, "gebruik: getip adres\n");
        exit(1);
    }

    if ((h=gethostbyname(argv[1])) == NULL) { /* host info ophalen */
        perror("gethostbyname");
        exit(1);
    }

    printf("Hostnaam : %s\n", h->h_name);
    printf("IP Adres : %s\n", inet_ntoa(*(struct in_addr *)h->h_addr));

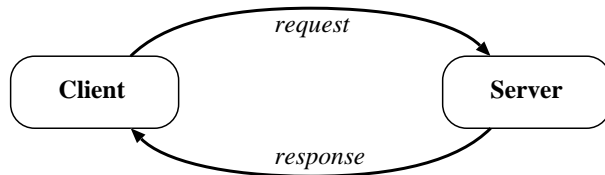
    return 0;
}
```

Met `gethostbyname()` kun je `perror()` niet gebruiken (aangezien `errno` niet gebruikt wordt). Gebruik in plaats daarvan `perror()`.

Het is redelijk voordehandliggend. Je geeft de string die de machinenaam ("www.overheid.nl") bevat door aan `gethostbyname()` en grijpt vervolgens de informatie uit de teruggeven struct `hostent`.

De enige mogelijke afwijking zou in het afdrukken van het IP adres kunnen zitten. `h->h_addr` is een `char *`, maar `inet_ntoa()` verwacht een `struct in_addr`. Dus cast ik `h->h_addr` naar een `struct in_addr *`, en vervolgens derefereren (dereference) om bij de data te kunnen.

Figuur 2. Client-Server Interactie.



5. Client-Server Achtergrond

Het is een client-server wereld, schat. Zo'n beetje alles op het netwerk gaat over client processen die praten met server processen en vice-versa. Neem bijvoorbeeld **telnet**. Wanneer je verbinding maakt met een andere host op poort 23 met telnet (de client), wordt een programma op die host (de **telnet** genoemd, de server) tot leven geroepen. Het handelt de inkomende telnet verbinding af, zet je voor een login prompt, etc.

De uitwisseling van informatie tussen client en server is samengevat in Figuur 2.

Merk op dat het client-server paar `SOCK_STREAM` kan praten, of `SOCK_DGRAM`, of wat dan ook (zolang ze maar het zelfde spreken). Een paar goede voorbeelden van client-server paren zijn **telnet/telnetd**, **ftp/ftpd**, of **bootp/bootpd**. Iedere keer dat je **ftp** gebruikt, is er een programma aan de andere kant, **ftpd**, dat je tot dienst staat.

Vaak draait er maar één server op een machine, en die server handelt meerdere clients af m.b.v. `fork()`. De basisroutine is: de server wacht op een verbinding, `accept()` deze, en `fork()` t een 'child process' (kind proces) om het af te laten handelen. Dit is wat onze voorbeeldserver doet in het volgende onderdeel.

5.1. Een eenvoudige Stream Server

Het enige wat deze server doet is het versturen van de string "Hallo, wereld!\n" over een stream verbinding. Het enige wat je moet doen om deze server te testen is het in het een venster te draaien, en ernaartoe telnetten via een ander met:

```
$ telnet anderehostnaam 3490
```

Waarbij `anderehostnaam` de naam is van de machine waarop je de server draait.

De servercode¹³:

```
/*
** server.c - een stream socket 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>

/* de poort waar gebruikers verbinding mee maken */
#define MIJNPOORT 3490

/* hoeveel nog af te handelen verbindingen er in de wachtrij passen */
#define BACKLOG 10

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

int main(void)
{
    int sockfd, new_fd; /* luisteren op sock_fd, nieuwe verbinding op new_fd */
    struct sockaddr_in mijn_adres; /* mijn adresinformatie */
    struct sockaddr_in hun_adres; /* connector's adresinformatie */
    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);
    }

    mijn_adres.sin_family = AF_INET; /* host byte volgorde */
    mijn_adres.sin_port = htons(MIJNPOORT); /* short, netwerk byte volgorde */
    mijn_adres.sin_addr.s_addr = INADDR_ANY; /* automatisch vullen met mijn IP */
```

```
memset(&(mijn_adres.sin_zero), '\0', 8); /* rest van de struct op nul */

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

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

sa.sa_handler = sigchld_handler; /* alle dode processen opvegen */
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
if (sigaction(SIGCHLD, &sa, NULL) == -1) {
    perror("sigaction");
    exit(1);
}

while(1) { // hoofd accept() lus
    sin_size = sizeof(struct sockaddr_in);
    if ((new_fd = accept(sockfd, (struct sockaddr *)&hun_adres, &sin_size)) == -1) {
        perror("accept");
        continue;
    }
    printf("server: verbinding gekregen van %s\n", inet_ntoa(hun_adres.sin_addr));
    if (!fork()) { /* dit is het kind proces */
        close(sockfd); /* kind heeft de listener niet nodig */
        if (send(new_fd, "Hallo, wereld!\n", 15, 0) == -1)
            perror("send");
        close(new_fd);
        exit(0);
    }
    close(new_fd); /* ouder heeft deze niet nodig */
}

return 0;
}
```

Mocht je nieuwsgierig zijn, ik heb de code in één grote `main()` functie gestopt om de syntaxis een beetje helder te houden (hoop ik). Voel je vrij om het in kleinere functies te verdelen als je dat wil.

Het hele `sigaction()` gebeuren is misschien ook nieuw voor je—da's niet erg. De code die er staat is verantwoordelijk voor het oogsten van zombie processen die tevoorschijn komen als de `fork()` te kind processen afsluiten. Als je heel veel zombies genereert en ze niet opruimt, krijg je je systeembeheerder op je dak.

Je kunt de data van deze server krijgen door de client te gebruiken die in het volgende onderdeel staat.

5.2. Een Eenvoudige Stream Client

Dit geval is nog makkelijker dan de server. Alles wat 'ie doet is verbinding maken met de host die je aangeeft op de commandoregel, op poort 3490. Het haalt de string op die de server verstuurt

De source voor de client¹⁴:

```
/*
** client.c - een stream socket client demo
*/

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

/* de poort waar de client verbinding mee maakt */
#define POORT 3490

/* max. aantal bytes dat we per keer kunnen ophalen */
#define MAXDATASIZE 100

int main(int argc, char *argv[])
{
    int sockfd, aant_bytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in hun_adres; /* connector's adresinformatie */

    if (argc != 2) {
        fprintf(stderr, "gebruik: client hostnaam\n");
    }
}
```

```
    exit(1);
}

if ((he = gethostbyname(argv[1])) == NULL) { /* host info ophalen */
    perror("gethostbyname");
    exit(1);
}

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

hun_adres.sin_family = AF_INET; /* host byte volgorde */
hun_adres.sin_port = htons(POORT); /* short (16 bit), netwerk byte volgorde */
hun_adres.sin_addr = *((struct in_addr *)he->h_addr);
memset(&(hun_adres.sin_zero), '\0', 8); /* rest van de struct op nul */

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

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

buf[aant_bytes] = '\0';

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

close(sockfd);

return 0;
}
```

Merk op dat als je de server niet start voordat je de client draait, `connect()` de melding "Connection refused" (Verbinding geweigerd) teruggeeft. Erg handig.

5.3. Datagram Sockets

Eigenlijk heb ik hier weinig over te zeggen, dus laat ik maar gewoon een paar voorbeeldprogramma's zien: `talker.c` en `listener.c`.

listener zit op een machine te wachten op binnenkomende pakketjes op poort 4950. **talker** verstuurt een pakketje naar de poort op de aangegeven machine, met daarin wat de gebruiker ook maar op de commandoregel intikt.

Hier is de source voor `listener.c`¹⁵:

```
/*
** listener.c - een datagram socket "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>

/* de poort waar gebruikers verbinding mee maken */
#define MIJNPOORT 4950

#define MAXBUFLEN 100

int main(void)
{
    int sockfd;
    struct sockaddr_in mijn_adres; /* mijn adresinformatie */
    struct sockaddr_in hun_adres; /* connector's adresinformatie */
    int adres_len, aant_bytes;
    char buf[MAXBUFLEN];

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

    mijn_adres.sin_family = AF_INET; /* host byte volgorde */
    mijn_adres.sin_port = htons(MIJNPOORT); /* short, netwerk byte volgorde */
    mijn_adres.sin_addr.s_addr = INADDR_ANY; /* automatisch vullen met mijn IP */
    memset(&(mijn_adres.sin_zero), '\0', 8); /* rest van de struct op nul */
}
```

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

    adres_len = sizeof(struct sockaddr);
    if ((aant_bytes = recvfrom(sockfd, buf, MAXBUFLen - 1 , 0,
(struct sockaddr *)&hun_adres, &adres_len)) == -1) {
        perror("recvfrom");
        exit(1);
    }

    printf("pakket gekregen van %s\n",inet_ntoa(hun_adres.sin_addr));
    printf("pakket is %d bytes lang\n",aant_bytes);
    buf[aant_bytes] = '\0';
    printf("pakket bevat \"%s\"\n",buf);

    close(sockfd);

    return 0;
}
```

Merk op dat we in de aanroep naar `socket()` eindelijk `SOCK_DGRAM` gebruiken. Merk ook op dat het niet nodig is om te `listen()` en of te `accept()`en. Dit is een van die extra's van het gebruik van ongebonden datagram sockets!

Vervolgens komt de source voor `talker.c`¹⁶:

```
/*
** talker.c - een datagram "client" demo
*/

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

/* de poort waar gebruikers verbinding mee maken */
#define MIJNPOORT 4950
```

```
int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in hun_adres; /* connector's adresinformatie */
    struct hostent *he;
    int aant_bytes;

    if (argc != 3) {
        fprintf(stderr, "gebruik: talker hostnaam bericht\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) { /* host info ophalen */
        perror("gethostbyname");
        exit(1);
    }

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

    hun_adres.sin_family = AF_INET; /* host byte volgorde */
    hun_adres.sin_port = htons(MIJNPOORT); /* short, netwerk byte volgorde */
    hun_adres.sin_addr = *((struct in_addr *)he->h_addr);
    memset(&(hun_adres.sin_zero), '\0', 8); /* rest van de struct op nul */

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

    printf("%d bytes verzonden naar %s\n", aant_bytes, inet_ntoa(hun_adres.sin_addr));

    close(sockfd);

    return 0;
}
```

En da's alles! Draai **listener** op een of andere machine, en **talker** op een andere. Kijk ze eens communiceren! Pret en vermaak voor de hele nucleaire familie!

Op een piepklein stukje detail na dan, dat ik al vele malen eerder heb verteld: verbonden datagram sockets. Ik moet het er hier over hebben, aangezien we in het datagram onderdeel van het document zijn. Laten we eens zeggen dat **talker** de functie `connect()` aanroept en het adres van de **listener** specificeert. Vanaf dat moment mag de **talker** alleen verzenden naar en ontvangen van het bij de `connect()` aanroep opgegeven adres. Om deze reden hoeft je `sendto()` en `recvfrom()` niet te gebruiken; je gebruikt simpelweg `send()` en `recv()`.

6. Enigszins Geavanceerde Technieken

Deze zijn niet *echt* geavanceerd, maar ze liggen voorbij de reeds behandelde basisstof. Eigenlijk, als je tot hier bent gekomen, mag je jezelf als redelijk bedreven beschouwen in de basics van Unix netwerk programmeren! Gefeliciteerd!

Dus hier gaan we dapper de nieuwe wereld binnen van de wat esoterischer zaken die je zou willen leren over sockets. Flink zijn!

6.1. Blocking

Blokkering ("Blocking"). Je hebt erover gehoord—maar wat is het nou eigenlijk? In een notendop, "block" is techie jargon voor "slaap". Je hebt vast al wel opgemerkt dat wanneer je **listener** draait, het daar maar een beetje zit te wachten tot er een pakketje komt. Wat er gebeurt is dat het `recvfrom()` aanroept terwijl er nog geen data was, en zodoende wordt er van `recvfrom()` gezegd dat 'ie "blockt" (slapen dus), totdat er data arriveert.

Veel functies blokkeren. `accept()` blokkeert. Alle `recv()` functies blokkeren. De reden waarom ze dit doen is omdat ze dat mogen. Wanneer je de socket descriptor aanmaakt met `socket()`, zet de kernel deze op "blocking". Als je niet wilt dat een socket blokkeert, moet je een aanroep maken naar `fcntl()`:

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

Door een socket op "non-blocking" te zetten kun je de socket peilen ("poll") voor data. Als je probeert te lezen van een non-blocking socket en er staat geen data te wachten, mag de socket niet blokkeren—het geeft dan `direct -1 terug` en zet `errno` op `EWOULDBLOCK`.

Algemeen gezegd, echter, is deze manier van "polling" een slecht idee. Als je je programma in een bezig-wacht (busy-wait) zet om data van de socket te halen, vreet het CPU-tijd op als een gek. Een meer elegante oplossing voor het controleren of er ergens data te lezen valt komt in het volgende onderdeel aan bod bij `select()`.

6.2. `select()`—Synchrone I/O Multiplexing

Deze functie is ietwat eigenaardig, maar erg nuttig. Neem de volgende situatie: je bent een server en je wil luisteren naar inkomende connecties maar ook blijven lezen van de connecties die je al hebt.

Geen probleem, zeg je, gewoon een `accept()` en een paar `recv()`s. Niet zo snel, knul! Wat nou als je aan het blocken bent op een `accept()` aanroep? Hoe ga je tegelijkertijd data `recv()`en? "Non-blocking sockets gebruiken!" Echt niet! Je wil de CPU niet de hele tijd bezig houden. Maar wat dan?

`select()` geeft je de kracht om verscheidene sockets tegelijkertijd in de gaten te houden. Het vertelt je welke klaar zijn om van gelezen te worden, welke klaar zijn om naar te schrijven, en welke sockets uitzonderingen ("exceptions") hebben gegenereerd, als je dat echt wil weten.

Zonder verder gedraal geef ik je hier de beschrijving van `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);
```

De functie houdt verzamelingen ("sets") van file descriptors in de gaten; in het bijzonder `readfds`, `writefds` en `exceptfds`. Als je wil weten of je van standaard invoer (standard input, "stdin") en een of andere socket descriptor `sockfd` kunt lezen, voeg dan de file descriptors 0 en `sockfd` toe aan de verzameling `readfds`. De parameter `numfds` moet de waarde van de hoogste file descriptor plus één krijgen. In dit voorbeeld moet het dus op `sockfd+1` ingesteld worden, aangezien dit gegarandeerd hoger is dan `stdin` (0).

Wanneer `select()` terugkeert, wordt `readfds` aangepast om te reflecteren welke van de file descriptors die je hebt geselecteerd, klaar is om van te lezen. Je kunt het testen met de macro `FD_ISSET()`, zoals verderop.

Voordat we echt verder gaan wil ik het even hebben over hoe je deze verzamelingen kunt manipuleren. Elke verzameling is van het type `fd_set`. De volgende macro's opereren over dit type:

- `FD_ZERO(fd_set *set)` – wist een file descriptor set
- `FD_SET(int fd, fd_set *set)` – voegt `fd` toe aan de set
- `FD_CLR(int fd, fd_set *set)` – verwijdert `fd` uit de set

- `FD_ISSET(int fd, fd_set *set)` – Test om te bepalen of `fd` in de verzameling zit

Tenslotte, wat is dit vage type `struct timeval`? Nou, soms wil je geen eeuwigheid wachten tot iemand je data stuurt. Misschien wil je eens in de 96 seconden een "Nog bezig..." berichtje afdrukken naar de terminal terwijl er nog niks gebeurd is. Deze tijdsstructuur stelt je in staat om een "timeout" periode aan te geven. Als de tijd verstreken is en `select()` heeft nog steeds geen file descriptors die klaar zijn, keert hij terug zodat je verder kunt.

De `struct timeval` heeft de volgende velden:

```
struct timeval {
    int tv_sec;      // seconden
    int tv_usec;    // microseconden
};
```

Zet simpelweg `tv_sec` op het aantal te wachten seconden, en zet `tv_usec` op het aantal te wachten microseconden. Ja, *microseconden* dus, niet milliseconden. Er zitten 1.000 microseconden in een milliseconde, en 1.000 milliseconden in een seconde. Dus zitten er 1.000.000 microseconden in een seconde. Maar waarom "usec"? De "u" moet een beetje op de griekse letter μ (Mu) lijken die we gebruiken voor "micro". Verder, wanneer de functie terugkeert, *zou timeout* bijgewerkt kunnen zijn om de resterende tijd weer te geven. Dit hangt af van de Unix-smaak die je draait.

Joepie! We hebben een microseconde resolutie timer! Nou, reken er maar niet op. Standaard Unix "timeslices" zijn zo rond de 100 milliseconden, dus dikke kans dat je minstens zo lang moet wachten, hoe klein je je `struct timeval` ook instelt.

Andere zaken van belang: Als je de velden in je `struct timeval` instelt op 0, genereert `select()` meteen een time-out, daarmee effectief alle file descriptors in je verzamelingen peilend. Als je de parameter `timeout` op `NULL` instelt, zal er nooit een timeout gegenereert worden, waardoor er gewacht zal worden tot er een file descriptor vrijkomt. Tenslotte, als het je niet uitmaakt of je moet wachten op een bepaalde verzameling, kun je deze ook op `NULL` zetten in de aanroep naar `select()`.

Het volgende stukje code¹⁷ wacht 2.5 seconden tot er iets op de standaard invoer verschijnt:

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

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

/* file descriptor voor standaard invoer */
#define STDIN 0
```

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

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

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

    /* writefds en exceptfds zijn niet van belang */
    select(STDIN+1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds))
        printf("Een toets werd ingedrukt!\n");
    else
        printf("Timeout.\n");

    return 0;
}
```

Als je op een "line-buffered terminal" zit, zul je op RETURN moeten drukken, anders krijg je alsnog een timeout voor je kiezen.

Misschien denken sommigen van jullie nu dat dit een geweldige manier is om op data te wachten op een datagram socket en je hebt gelijk: het zou *kunnen*. Sommige Unices kunnen `select()` op deze manier gebruiken, en sommige niet. Je zou moeten kijken wat je locale man page hier over te zeggen heeft als je het wil proberen.

Sommige Unices updaten de tijd in je `struct timeval` om de nog te resterende tijd tot een timeout weer te geven. Maar andere weer niet. Vertrouw daar maar niet op als je portable wil zijn. (Gebruik `gettimeofday()` als je de verlopen tijd in de gaten wil houden. 't Zuigt, ik weet het, maar zo is het nou eenmaal.)

Wat gebeurt er als een socket in de leesverzameling de verbinding sluit? In dat geval keert `select()` terug met die socket descriptor ingesteld als "klaar om te lezen". Wanneer je er dan eenmaal van `recv()`t, geeft `recv()` de waarde 0 terug. Op die manier weet je dat de client de verbinding heeft verbroken.

Nog een laatste opmerking over `select()`: als je een socket hebt dat aan het `listen()` en is, kun je controleren of er een nieuwe verbinding is door de file descriptor van die socket in de `readfds`-verzameling te zetten.

En dat, vrienden, is een snelle blik op de almachtige `select()` functie.

Maar, op verzoek volgt hier een uitgebreid voorbeeld. Helaas is het verschil tussen het uiterst simpele voorbeeld hierboven en het voorbeeld hieronder significant. Maar kijk maar eens, en lees dan de beschrijving die er op volgt.

Dit programma¹⁸ gedraagt zich als een simpele multi-user chat server. Start het in een venster, en **telnet** er naartoe (**telnet hostname 9034**) vanuit meerdere andere vensters.. Wanneer je iets typt in een **telnet**-sessie, zou het moeten verschijnen alle andere vensters.

```
/*
** selectserver.c - een afgezaagde multi-user chat server
*/

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

#define POORT 9034 /* de poort waar we op luisteren */

int main(void)
{
    fd_set master; /* hoofd file descriptor lijst */
    fd_set read_fds; /* tijdelijke file descriptor lijst voor select() */
    struct sockaddr_in mijn_adres; /* server adres */
    struct sockaddr_in hun_adres; /* client adres */
    int fdmax; /* maximum file descriptor nummer */
    int listener; /* luisterende socket descriptor */
    int newfd; /* nieuwe ge-accepteerde socket descriptor */
    char buf[256]; /* buffer voor client data */
    int aant_bytes;
    int yes=1; /* voor setsockopt() SO_REUSEADDR, verderop */
    int adres_len;
    int i, j;

    FD_ZERO(&master); /* de hoofd en tijdelijke sets op nul */
    FD_ZERO(&read_fds);

    /* luistersocket ophalen */
    if ((listener = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    /* hinderlijke "address already in use" foutmelding verhelpen */
    if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes,
                   sizeof(int)) == -1) {
```

```
        perror("setsockopt");
        exit(1);
    }

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

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

    /* luistersocket aan hoofdset toevoegen */
    FD_SET(listener, &master);

    /* grootste file descriptor bijhouden */
    fdmax = listener; /* tot nu toe is het deze */

    // hoofdloop
    for(;;) {
        read_fds = master; /* kopiëren */
        if (select(fdmax + 1, &read_fds, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(1);
        }

        /* bestaande verbindingen doorzoeken voor te lezen data */
        for(i = 0; i <= fdmax; i++) {
            if (FD_ISSET(i, &read_fds)) { /* we hebben er een! */
                if (i == listener) {
                    /* nieuwe verbindingen verwerken */
                    adres_len = sizeof(hun_adres);
                    if ((newfd = accept(listener, (struct sockaddr *)&hun_adres,
                        &adres_len)) == -1) {
                        perror("accept");
                    } else {
                        FD_SET(newfd, &master); /* aan hoofdset toevoegen */
                        if (newfd > fdmax) { /* maximum in de gaten houden */

```

```

        fdmax = newfd;
    }
    printf("selectserver: nieuwe verbinding van %s op "
           "socket %d\n", inet_ntoa(hun_adres.sin_addr), newfd);
}
} else {
    /* data van client verwerken */
    if ((aant_bytes = recv(i, buf, sizeof(buf), 0)) <= 0) {
/* fout, of client sloot verbinding */
if (aant_bytes == 0) {
        /* verbinding gesloten */
        printf("selectserver: socket %d hing op\n", i);
    } else {
        perror("recv");
    }
    close(i); /* doe! */
    FD_CLR(i, &master); /* uit hoofdset verwijderen */
} else {
/* we kregen data van een client */
    for(j = 0; j <= fdmax; j++) {
        /* versturen naar iedereen! */
        if (FD_ISSET(j, &master)) {
            /* behalve de 'listener' en onszelf */
            if (j != listener && j != i) {
                if (send(j, buf, aant_bytes, 0) == -1) {
                    perror("send");
                }
            }
        }
    }
}
} /* dit is ZO LELIJK! */
}
}
}
return 0;
}

```

Merk op dat ik twee file descriptor sets in de code heb: *master* en *read_fds*. De eerste, *master*, bevat alle socket descriptors die momenteel verbonden zijn, evenals de socket descriptor die luistert naar nieuwe verbindingen.

De reden waarom ik de *master* set gebruik is omdat `select()` wijzigingen aanbrengt in de set die je aan hem meegeeft om aan te kunnen geven van welke sockets gelezen kan worden. Aangezien ik de verbindingen in de gaten

moet houden van de ene `select()` aanroep op de andere, moet ik ze ergens veilig opslaan. Op het laatste moment kopiëer ik de `master` set naar de `read_fds` set, en roep dan `select()` aan.

Maar betekent dat niet dat, telkens als ik een nieuwe verbinding krijg, ik die toe moet voegen aan de `master` set? Juist! En telkens als een verbinding sluit, moet ik hem verwijderen uit de `master` set? Ja, ook dan.

Merk op dat ik controleer of de `listener` klaar is om van gelezen te worden. Wanneer dat het geval is, betekent dat dat er een nieuwe verbinding staat te wachten, en ik `accept()` eer deze en voeg hem toe aan de `master` set. Vergelijkbaar, wanneer een client verbinding klaar is om van gelezen te worden, en `recv()` geeft 0 terug, dan weet ik dat de client de verbinding heeft gesloten, en moet ik deze van de `master` set verwijderen.

Als de client `recv()` een niet-nul teruggeeft, echter, weet ik dat enige data is ontvangen. Dus haal ik het op, en loop dan door de `master` lijst en verstuur die data naar de rest van alle clients.

En dat, vrienden, is een minder-dan-simpele beschouwing van de almachtige `select()` functie.

6.3. Omgaan met partiële `send()`s

Herinner je terug in de sectie over `send()` waar ik zei dat `send()` mogelijk niet alle bytes verstuurt die je had aangegeven? Dat wil zeggen, dat je wil dat er 512 bytes verstuurd worden, maar hij geeft 412 terug. Wat is er met de laatste 100 bytes gebeurd?

Nou, die staan nog steeds in je buffer te wachten om verstuurd te worden. Door omstandigheden buiten jouw controle heeft de kernel besloten om niet alle data in één hap te versturen. En nu is het aan jou om het laatste beetje de deur uit te helpen.

Je zou ook een functie kunnen schrijven die het voor je doet:

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

int sendall(int s, char *buf, int *len)
{
    int totaal = 0;           // aantal verzonden bytes
    int bytesover = *len;    // hoeveel we nog moeten
    int n;

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

```
*len = totaal; // werkelijk verzonden aantal teruggeven  
  
return n==-1?-1:0; // return -1 bij fout, 0 bij succes  
}
```

In dit voorbeeld is *s* de socket waarnaar je de data wilt versturen, *buf* is de buffer met de data, en *len* is een pointer naar een *int* welke het aantal bytes dat in de buffer zit bevat.

De functie geeft *-1* terug bij een fout (en *errno* bevat nog steeds de waarde die door *send()* werd gezet). Ook wordt in *len* het aantal verzonden bytes teruggegeven. Dit zal hetzelfde zijn als het aantal bytes wat je had aangegeven, tenzij er een fout is opgetreden. *sendall()* zal z'n best doen, hijgend en puffend, om de data de deur uit te doen, maar als er een fout optreedt, komt die alsnog terug.

Ten overvloede is hier een voorbeeld van de functieaanroep:

```
char buf[10] = "Beej!";  
int len;  
  
len = strlen(buf);  
if (sendall(s, buf, &len) == -1) {  
    perror("sendall");  
    printf("We hebben slechts %d bytes verstuurd door de fout!\n", len);  
}
```

Wat gebeurt er aan de kant van de ontvanger als er een deel van een pakketje binnenkomt? Als de pakketjes van variabele lengte zijn, hoe weet de ontvanger dan waar het ene pakketje begint en de andere eindigt? Yep, de werkelijkheid doet pijn. Je zult de boel waarschijnlijk moeten *inkapselen* (weet je nog, in de sectie over data inkapsulatie helemaal in het begin?). Lees verder voor de details!

6.4. Data Inkapsulatie

Wat betekent het inkapselen van data nou eigenlijk? In het eenvoudigste geval betekent het dat je een kop (*header*) op de data plakt met of wat indentificerende informatie, of een pakket lengte, of beide.

Hoe zou je header eruit moeten zien? Nou, het is eigenlijk alleen maar wat binaire data die representeert wat jij noodzakelijk acht voor je project.

Wow. Da's vaag.

Okee. Stel bijvoorbeeld dat je een multi-user chat programma hebt dat *SOCK_STREAMS* gebruikt. Wanneer een gebruiker iets typt ("zegt"), zijn er twee stukjes informatie die naar de server verzonden moeten worden: wat er gezegd werd, en door wie.

Tot zover alles nog in orde? "Wat is het probleem?" hoor ik je vragen.

Het probleem is dat de berichten van variabele lengte kunnen zijn. Iemand die "tom" heet zou "Hoi" kunnen zeggen, en iemand anders die "Benjamin" heet zou "Hee jongens, hoe is 't?" kunnen zeggen.

Dus je `send()` dit alles naar de clients zoals het binnenkomt. Je uitgaande datastroom ziet er dan zo uit:

```
t o m H o i B e n j a m i n H e e j o n g e n s , h o e i s ' t ?
```

Enzovoorts. Hoe weet de client waar het ene bericht begint en de andere stopt? Je zou alle berichten even lang kunnen maken en `sendall()` zoals we deze hebben hierboven hebben geïmplementeerd aan kunnen roepen. Maar dan verspil je bandbreedte! We willen geen 1024 bytes `send()` en alleen maar zodat "tom" "Hoi" kan zeggen.

Dus *inkapsuleren* we de data in een kleine header en pakket structuur. Zowel de client als de server weten hoe ze deze data kunnen in- en uitpakken (soms ook wel "marshal" en "unmarshal" genoemd). Geloof het of niet, maar we staan op het punt om een *protocol* te definiëren dat beschrijft hoe een client en server communiceren!

Laten we in dit geval eens aannemen dat de gebruikersnaam een vaste lengte heeft van 8 karakters, opgevuld met '\0'. En laten we aannemen dat de data van variabele lengte is, met een maximum lengte van 128 karakters. Laten we eens kijken naar een voorbeeld pakketstructuur die we in deze situatie zouden kunnen gebruiken:

1. `len` (1 byte, unsigned) – De totale lengte van het pakket, inclusief de 8-byte naam en de chat data.
2. `name` (8 bytes) – De gebruikersnaam, met NULlen opgevuld indien nodig.
3. `chatdata` (*n*-bytes) – De data zelf, niet meer dan 128 bytes. De lengte van het pakket zou dan de lengte van deze data zijn, plus 8 (de lengte van het naamveld).

Waarom ik heb gekozen voor de 8-byte en 128-byte limieten voor de velden? Die heb ik uit de lucht gegrepen, aangenomen dat ze lang genoeg zijn. Misschien is 8 bytes te krap voor je behoeften, en neem je een 30-byte veld voor de naam, maakt niet uit. De keuze is aan jou.

Met bovenstaande pakketdefinitie zou het eerste pakket de volgende informatie bevatten (in hex en ASCII):

```
0B      74 6F 6D 00 00 00 00 00      48 6F 69
(lengte) T o m      (vulling)      H o i
```

Het tweede is vergelijkbaar:

```
14      42 65 6E 6A 61 6D 69 6E      48 65 65 20 6A 6F 6E 67 65 6E ...
(lengte) B e n j a m i n      H e e j o n g e n ...
```

(De lengte wordt in Netwerk Byte Volgorde opgeslagen, uiteraard. In dit geval is het slechts één byte dus maakt het niet uit, maar in het algemeen zul je al je binaire integers in je pakketten op willen slaan in Netwerk Byte Volgorde.)

Wanneer je deze data verstuurt, kun je beter het zekere voor het onzekere nemen en een commando gebruiken dat vergelijkbaar is met `sendall()`, zodat je zeker weet dat alle data is verstuurd, zelfs als het meerdere aanroepen naar `send()` vereist om het allemaal de deur uit te krijgen.

Op vergelijkbare wijze, wanneer je deze data ontvangt, zul je wat extra werk moeten doen. Om zeker te zijn zou je moeten aannemen dat je slechts een deel van een pakketje ontvangt (misschien ontvangen we bijvoorbeeld wel 00 14 42 65 6E van Benjamin, zie boven, als we `recv()` aanroepen). We moeten `recv()` blijven aanroepen tot we het pakketje volledig hebben ontvangen.

Maar hoe? Nou, we weten het aantal bytes dat we in totaal moeten ontvangen om een compleet pakketje te ontvangen, aangezien dat getal aan het begin van het pakketje zit geplakt. We weten ook dat de maximum grootte van het pakket 1+8+128 is, oftewel 137 bytes (want zo hebben we het pakketje gedefinieerd).

Wat je kunt doen is een array declareren die groot genoeg is voor twee pakketjes. In deze werk array reconstrueer je pakketjes zoals ze arriveren.

Elke keer dat je data `recv()` stop je het in de werk array en controleer je of het pakket compleet is. Dat wil zeggen, het aantal bytes in de buffer is groter dan of gelijk aan de lengte gespecificeerd in de header (+1, want de lengte van de header is exclusief de byte voor de lengte zelf). Als het aantal bytes in de buffer minder is dan 1, dan is het pakket duidelijk niet compleet. Hier moet je echter een speciaal geval van maken, aangezien het eerste byte troep is en je er niet op kunt vertrouwen dat het de juiste pakketlengte bevat.

Wanneer het pakket compleet is kun je ermee doen wat je wil. Gebruik het, en verwijder het van je werk buffer.

Whew! Heb je dat in je hoofd zitten? Nou, hier is het tweede punt: het kan zijn dat je het over het einde van het ene pakket en dus een deel van het volgende pakket hebt gelezen in een enkele aanroep van `recv()`. Oftewel, je hebt een werk buffer met een compleet pakket, en een incompleet deel van het volgende pakket! Sodeju. (Maar dit is waarom je je werk buffer groot genoeg hebt gemaakt voor *twee* pakketjes – speciaal voor dit geval!)

Aangezien je de lengte van het eerste pakket kunt vinden in de header, en je het aantal bytes in de werk buffer bijhoudt, kun je door ze van elkaar af te trekken uitrekenen hoeveel bytes in de werk buffer van het tweede (involledige) pakket zijn. Als je het eerste pakket verwerkt hebt, kun je het uit de werk buffer verwijderen het het incomplete pakketje doorschuiven naar het begin van de werk buffer, zodat alles klaar is voor de volgende `recv()` aanroep.

(Een aantal lezers zullen opmerken dat het verplaatsen van het incomplete pakketje naar het begin van de werk buffer tijd kost, en het programma zo geschreven kan worden dat dat niet nodig is door een circulaire buffer te gebruiken. Helaas voor de rest van jullie valt een discussie over circulaire buffers buiten het bestek van dit boek. Als je nog steeds nieuwsgierig bent, pak dan eens een boek over data structuren.)

Ik heb nooit gezegd dat het makkelijk was. Okee, dat heb ik wel gezegd. En dat is het ook; je moet alleen oefenen en voor je het weet gaat het vanzelf. Bij Excalibur, ik zweer het!

7. Meer Referenties

Je bent tot hier gekomen, en nou schreeuw je om meer! Waar kun je nog meer naartoe om meer over dit soort dingen te leren?

7.1. man Pagina's

Probeer om te beginnen eens de volgende man pages:

- `socket()`¹⁹
- `socket options`²⁰
- `bind()`²¹
- `connect()`²²
- `listen()`²³
- `accept()`²⁴
- `send()`²⁵
- `recv()`²⁶
- `sendto()`²⁷
- `recvfrom()`²⁸
- `close()`²⁹
- `shutdown()`³⁰
- `getpeername()`³¹
- `getsockname()`³²
- `gethostbyname()`³³
- `gethostbyaddr()`³⁴
- `getprotobyname()`³⁵
- `fcntl()`³⁶
- `select()`³⁷
- `perror()`³⁸

- `gettimeofday()`³⁹

7.2. Boeken

Voor old-school, echte hou-em-in-je-hand pulp papier boeken zou je de volgende uitstekende handleidingen kunnen proberen. Let op het prominente Amazon.com logo. Wat al dit schaamteloze commercialisme betekent is dat ik wat terug krijg (Amazon.com winkelcrediet, welteverstaan) voor het verkopen van deze boeken via deze handleiding. Dus als je toch al een van de volgende boeken wilde bestellen, waarom zou je me dan geen speciaal bedankje sturen door je geldsmijterij te beginnen via een van de onderstaande links.

Bovendien, meer boeken voor mij zou misschien kunnen leiden tot meer handleidingen voor jou ; -)



Unix Network Programming, volumes 1-2 by W. Richard Stevens. Published by Prentice Hall. ISBNs for volumes 1-2: 013490012X⁴¹, 0130810819⁴².

Internetworking with TCP/IP, volumes I-III by Douglas E. Comer and David L. Stevens. Published by Prentice Hall. ISBN nummers voor delen I, II en III: 0130183806⁴³, 0139738436⁴⁴, 0138487146⁴⁵.

TCP/IP Illustrated, volumes 1-3 by W. Richard Stevens and Gary R. Wright. Published by Addison Wesley. ISBN nummers voor delen 1, 2 en 3: 0201633469⁴⁶, 020163354X⁴⁷, 0201634953⁴⁸.

TCP/IP Network Administration by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 1565923227⁴⁹.

Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by Addison Wesley. ISBN 0201563177⁵⁰.

Using C on the UNIX System by David A. Curry. Published by O'Reilly & Associates, Inc. ISBN 0937175234. *Wordt niet meer gedrukt.*

7.3. Web Referenties

Op het web:

*BSD Sockets: A Quick And Dirty Primer*⁵¹ (heeft ook andere geweldige info over UNIX systeempogrammeren!)

*The Unix Socket FAQ*⁵²

*Client-Server Computing*⁵³

*Intro to TCP/IP*⁵⁴ (gopher)

*Internet Protocol Frequently Asked Questions*⁵⁵

*The Winsock FAQ*⁵⁶

7.4. RFCs

RFCs⁵⁷—the real dirt:

*RFC-768*⁵⁸—The User Datagram Protocol (UDP)

*RFC-791*⁵⁹—The Internet Protocol (IP)

*RFC-793*⁶⁰—The Transmission Control Protocol (TCP)

*RFC-854*⁶¹—The Telnet Protocol

*RFC-951*⁶²—The Bootstrap Protocol (BOOTP)

*RFC-1350*⁶³—The Trivial File Transfer Protocol (TFTP)

8. Gebruikelijke Vragen

Vraag: Waar kan ik die header files krijgen?

Antwoord: Als je ze niet al op je systeem hebt, dan heb je ze waarschijnlijk ook niet nodig. Sla de handleiding voor jouw specifieke platform er even op na. Als je voor Windows aan het schrijven bent, heb je alleen `#include <winsock.h>` nodig.

Vraag: Wat moet ik doen wanneer `bind()` de melding "Address already in use" geeft?

Antwoord: Je moet `setsockopt()` gebruiken met de `SO_REUSEADDR` optie op de luisterende socket. Zie ook de sectie over `bind()` en de sectie over `select()` voor een voorbeeld.

Vraag: Hoe krijg ik een lijst van open sockets op het systeem?

Antwoord: Gebruik het `netstat` commando. Kijk even in de `man` pagina voor volledige details, maar je zou enige nuttige uitvoer moeten krijgen met:

```
$ netstat
```

De truuk is alleen om uit te vogelen welke socket met welk programma is geassocieerd. :-)

Vraag: Hoe kan ik de routing table bekijken?

Antwoord: Voer het **route** commando uit (in `/sbin` op de meeste Linux systemen) of het commando **netstat -r**.

Vraag: Hoe kan ik de client en server programma's uitvoeren als ik maar één computer heb? Moet ik geen netwerk hebben om netwerkprogramma's te schrijven?

Antwoord: Gelukkig voor jou hebben vrijwel alle machines een loopback netwerk "apparaat" dat in de kernel zit en doet alsof het een netwerkkaart is. (Dit is de interface die als `lo` wordt vermeldt in de routing table).

Doe eens even alsof je op een machine ingelogd bent die "geit" heet. Draai de client in een venster en de server in een ander. Of start de server in de achtergrond en draai de client in hetzelfde venster. Het leuke van het loopback device is dat je zowel **client goat** als **client localhost** kunt gebruiken (aangezien "localhost" waarschijnlijk is gedefiniëerd in je `/etc/hosts` bestand) en je client en server babbelen vervolgens lekker met elkaar zonder een netwerk!

Simpel gezegd zijn er geen wijzigingen nodig om de code aan de praat te krijgen op een enkele stand-alone machine! Jippie!

Vraag: Hoe kom ik erachter of de andere kant de verbinding heeft gesloten?

Antwoord: Daar kom je achter doordat `recv()` de waarde 0 teruggeeft.

Vraag: Hoe implementeer ik een "ping" utility? Wat is ICMP? Waar kan ik meer te weten komen over kale sockets (raw sockets) en `SOCK_RAW`?

Antwoord: Al je vragen over kale sockets worden beantwoord in de UNIX Network Programming boeken van W. Richard Stevens. Zie het boeken onderdeel van deze handleiding.

Vraag: Hoe schrijf ik voor Windows?

Antwoord: Ten eerste, verwijder Windows en installeer Linux of BSD. } ; -). Nee, je hoeft alleen maar naar de sectie over schrijven voor Windows in de introductie te gaan.

Vraag: Hoe compileer ik voor Solaris/SunOS? Ik blijf maar linker errors krijgen wanneer ik probeer te compileren!

Antwoord: De linker errors komen omdat Sun dozen niet automatisch de socket libraries mee compileren. Zie de sectie over compileren voor Solaris/SunOS in de introductie voor een voorbeeld.

Vraag: Waarom kapt `select()` er steeds mee bij ontvangst van een signaal (*signal*)?

Antwoord: Signals hebben de neiging om geblokkeerde systeemaanroepen `-1` terug te laten geven, met `errno` ingesteld op `EINTR`. Wanneer je een signal handler instelt met de `sigaction()` kun je de vlag op `SA_RESTART` zetten, wat ervoor zou moeten zorgen dat de systeemaanroep wordt herstart nadat het werd onderbroken.

Natuurlijk werkt dit niet altijd.

Mijn favoriete oplossing hiervoor maakt gebruik van een `goto` statement. Je weet dat dit je leraren mateloos irriteert, dus ga ervoor!

```
select_restart:
    if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
        if (errno == EINTR) {
            // een of ander signaal heeft ons onderbroken, dus herstarten
            goto select_restart;
        }
        // echte errors hier afhandelen:
        perror("select");
    }
```

Tuurlijk, je *hoeft* `goto` in dit geval niet te gebruiken; je kunt ook andere structuren gebruiken. Maar ik vind dat het gebruik van `goto` netter is.

Vraag: Hoe implementeer ik een timeout op een aanroep naar `recv()`?

Antwoord: Gebruik `select()`! Het stelt je in staat een timeout parameter aan te geven voor socket descriptors waarvan je wilt lezen. Je zou al die functionaliteit ook in een enkele functie kunnen stoppen, zo:

```
#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;
```

```
// file descriptor set opstellen
FD_ZERO(&fds);
FD_SET(s, &fds);

// de struct timeval voor de timeout instellen
tv.tv_sec = timeout;
tv.tv_usec = 0;

// wacht op timeout of ontvangst data
n = select(s+1, &fds, NULL, NULL, &tv);
if (n == 0) return -2; // timeout!
if (n == -1) return -1; // error

// er moet data zijn, dus voer een normale recv() uit
return recv(s, buf, len, 0);
}

// Voorbeelaanroep van recvtimeout():
.
.
n = recvtimeout(s, buf, sizeof(buf), 10); // 10 seconden timeout

if (n == -1) {
    // fout opgetreden
    perror("recvtimeout");
}
else if (n == -2) {
    // timeout opgetreden
} else {
    // wat data in buf ontvangen
}
.
.
```

Merk op dat `recvtimeout()` in geval van een timeout `-2` teruggeeft. Waarom geven we niet `0` terug? Nou, zoals je je misschien herinnert, een return waarde van `0` bij een aanroep naar `recv()` geeft aan dat de andere kant de verbinding heeft gesloten. Dus die return waarde heeft al een betekenis, en `-1` betekent "fout", dus heb ik `-2` als timeout indicator gekozen.

Vraag: Hoe encrypt of comprimeer ik de data voor ik het over een socket stuur?

Antwoord: Een makkelijke manier om encryptie te gebruiken is SSL (secure socket layer), maar dat ligt buiten het bestek van deze handleiding.

Maar aangenomen dat je je eigen compressor- of encryptiesysteem wil implementeren, is het het makkelijkst om je data te zien als lopend door een serie stappen tussen beide uiteinden. Elke stap verandert de data op een of andere manier.

1. de server leest data uit een bestand (of wat dan ook)
2. de server encrypt de data (dit voeg je zelf toe)
3. de server `send()` de versleutelde data

Nu andersom:

4. de client `recv()` de versleutelde data
5. de client decrypt de data (dit voeg je zelf toe)
6. de client schrijft de data naar een bestand (of wat dan ook)

Op het punt waar je encrypt/decrypt zou je ook kunnen comprimeren. Of allebei! Denk er alleen wel aan dat je comprimeert voor je encrypt. :)

Zolang de client maar netjes de stappen van de server ongedaan maakt, blijft de data prima, hoeveel tussenstappen je ook toevoegd.

Dus het enige wat je moet doen om m'n code te gebruiken is de plaats vinden waar de data gelezen wordt en verstuurd wordt (gebruik makend van `send()` over het netwerk, en prop er wat code tussen die de encryptie doet.

Vraag: Wat is die "PF_INET" die maar op blijft duiken? Is het gerelateerd aan AF_INET?

Antwoord: Ja, inderdaad. Zie het onderdeel over `socket()` voor details.

Vraag: Hoe kan ik een server schrijven die shell commando's accepteert van een client en die uitvoert?

Antwoord: Om het simpel te houden, laten we zeggen dat de client de verbinding `connect()`, `send()` en `close()` t (dat wil zeggen, er worden geen verdere systeemaanroepen uitgevoerd zonder dat de client opnieuw verbinding maakt).

Het proces dat de client volgt is dit:

1. `connect()` en met de server
2. `send("/sbin/ls > /tmp/client.out")`
3. `close()` de verbinding

Ondertussen verwerkt de server de data is voert deze uit:

1. `accept()` de verbinding van de client
2. `recv(str)` de commando string
3. `close()` de verbinding
4. `system(str)` om het commando uit te voeren

Waarschuwing! Het draaien van een server die uitvoert wat de client zegt is hetzelfde als het uitvoeren van externe toegang (shell access) en mensen kunnen dingen aan je account veranderen wanneer ze verbinding maken met de server. Bijvoorbeeld, in het bovenstaande voorbeeld, wat als de client "**rm -rf ~**" verstuurt? Het verwijdert alle bestanden in je account!

Dus je denkt slim te zijn en voorkomt dat de client alles behalve een paar utilities niet meer kan gebruiken waarvan je weet dat ze veilig zijn, zoals het **foobar** utility:

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

Maar je bent nog steeds niet veilig, helaas: wat als de client "**foobar ; rm -rf ~**" invoert? Het veiligste wat je kan doen is een kleine routine te schrijven die een escape ("\") karakter voor alle niet-alfanumerieke karakters plaatst (inclusief spaties, indien van toepassing) in de argumenten voor het commando.

Zoals je ziet is security een flink onderwerp wanneer de server dingen gaat uitvoeren die de client verstuurt.

Vraag: Ik stuur een sloot aan data, maar wanneer ik `recv()`, wordt er maar 536 of 1460 bytes aan data peer keer ontvangen. Maar als ik het op m'n lokale machine draai, wordt alle data in één keer ontvangen. Wat is er aan de hand?

Antwoord: Je loopt tegen de MTU aan—het maximum aantal bytes wat het fysieke medium aankan. Op de lokale machine gebruik je het loopback apparaat welke zonder probleem 8K of meer aankan. Maar op ethernet, wat maar

1500 bytes met een header aankan, loop je tegen dat limiet aan. Over een modem, met 576 MTU (weer inclusief header) loop je tegen het nog lagere limiet aan.

Ten eerste moet je ervoor zorgen dat alle data verstuurd wordt (zie de `sendall()` functie implementatie voor details). Wanneer je daar zeker van bent, moet je `recv()` in een loop (lus) aanroepen totdat al de data gelezen is.

Lees het onderdeel over Data Enkapsulatie voor details over het ontvangen van complete pakketten met gebruik van meerdere aanroepen naar `recv()`.

Vraag: Ik zit op een Windows doos en ik heb geen `fork()` systeemaanroep of een of andere vorm van `struct sigaction`. Wat nu?

Antwoord: Als ze al ergens zijn, dan zitten ze in POSIX libraries welke bij je compiler zouden kunnen zitten. Aangezien ik geen Windows doos heb kan ik het je niet vertellen, maar ik meen me te herinneren dat Microsoft een POSIX compatibiliteitslaag heeft, en daar zou `fork()` tussen moeten zitten (en misschien zelf `struct sigaction`.
doorzoek de help documentatie die bij VC++ zit op "fork" of "POSIX". Misschien dat dat je wat oplevert.

Als dat nou helemaal niet werkt, gooi dan de hele `fork()/struct sigaction` zooi maar weg en vervang het door het win32 equivalent: `CreateProcess()`. Ik weet niet hoe `CreateProcess()` gebruikt moet worden—het vereist een baziljoen argumenten, maar dat zou in de documentatie moeten staan die bij VC++ zit.

Vraag: Hoe verstuur ik data veilig via TCP/IP met gebruik van encryptie?

Antwoord: Werp eens een blik op het OpenSSL project⁶⁴.

Vraag: Ik zit achter een firewall—hoe vertel ik andere mensen buiten de firewall m'n IP adres zodat ze verbinding kunnen maken met m'n machine?

Antwoord: Helaas is het doel van een firewall voorkomen dat mensen van buiten de firewall verbinding kunnen maken met machines binnen de firewall, dus juist het toestaan daarvan wordt gezien als een veiligheidsbreuk.

Dit wil niet zeggen dat alles verloren is. Bijvoorbeeld, je kunt vaak wel `connect()` en door de firewall naar buiten als het een of andere vorm van masquerading of NAT doet. Ontwerp je programma's zó dat jij altijd degene bent die de verbinding initialiseert, en alles komt goed.

Als je daar niet tevreden mee bent kun je je systeembeheerders vragen of ze een gaatje in de firewall willen prikken zodat mensen verbinding met je machine kunnen maken. De firewall de verbinding naar jou doorsturen hetzij via z'n NAT software, of door een proxy of iets dergelijks.

Let erop dat een gat in de firewall niet iets is om lichtzinnig over te doen. Je moet er voor zorgen dat je niet de verkeerde mensen toegang tot je interne netwerk geeft; als je een beginner bent, is het een stuk moeilijker om software veilig te maken dan je zou denken.

Maak je systeembeheerder niet kwaad op me ; -)

9. Disclaimer en een Hulpoproep

Nou, dat is het wel zo'n beetje. Hopelijk was in elk geval een deel van de informatie in dit document een beetje nauwkeurig en ik hoop echt dat er geen enorme fouten in zitten. Maar die zijn er natuurlijk altijd.

Dus laat dit een waarschuwing voor je zijn! Het spijt me indien een van de onnauwkeurigheden in dit document je enig leed hebben bezorgd, maar je kunt me gewoon niet verantwoordelijk houden. Zie je, ik sta achter geen enkel woord in dit document, juridisch gezien. Het hele verhaal zou compleet en volledig verkeerd kunnen zijn!

Maar waarschijnlijk is het dat niet. Uiteindelijk heb ik vele, vele uren zitten prutsen met deze stof, en verscheidene TCP/IP netwerk programmaatjes geschreven op m'n werk, en multiplayer game engines geschreven, enzovoorts. Maar ik ben niet de god van de sockets; ik ben zomaar iemand.

Overigens, als iemand enige constructieve (of destructieve) kritiek heeft over dit document, stuur dan een mailtje naar <beej@piratehaven.org> en ik zal kijken of ik het een en ander recht kan zetten.

Mocht je je afvragen waarom ik dit heb gedaan, nou ja, voor het geld. Ha! Nee, ik heb het gedaan omdat een hoop mensen me socket gerelateerde vragen stelden, en wanneer ik ze vertelde dat ik zat te denken om een socket pagina te maken zeiden ze, "Cool!" Bovendien, ik heb het gevoel dat al deze zuurverdiende kennis verloren gaat als ik het niet deel met anderen. Het web blijkt gewoon met perfecte medium. Ik moedig anderen aan om vergelijkbare informatie aan te bieden wanneer het maar kan.

Genoeg hierover-verder programmeren! ; -)

Noten

1. <http://www.ecst.csuchico.edu/~beej/guide/net/>
2. <http://tangentsoft.net/wskfaq/>
3. <http://www.tuxedo.org/~esr/faqs/smart-questions.html>
4. <http://analyser.oli.tudelft.nl/beej/>
5. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/send.2.inc>
6. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recv.2.inc>
7. <http://www.rfc-editor.org/rfc/rfc793.txt>

8. <http://www.rfc-editor.org/rfc/rfc791.txt>
9. <http://www.rfc-editor.org/rfc/rfc768.txt>
10. <http://www.rfc-editor.org/rfc/rfc791.txt>
11. <http://www.rfc-editor.org/rfc/rfc1413.txt>
12. `getip.c`
13. `server.c`
14. `client.c`
15. `listener.c`
16. `talker.c`
17. `select.c`
18. `selectserver.c`
19. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/socket.2.inc>
20. <http://linux.com.hk/man/showman.cgi?manpath=/man/man7/socket.7.inc>
21. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/bind.2.inc>
22. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/connect.2.inc>
23. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/listen.2.inc>
24. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/accept.2.inc>
25. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/send.2.inc>
26. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recv.2.inc>
27. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/sendto.2.inc>
28. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recvfrom.2.inc>
29. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/close.2.inc>
30. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/shutdown.2.inc>
31. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getpeername.2.inc>
32. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getsockname.2.inc>
33. <http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyname.3.inc>
34. <http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyaddr.3.inc>

35. <http://linux.com.hk/man/showman.cgi?manpath=/man/man3/getprotobyname.3.inc>
36. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/fcntl.2.inc>
37. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/select.2.inc>
38. <http://linux.com.hk/man/showman.cgi?manpath=/man/man3/perror.3.inc>
39. <http://linux.com.hk/man/showman.cgi?manpath=/man/man2/gettimeofday.2.inc>
40. <http://www.amazon.com/exec/obidos/redirect-home/beejsguides-20>
41. <http://www.amazon.com/exec/obidos/ASIN/013490012X/beejsguides-20>
42. <http://www.amazon.com/exec/obidos/ASIN/0130810819/beejsguides-20>
43. <http://www.amazon.com/exec/obidos/ASIN/0130183806/beejsguides-20>
44. <http://www.amazon.com/exec/obidos/ASIN/0139738436/beejsguides-20>
45. <http://www.amazon.com/exec/obidos/ASIN/0138487146/beejsguides-20>
46. <http://www.amazon.com/exec/obidos/ASIN/0201633469/beejsguides-20>
47. <http://www.amazon.com/exec/obidos/ASIN/020163354X/beejsguides-20>
48. <http://www.amazon.com/exec/obidos/ASIN/0201634953/beejsguides-20>
49. <http://www.amazon.com/exec/obidos/ASIN/1565923227/beejsguides-20>
50. <http://www.amazon.com/exec/obidos/ASIN/0201563177/beejsguides-20>
51. <http://www.cs.umn.edu/~bentlema/unix/>
52. <http://www.ibrado.com/sock-faq/>
53. <http://pandonia.canberra.edu.au/ClientServer/>
54. gopher://gopher-chem.ucdavis.edu/11/Index/Internet_aw/Intro_the_Internet/intro.to.ip/
55. <http://www-iso8859-5.stack.net/pages/faqs/tcpip/tcpipfaq.html>
56. <http://tangentsoft.net/wskfaq/>
57. <http://www.rfc-editor.org/>
58. <http://www.rfc-editor.org/rfc/rfc768.txt>
59. <http://www.rfc-editor.org/rfc/rfc791.txt>
60. <http://www.rfc-editor.org/rfc/rfc793.txt>
61. <http://www.rfc-editor.org/rfc/rfc854.txt>

62. <http://www.rfc-editor.org/rfc/rfc951.txt>
63. <http://www.rfc-editor.org/rfc/rfc1350.txt>
64. <http://www.openssl.org/>