Beej’s Guide to Network Programming

Używanie gniazd internetowych

Brian "Beej" Hall

beej@piratehaven.org

Bartosz Zapałowski

bartek@klepisko.eu.org

Historia zmian

Zmiana Wersja 1.0.0 Sierpień, 1995 Revised by: beej
Pierwsza wersja.
Zmiana Wersja 1.5.5 13 Styczeń, 1999 Revised by: beej
Ostatnia wersja HTML.
Zmiana Wersja 2.0.0 6 Marzec, 2001 Revised by: beej
Przekonwertowane do formatu DocBook XML, poprawki, dodatki.
Zmiana Wersja 2.3.1 8 Październik, 2001 Revised by: beej
Poprawione literówki, błędy składniowe w client.c, dodane parę rzeczy w seckji FAQ.

Spis treści

1. Wprowadzenie

Cześć! Programowanie gniazd zwala Cię z nóg? Czy jest to zbyt trudne do nauczenia ze stron **man**a? Chcesz pisać programy Internetowe, ale nie masz czasu na szperanie pomiędzy `struct`urami aby się dowiedzieć czy najpierw muszisz użyć `bind()`a zanim się połączysz (`connect()`) itp.

Hmm, zgadnij co! Właśnie wykonałem tą brudną robotę i umieram z chęci podzielenia się tą informacją ze wszystkimi. Jesteś w dobrym miejscu. Ten dokument powinien dać przeciętnemu programiście C informacji, których potrzebuje aby uporać się z sieciowym brudem.

1.1. Dla kogo?

Ten dokument został napisany jako przewodnik, nie jako źródło informacji. Jest chyba najlepszy dla osób, które dopiero zaczynają programowanie gniazd i szukają wsparcia. Z całą pewnością nie jest to *kompletny* przewodnik do programowania gniazd.

Pełen nadzieji myślę, że będzie to wystarczająca dla podręczników mana, by nabrały one sensu... `:-)`

1.2. Platforma i kompilator

Kod zawarty w tym dokumencie był kompilowany na Linux PC używając kompilatora GNU **gcc**. Jednak powinien się on skompilować na każdej platformie używającej **gcc**. Oczywiście nie dotyczy to sytuacji, gdy programujesz w Windows -- przeczytaj sekcję dla programistów Windows.

1.3. Oficjalna strona główna

Oficjalna strona główna tego dokumentu w oryginale mieści się na serwerze California State University, Chicho pod adresem `http://www.ecst.csuchico.edu/~beej/guide/net/`.

1.4. Uwagi dla programistów Solaris/SunOS

Kompilując dla Solaris/SunOS, muszisz zdefiniować kilka dodatkowych przełączników w lini komend aby zlinkować dla poprawnych bibliotek. Aby to zrobić po prostu dodaj "`-lnsl -lsocket -lresolv`" na końcu polecenia kompilacji, np.:

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

Jeśli nadal otrzymujesz błędy spróbuj jeszcze dodać "`-lxnet`" na końcu tego polecenia. Nie wiem co to daje dokładnie, ale niektórzy ludzie tego potrzebują.

Kolejnym miejscem sprawiającym problemy jest wywołanie `setsockopt()`. Prototyp tej funkcji różni się od tej na moim Linuksie, więc zamiast:

```
int yes=1;
```

wpisz to:

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

Ponieważ nie mam systemu SunOS, więc nie testowałem żadnej z powyższyły informacji -- to jest tylko to, co ludzie mi powiedzieli poprzez emaila.

1.5. Uwagi dla programistów Windows

Nie za bardzo lubię Windows - namawiam Cię do spróbowania systemu Linux, BSD lub Unix. Jednak możesz używać informacji zawartych w tym dokumencie również pod Windows.

Po pierwsze - zignoruj sporą ilość wszystkich nagłówków systemowych, których tu użyłem. Wszystko, co musisz dołączyć to:

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

Czekaj! Musisz również zrobić wywołanie do funkcji `WSAStartup()` zanim cokolwiek będziesz chciał zrobić przy użyciu biblioteki gniazd. Kod do tego celu wygląda mniej więcej tak:

```
#include <winsock.h>

{
    WSADATA wsaData;   // jeśli to nie zadziała
    //WSAData wsaData; // użyj tego

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

Musisz również powiedzieć kompilatorowi, żeby zlinkował program z biblioteką Winsock, zazwyczaj nazywanej `wsock32.lib` lub `winsock32.lib` lub coś w tym stylu. Używając Visual C++, powyższa czynność może być wykonana poprzez menu `Project` pod opcją `Settings`. Kliknij zakładkę `Link` i odszukaj "Object/library modules". Dodaj "wsock32.lib" do tej listy.

W końcu, musisz wywołać `WSACleanup()`, gdy już skończysz pracę z biblioteką gniazd. Zobacz pomoc on-line, jeśli chcesz znać szczegóły.

Jak już to zrobisz, reszta przykładów w tym przewodniku powinna działać z pewnymi wyjątkami. Po pierwsze - nie możesz używać `close()` do zamykania gniazd--zamiast tego musisz użyć `closesocket()`. Również, `select()` działa tylko z deskryptorami gniazd, nie deskryptorami plików (takich jak `0` dla `stdin`).

Jest również klasa gniazd, której możesz użyć - `CSocket`. Sprawdź strony pomocy swojego kompilatora, by uzyskać więcej informacji.

Aby uzyskać więcej informacji o Winsock, przeczytaj Winsock FAQ[2].

W końcu, słyszałem, że Windows nie ma funkcji `fork()`, która jest używana w kilku przykładach. Może musisz zlinkować z biblioteką POSIX lub czymś, żeby to działało. Możesz też użyć `CreateProccess()` w zamian.

`fork()` nie bierze żadnych argumentów, a `CreateProcess()` wymaga około 48 bilionów argumentów ;-). Jeśli nie potrzebujesz takiej funkcji, `CreateThread()` jest trochę prostsza do strawienia... Niestety, dyskusja o wielowątkowości wychodzi poza ramy tego dokumentu. Witaj we wspaniałym świecie programowania pod Win32.

1.6. Zanim do mnie napiszesz

Generalnie jestem gotowy by odpowiadać na pytania w emailach, więc możesz spokojnie pisać, ale nie mogę zagwarantować odpowiedzi. Prowadzę bardzo zajęte życie i są takie chwile, że po prostu nie mogę udzielić odpowiedzi. Kiedy tak się dzieje, zazwyczaj usuwam wiadomość. Nie bierz tego do siebie; Po prostu wiem, że nie będę miał czasu, by udzielić Ci szczegółowej odpowiedzi.

Z reguły im bardziej skomplikowanie jest pytanie tym mniejsze szanse na to, że będę w stanie odpowiedzieć. Jeśli dokładnie napiszesz o co Ci chodzi oraz załączysz ważne informacje (takie jak platforma, kompilator, błędy jakie otrzymujesz i wszystko inne co mogłoby mi pomóc w znalezieniu rozwiązania) wtedy masz większe szanse na otrzymanie odpowiedzi. Po więcej szczegółów przeczytaj dokument ESR, Jak mądrze zadawać pytania[3].

Jeśli jednak jej nie otrzymasz, skup się nad swoim problemem, spróbuj sam uzyskać odpowiedź, i jeśli nadal problem będzie występował napisz ponownie dołączając informacje, które uzyskałeś i miejmy nadzieję, że to mi wystarczy by Ci pomóc.

Teraz jak Cie nauczyłem jak do mnie pisać (;-)), chciałbym powiedzieć, że z *pełnym* szacunkiem odnoszę się do wartości jaką uzyskał ten przewodnik w ciągu lat. To jest prawdziwy kop moralny, i cieszy mnie to, że jest on uważany za dobry! :) Dziękuję!

1.7. Tworzenie mirrorów strony

Możesz śmiało mirrorować tą stronę, dla celów publicznych lub prywatnych. Jeśli mirrorujesz dla celów publicznych i chcesz, żebym umieścił link to tej strony na stronie głównej, napisz do mnie <`beej@piratehaven.org`>.

1.8. Uwagi dla tłumaczy

Jeśli chcesz przetłumaczyć ten przewodnik na inny język, napisz do mnie <`beej@piratehaven.org`> a ja zrobię link na stronie głównej do twojego tłumaczenia.

Nie krępuj się by dodać swoje nazwisko i adres email do tłumaczenia.

Przykro mi, ale z powodu przestrzeni dyskowej, nie mogę hostować tłumaczeń.

1.9. Prawa autorskie i dystrybucja

Nauczyciele są specjalnie namawiani do rekomendowania lub dostarczania kopi tego przewodnika swoim studentow.

Ten przewodnik może być dowolnie przetłumaczony na inne języki, pod warunkiem, że tłumaczenie będzie dokładne, a przewodnik przedrukowany w całości. Tłumaczenie może również zawierać nazwisko i informacje kontaktowe tłumacza.

Źródła kodu w C przedstawione w tym dokumencie są oddane dla wszystkich.

Pisz <`beej@piratehaven.org`> po więcej informacji.

2. Co to jest gniazdo?

Słyszysz cały czas gatkę o "gniazdach", i zapewne zastanawiasz się co one tak właściwie znaczą. Więc, są one sposobem rozmowy z innymi programami używając Unixowych deksryptorów plików.

Co?

Dobra, mogłeś słyszeć kilku hackerów Unixa mówiących "Wooooooow, *wszystko* w Uniksie jest plikiem!". To o czym mówiła ta osoba jest fakt, że kiedy Uniksowe programy wykonują operacje I/O, wykonują je poprzez czytanie lub zapisywanie do dekryptora pliku. Deksryptor pliku jest po prostu liczbą (integer) przypisaną do otwartego pliku. Ale (i tu jest haczyk), ten plik może być połączeniem sieciowym, FIFO, pipem, terminalem, lub prawdziwym plikiem na dysku, lub wszystkim innym. W Uniksie wszystko *jest* plikiem! Więc gdy chcesz komunikować się z innym programem poprzez Internet zrobisz to za pomocą deskryptora pliku, lepiej uwież w to.

"Skąd wziąć ten desktyptor pliku dla komunikacji sieciowej, Panie Mądralo?" jest pewnie ostatnim pytaniem chodzącym Ci teraz po głowie, ale zamierzam na nie odpowiedzieć właśnie teraz: Wywołujesz funckje `socket()`. Zwraca ona deskryptor gniazda, i komunikujesz się używając specjalnej funkcji `send()` oraz `recv()` (**man send**[4], **man recv**[5]).

"Ale skoro jest to deksryptor pliku, to dlaczego nie mogę używać po prostu `read()` i `write()` do komunikacji przez gniazdo?" Krótką odpowiedzią jest "Możesz!". Dłuższą jest "Możesz, ale `send()` i `recv()` dają Ci większą kontrolę nad transmisją danych."

Co dalej? Może to: jest kilka rodzajów gniazd. Są: Internetowe adresy DARPA (gniazda internetowe), ścieżki na lokalnym systemie (gniazda Unixowe), adresy CCITT X.25 (gniazda X.25, które możesz normalnie spokojnie zignorować), i prawdopodobnie inne... Ten dokument "bawi się" tylko tymi pierwszymi: gniazdami Internetowymi.

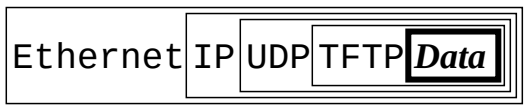
2.1. Dwa typy gniazd internetowych

Co to jest? Są dwa typy gniazd internetowych? Tak. No, nie. Kłamię. Jest ich więcej, ale nie chcę was przestraszyć. Zamierzam tylko powiedzieć, że "surowe gniazda" są również potężne i powinieneś się nimi zainteresować.

No dobra, co to za dwa typy gniazd internetowych? Jeden to "gniazdo strumieniowe"; drugi "gniazdo datagramowe", których nazwami są odpowiednio "`SOCK_STREAM`" i "`SOCK_DGRAM`". Gniazda datagramowe są czasami nazywane "gniazdami bezpołączeniowymi". (Mimo, że mogą one być połączone za pomocą `connect()` jeśli naprawdę tego chcesz. Spójrz na `connect()` poniżej.)

Gniazda strumieniowe są godnym zaufania dwukierunkowym połączeniem strumieniowym. Jeśli wyślesz dwa znaki do gniazda w kolejności "1, 2" dotrą one w kolejności "1, 2" na drugim końcu kabla. Będą również bezbłędne. Wszelkie błędy, których doświadczysz są Twoimi wyimaginowanymi błędami i nie będą tutaj poruszane.

Rysunek 1. Enkapsulacja danych.



Przez co są używane gniazda strumieniowe? Mogłeś słyszeć o programie telnet... On używa gniazd strumieniowych. Wszystkie znaki, które wpisujesz muszą dotrzeć w tej samej kolejności w jakiej je wpisywałeś, prawda? Przeglądarki internetowe używają protokołu HTTP, który to korzysta z gniazd strumieniowych aby pobrać strony. Możesz to sprawdzić używając telneta podając jako adres adres strony i port 80 i wpisując "`GET /`" - zwróci to HTMLa strony głównej.

Jak gniazda strumieniowe osiągają tak wysoki poziom jakości transmisji danych? Używają protokołu zwanego "The Transmission Control Protocol" (protokół kontroli transmisji), zwanego również w skrócie "TCP" (patrz RFC-793[6] po bardzo szczegółowy opis TCP). TCP martwi się, żeby dane docierały sekwencyjnie i bezbłędne. Prawdopodobnie słyszałes "TCP" wcześniej jako lepszą połowę "TCP/IP", gdzie "IP" to "Internet Protocol" (protokół Internetowy) (patrz RFC-791[7]). IP zajmuje się głównie routingiem i nie jest generalnie odpowiedzialne za integralność danych.

Fajnie. A co z gniazdami datagramowymi? Dlaczego są nazywane bezpołączeniowymi? Dlaczego nie są godne zaufania? Masz tu kilka faktów: jeśli wysyłasz datagram, może on dotrzeć. Może też dotrzeć za późno (jeśli w ogóle dotrze). Jeśli dotrze, dane zawarte w pakiecie będą bezbłędne.

Gniazda strumieniowe również korzystają z IP dla routingu, ale nie wykorzystują TCP; wykorzystują "User Datagram Protocol" (datagramowy protokół użytkownika?) - w skrócie "UDP" (patrz RFC-768[8]).

Dlaczego są bezpołączeniowe? Dlatego, że nie musisz utrzymywać otwartego połączenia tak jak to robisz w przypadku gniazd strumieniowych. Po prostu budujesz pakiet, doklejasz nagłówek IP z informacją o celu i wysyłasz. Połączenie nie jest potrzebne. Zazwyczaj są wykorzystywane do transmisji informacji pakiet-za-pakietem. Prosty przykład: **tftp**, **bootp**,itp.

"Wystarczy!" możesz krzyknąć. "Jak w takim razie te programy mogą działać, skoro datagramy mogą się zgubić?!" No cóż, mój ludzki kumplu, każdy z nich ma swój protokół nad UDP. Na przykład protokół tftp mówi, że dla każdego wysłanego pakietu, odbiorca musi odesłać pakiet mówiący "mam go!" (pakiet "ACK"). Jeśli nadawca nie otrzymuje odpowiedzi w ciągu, powiedzmy pięciu sekund, ponownie wysyła pakiet aż w końcu otrzyma ACK. Taka procedura informowania jest bardzo ważna w przypadku implementowania programów korzystających z `SOCK_DGRAM`.

2.2. Niskopoziomowy nonsense a teoria sieci

Po wspomnieniu wartsw protokołów, przyszedł czas na powiedzenie jak sieci naprawdę działają, i jak pakiety `SOCK_DGRAM` są zbudowane. Praktycznie możesz ominąć tą sekcję. Jednakże jest to dobra podstawa.

Hej dzieciaki, przyszedł czas na naukę o *enkapsulacji danych*! Jest to bardzo ważne. Jest to tak ważne, żeby możesz się tego uczyć jeśli będziesz uczęszczał na kurs sieciowy na Chico State `;-)`. A więc: pakiet się rodzi, pakiet jest owijany (przetwarzany) w nagłównku (czasami w stopce) przez pierwszy protokół (powiedzmy TFTP), następnie całość (nagłówek TFTP) jest przetwarzana przez następny protokół (powiedzmy UDP), potem znowu to samo tyle, że przez następny protokół (IP), i znowu przez ostatni protokół na warstwie sieciowej (fizycznej) (powiedzmy Ethernet).

Gdy inny komputer odbiera pakiet, sprzęt wycina nagłówek Ethernet, jądro wycina nagłówki IP i UDP, program TFTP wycina nagłówek TFTP, i w końcu ma dane.

Mogę w końcu powiedzieć o niesławnym *"Layered Network Model"* (wasrtwowy model sieci). Ten model opisuje system funkcjonowania sieci, który ma wiele zalet w stosunku do innych modeli. Na przydkład, możesz pisać program wykorzystujące gniazda, które są zawsze takie same (programy) bez zamartwiania się jak dane są fizycznie przesyłane (serial, Ethernet, AUI, cokolwiek), ponieważ programy na niższych poziomach zajmują się tym za Ciebie. Prawdziwe urządzenia sieciowe jak i topologia sieci są niewidoczne dla programisty gniazd.

Bez żadnego dalszego głędzenia zaprezentuję warsty tego modelu sieci:

- Aplikacja
- Presentacja
- Sesja
- Transport
- Sieć
- Łącze danych
- Fizyczna

Warstwa fizyczna jest sprzętem (serial, Ethernet...). Warstwa aplikacji jest tak daleka od warstwy fizycznej jak tylko to sobie możesz wyobrazić--jest to miejsce, gdzie użytkownicy działają przez sieć.

Ten model jest tak uogólniony, że prawdopodobnie mógłbyś go użyć do naprawy swojego samochodu gdybyś tylko naprawdę tego chciał. Warstwowy model bardziej związany z Uniksem mógłby wyglądać taj:

- Warstwa aolikacji (*telnet, ftp, ipt.*)
- Warstwa transportu host-host (*TCP, UDP*)
- Warstwa internetowa (*IP i routing*)
- Warstwa dostępu do sieci (*Ethernet, ATM, lub cokolwiek*)

W tym momencie możesz zobaczyć jak te warstwy współpracują ze sobą aby przetworzyć dane.

Widzisz ile jest roboty przy budowaniu prostego pakietu? Jeejku! I ty to wszystko musisz wpisać do nagłówka pakietu używając "**cat**"! To był żart. Wszystko co musisz zrobić przy gniazdach strumieniowych to wysłać (`send()`) dane. Wszystko co musisz zrobić przy gniazdach datagramowych to enkapsulować pakiet wybraną metodą i wysłać go (`sendto()`). Jądro zbuduję warstwę transportową i internetową za ciebie, a sprzęt doda warstwę dostępu do sieci. Ach, dzisiejsza technologia.

I tak się kończy nasze krótkie wprowadzenie w teorię sieci. Aha, zapomniałem Ci powiedzieć wszystkiego co chciałem powiedzieć o rutingu: nic! Dokładnie, nie zamierzam o tym mówić. Ruter wycina pakiet z nagłówka IP,

sprawdza tabelę rutingu, bla bla bla. Sprawdź IP RFC[9] jeśli Ci naprawdę na tym zależy. Nawet jeśli się tego nie nauczysz, przeżyjesz.

3. `structury` i przetwarzanie danych

No, w końcu tu jesteśmy. Pora, żebym powiedział trochę o programowaniu. W tej sekcji odryję przed Tobą różne typy danych używanych przez interfejsy gniazd, ponieważ niektóre z nim są bardzo trudne do rozszyfrowania.

Na początek coś łatwego: deskryptor gniazda. Deskryptor gniazda jest następującego typu:

```
int
```

Po prostu normalny `int`.

Teraz zaczną się pojawiać dziwne rzeczy, więc przeczytaj to i zgódź się ze mną. Pamiętaj: są dwa sposoby reprezentacji bajtu: na początku najbardziej znaczący bit (czasami zwany oktetem) lub najmniej znaczący bit na początku. Ta pierwsza reprezentacja jest nazywana sieciową kolejnością bajtów, a druga kolejnością bajtów hosta. Niektóre maszyny przechowują liczby używają sieciowej kolejności bajtów, niektóre nie. Kiedy mówię, że coś ma być w sieciowej kolejności bajtów, wtedy musisz wywołać specjalną funkcję (np. `htons()`), aby zamienić liczbę z kolejności bajtów hosta. Jeśli nie powiem, że liczba musi być w sieciowej kolejności bajtów, wtedy musisz zostawić ją w kolejności bajtów hosta.

(Dla ciekawych, "sieciowa kolejność bajtów" jest również znana jako "Big-Endian Byte Order".)

Moja Pierwsza Struktura™ -- `struct sockaddr`. Ta struktura przechowuje informacje o adresie gniazda dla różnych typów gniazd:

```
struct sockaddr {
    unsigned short    sa_family;    // rodzina adresów, AF_xxx
    char              sa_data[14];  // 14-bajtowy adres protokołowy
};
```

`sa_family` może być różnymi rzeczami, ale w tym dokumencie będzie tylko `AF_INET`. *`sa_data`* przechowuje docelowy adres i numer portu dla gniazda. Jednak nie będziesz musiał pakować ręcznie adres w *`sa_data`*.

Żeby się uporać z `struct sockaddr` programiści stworzyli równorzędną strukturę: `struct sockaddr_in` ("in" jak Internet).

```
struct sockaddr_in {
    short int          sin_family;  // rodzina adresów
    unsigned short int sin_port;    // numer portu
    struct in_addr     sin_addr;    // adres internetowy
    unsigned char      sin_zero[8]; // dla zachowania rozmiaru struct sockaddr
};
```

Ta struktura ułatwia dostęp do elementów adresu gniazda. Zauważ, że *`sin_zero`*(które jest dołączone do struktury dla wyrównania wielkości struktury z `struct sockaddr`) powinno być ustawione na zero (wszystkie elementy) przez funkcję `memset()`. Również *ważną* informacją jest to, że wskaźnik na `struct sockaddr_in` może być

zrzutowany na `struct sockaddr` i vice-versa. Więc mimo, że `socket()` chce `struct sockaddr*`, możesz używać `struct sockaddr_in` i rzutować to w ostatniem chwili! Zauważ też, że *`sin_family`* odpowiada *`sa_family`* w `struct sockaddr` i powinno być ustawione na "`AF_INET`". W końcu *`sin_port`* i *`sin_addr`* muszą być w *sieciowej kolejności bajtów*!

"Ale," możesz się zastanawiać, "jak cała struktura, `struct in_addr sin_addr`, może być w Network Byte Order?". To pytanie wymaga uważnenego przyjrzenia się strukturze `struct in_addr`:

```
// adres internetowy (struktura z przyczyn historycznych)
struct in_addr {
    unsigned long s_addr; // to ma rozmiar 32 bitów, lub 4 bajtów
};
```

Kiedyś to *było* unią (union), ale teraz już się tego nie spotyka. Dobre posunięcie. Więc jeśli zadeklarowałeś *`ina`* jako `struct sockaddr_in`, wtedy *`na.sin_addr.s_addr`* odpowiada 4 bajtowemu adresowi IP (w sieciowej kolejności bajtów). Zauważ, że nawet gdy twój system nadal używa okropnej unii (union) dla `struct in_addr` nadal możesz się odwoływać do 4 bajtowego adresu IP w identyczny sposób jaki czyniliśmy wcześniej (a to z podowu `#define`ów).

3.1. Zmiana reprezentacji bajtów

W końcu trafiliśmy do następnej sekcji. Do tej pory było dużo gadania o konwersji z Network na Host Byte Order -- teraz przyszedł czas na akcję!

Jołkej. Są dwa typy, które możesz konwertować: `short` (dwa bajty) i `long` (cztery bajty). Poniższe funkcje działają równie dobrze dla wariacji `unsigned`. Powiedzmy, że chcesz skonwertować `short` z kolejności bajtów hosta na sieciową kolejność bajtów. Zacznij z "h" dla "Host", następnie "to", w końcu "n" dla "Network" i "s" dla "short": h-to-n-s, lub `htons()` (czyt. "Host to Network Short").

Jest to prawie za łatwe...

Możesz używać każdej kombinacji "n", "h", "s" i "l" , jakiej tylko chcesz (oprócz tych głupich). Na przykład, nie ma funkcji `stolh()` ("Short to Long Host"), ale jest:

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

Now, you may think you're wising up to this. Pewnie myślisz "Co mam zrobić, jeśli muszę zmienić kolejność bajtów w `char`?". Wtedy możesz sobie pomyśleć "Eee, nie ważne". Możesz również pomyśleć, że skoro twoja maszyna 68000 już używa Network Byte Order, nie musisz wywoływać `htonl()` na adresach IP. Miałbyś rację. *ALE* jeśli spróbujesz otworzyć port na maszynie, która używa odwrotnej kolejności niż Network Byte Order, twój program nie zadziała poprawnie. Pisz przenośnie! To jest świat Unix! (tak bardzo jak Bill Gates chciałby, żeby było odwrotne). Pamiętaj: Spraw, by twoje bajty były w Network Byte Order zanim umieścisz je w sieci.

Ostatni punkt: dlaczego *`sin_addr`* oraz *`sin_port`* muszą być w Network Byte Order w strukturze `struct sockaddr_in`, podczas gdy *`sin_family`* nie? Odpowiedź: *`sin_addr`* oraz *`sin_port`* są kapsułkowane w pakiecie w warstwach odpowiednio IP i UDP. Stąd muszą być w Network Byte Order. Jakkolwiek, pole *`sin_family`* jest używane tylko przez jądro do określenia jaki typ adresu przechowuje struktura, więc musi być w Host Byte Order. Również, ponieważ *`sin_family`* *nie* jest wysyłane do sieci, może być w Host Byte Order.

3.2. Adresy IP oraz jak sobie z nimi radzić

Na szczęście jest kilka funkcji, która pozwalają na manipulowanie adresami IP. Nie musisz ich rozpracowywać i składować w `long` za pomocą operatora `<<`.

Na początek powiedzmy, że mamy strukturę `struct sockaddr_in ina` oraz adres IP "`10.12.110.57`", który chcesz przechowywać w niej. Funkcją, której chcesz użyć jest `inet_addr()` (od tłumacza: ta funkcja jest przestarzała, użyj `inet_pton()`), która zamienia adres IP na notację numerowo-kropkową i składuje go w `unsigned long`. Przypisanie adresu IP wygląda tak:

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

Zauważ, że `inet_addr()` zwraca adres w Network Byte Order -- nie musisz wywoływać `htonl()`. Fajnie!

Powyższy kawałek kodu nie jest zbyt dobry, ponieważ nie ma tam sprawdzania błędów. Zauważ, że `inet_addr()` zwraca `-1` w przypadku błędu. Pamiętasz liczby binarne? `(unsigned)-1` dziwnym trafem odpowiada adresowi IP `255.255.255.255`! To jest adres rozgłoszeniowy! Niedobrze. Pamiętaj by sprawdzać błędy poprawnie.

Właściwie, to jest lepszy interface, którego możesz użyć zamiast `inet_addr()`: jest to `inet_aton()` ("aton" oznacza "ascii to network" (zamień ascii na adres sieciowy)):

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

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

Poniżej jest przykładowe użycia wstawiania danych do `struct sockaddr_in` (ten przykład będzie dla ciebie bardziej sensowny, gdy dojdziesz do sekcji o `bind()` oraz `connect()`).

```
struct sockaddr_in my_addr;

my_addr.sin_family = AF_INET;         // host byte order
my_addr.sin_port = htons(MYPORT);     // short, network byte order
inet_aton("10.12.110.57", &(my_addr.sin_addr));
memset(&(my_addr.sin_zero), '\0', 8); // wyzeruj resztę struktury
```

`inet_aton()`, *w przeciwieństwie do praktycznie każdej funkcji operującej na gniazdach*, zwraca wartość niezerową w przypadku powodzenia oraz zero w przypadku błędu. (Jeśli ktoś wie dlaczego, proszę niech mi powie.) Adres IP jest umieszczany w zmiennej *`inp`*.

Niestety, nie wszystkie platformy dostarczają `inet_aton()` dlatego, mimo, że jest ona preferowana, starsza i częściej używana funkcje `inet_addr()` jest używana w tym przewodniku.

Tak więc, teraz umiesz zamieniać adres IP zapisane jako tekst w jego binarny odpowiednik. Co powiesz na proces odwrotny? Co, jeśli miałbyś `struct in_addr` i chciałbyś wyświetlić to w formacie liczb rozdzielonych kropkami? W tym przypadku użyłbyś funkcji `inet_ntoa()` ("ntoa" oznacza "network to ascii" - zamień adres sieciowy na ciąg znaków ascii) (od tłumacza: zamiast `inet_ntoa()` polecam `inet_ntop()`) tak jak tu:

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

Powyższe wyświetli adres IP. Zauważ, że `inet_ntoa()` przyjmuje `struct in_addr` jako swój argument, a nie `long`. Zauważ również, że funkcja ta zwraca wskaźnik do typu char. Ten sposób działania funkcji sprawia, że wynik jej działania jest umieszczany w statycznie alokowanym buforze w obrębie funkcji `inet_ntoa()` , tak więc za każdym wywołaniem funkcji `inet_ntoa()` nadpisze ona ostani adres IP, o który pytałeś. Przykład:

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

Powyższe da wynik:

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

Jeśli musisz zachować adres, użyj `strcpy()`, by wynik skopiować do własnego bufora.

To tyle na ten temat póki co. Dalej nauczysz się jak zamieniać tekst, taki jak "whitehouse.gov" na jego odpowiedni adres IP (patrz DNS, poniżej.)

4. Wywołania systemowe

W tej sekcji przyjrzymy się bliżej wywołaniom systemowym, które dają dostęp do funkcjonalności sieci na maszynie Uniksowej. Gdy wywołujesz jedną z tych funkcji, jądro przejmuje wywołanie i załatwia całą sprawę automagicznie.

Miejscem, w którym większość ludzi się zatrzymuje jest prawidłowa kolejność wywoływania tych funkcji. W tej sytuacji podręcznik **man** nie jest zbyt pomocny, co już zapewne mogłeś zauważyć. Żeby pomóc ci w tej okropnej sytuacji, starałem się umieścić wywołania systemowe w kolejnych sekcjach w *takiej samej* kolejności, w jakiej będziesz je wywoływał w swoich programach.

Powyższe, połączone z wieloma przykładami kodu umieszczonymi tu i tam, z niewielką ilością mleka i ciasteczek (w które mam nadzieję już się zaopatrzyłeś), jak również odwagą, będziesz przesyłał dane przez Internet jak Son of Jon Postel (od tłumacza: niech to ktoś ładnie przetłumaczy!).

4.1. `socket()` -- Daj deskryptor pliku!

Zapewne nie mogę już dłużej tego odkładać - muszę powiedzieć o wywołaniu systemowym `socket()`. Oto jej prototyp:

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

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

Ale co to za argumenty? Pierwszy, *`domena`* powinien być ustawiony na "`AF_INET`", tak jak w `struct sockaddr_in` powyżej (od tłumacza: obecnie za właściwe uznaje się wstawianie tutaj wartości `PF_INET`). Następny parametr, *`typ`*, mówi jądru jaki jest to typ gniazda: `SOCK_STREAM` lub `SOCK_DGRAM`. W końcu, *`protokol`* ustaw na `0`, żeby pozwolić funkcji `socket()` wybrać właściwy protokół bazując na parametrze *`typ`*. (Uwagi: jest dużo więcej typów, które mogą być wstawione jako *`domena`* niż tutaj napisałem. Masz również większy wybór co do parametru *`type`* niż tutaj napisałem. Zobacz stronę podręcznika systemowego funkcji `socket()`. Jest również dużo lepszy sposób na ustalenie parametru *`protokol`*. Zobacz stronę podręcznika systemowego funkcji `getprotobyname()`.)

Funkcja `socket()` zwraca ci po prostu deskryptor gniazda, który możesz później użyć w innych wywołaniach systemowych, lub `-1` w przypadku błędu. W drugim przypadku zmienna globalna *`errno`* jest ustawiana na wartość błędu (patrz stronę podręcznika systemowego funkcji `perror()`).

W niektórych dokumentacjach zobaczysz wzmiankę o tajemniczym "`PF_INET`. Jest to dziwna, rzadko spotykana bestia, ale może trochę rozjaśnię ją trochę. Dawno, dawno temu, myślano, że prawdopodobnie rodzina adresów (za tym terminem stoi "AF" w "`AF_INET`") może obsługiwać wiele protokołów, do których odwoływano się za pomocą ich rodziny protokołów (za tym terminem stoi właśnie to "PF" w "`PF_INET`"). Jednak tak się nie stało. Jednakże prawidłową rzeczą jest użycie `AF_INET` w `struct sockaddr_in` oraz `PF_INET` w wywołaniu `socket()`. Jednak w praktyce możesz używać `AF_INET` gdziekolwiek. I ponieważ robi to W. Richard Stevens w swojej książce, tak też będę ja to robił tutaj. (od tłumacza: nie idź tropem Stevensa w tym przypadku, bo wtedy skażesz się na ewentualne godziny pracy potrzebne do zamiany `AF_INET` na `PF_INET` w odpowiednich miejscach, gdy tylko ich definicja się zmieni. Pamiętaj też o przenośności - to co jest prawdą pod Linuksem nie musi nią być pod *BSD.)

Dobra, dobra, ale co jest dobrego w gnieździe? Tak naprawdę gniazdo samo w sobie nie jest zbyt dobre. Musisz jeszcze czytać dalej i użyć kolejnych wywołań systemowych, żeby nabrało to sensu.

4.2. `bind()` -- Na jakim jestem porcie?

Jak już masz gniazdo, możesz potrzebować je przypisać pewnemu portowi na maszynie lokalnej. (jest to często wykonywane, jeśli zamierzasz wywoływać `listen()` dla umożliwienia przychodzenia połączeń na danym porcie -- MUDy często to robią, kiedy ci mówią "zatelnetuj się na x.y.z na port 6969".) Numer portu jest używany przez jądro dla określenia, który pakiet wchodzący powinien pójść do którego deskryptoru gniazda. Jeśli zamierzasz wywoływać tylko `connect()`, ta funkcja może być zbędna. Mimo to, przeczytaj ten rozdział, przynajmniej tak dla szpanu.

Oto prototyp dla funkcji `bind()`:

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

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

`sockfd` to deskryptor gniazda zwrócony przez `socket()`. *`my_addr`* to wskaźnik na `struct sockaddr`, który zawiera informacje o twoim adresie, a dokładniej porcie i adresie IP. *`addrlen`* może być ustawiony na wartość `sizeof(struct sockaddr)`.

Ugh. To trochę dużo do przyswojenia w tak krótkim czasie. Weźmy przykład:

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

#define MYPORT 3490

main()
{
    int sockfd;
    struct sockaddr_in my_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0); // zrób sprawdzanie błędów!

    my_addr.sin_family = AF_INET;         // host byte order
    my_addr.sin_port = htons(MYPORT);     // short, network byte order
    my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");
    memset(&(my_addr.sin_zero), '\0', 8); // wyzeruj resztę struktury

    // don't forget your error checking for bind():
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
    .
    .
    .
```

Jest tu kilka rzeczy wartych uwagi: *`my_addr.sin_port`* jest w Network Byte Order, tak jak *`my_addr.sin_addr.s_addr`*. Następną warto uwagi rzeczą jest to, że dla włączane pliki nagłówkowe mogą się różnić w zależności od systemu. Żeby być pewnym, powinieneś sprawdzić lokalne strony podręcznika systemowego **man**.

W końcu na temat `bind()`a mogę powiedzieć, że część procesu pobierania własnego adresu IP i/lub portu może być zautomatyzowane:

```
    my_addr.sin_port = 0; // wybierz losowo nieużywany port
    my_addr.sin_addr.s_addr = INADDR_ANY;  // użyj mojego adresu IP
```

Jak widzisz, ustawiając *`my_addr.sin_port`* na zero, pozwalasz funkcji `bind()` wybrać port za ciebie. Podobnie, ustawiając *`my_addr.sin_addr.s_addr`* na `INADDR_ANY`, pozwalasz automatycznie wypełnić adres IP maszyny, na której wykonuje się proces.

Jeśli byłeś wystarczająco uważny, zapewne rzucił ci się w oczy fakt, że nie umieściłem `INADDR_ANY` w Network Byte Order! Ale jestem niedobry, co? Jednak, mam coś na swoje usprawiedliwienie: `INADDR_ANY` jest tak naprawdę zerem. Zero jest nadal zerem nawet jak przestawisz kolejność bitów. Jednakże, puryści językowi zauważą, że `INADDR_ANY` na różnych maszynach może przyjmować różne wartości, takie jak np. 12, i że mój kod nie będzie tam działał. Nie stanowi to dla mnie problemu:

```
    my_addr.sin_port = htons(0); // wybierz losowo nieużywany port
```

```
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);  // żyj mojego adresu IP
```

No, teraz jesteśmy tak przenośni, że pewnie nie możesz w to uwierzyć. Chciałem tylko zwrócić uwagę na fakt, że w większości kawałkach kodu nie spotkasz się z sytuacją owijania `INADDR_ANY` w funkcję `htonl()`. (od tłumacza: ten pogląd skontrastuję z bardzo ciekawym zdaniem: "Jedzcie gówno - miliony much nie mogą się mylić".)

`bind()` również zwraca `-1` w przypadku błędu i ustawia odpowiednio *`errno`* na wartość błędu.

Inną rzeczą, wartą zauważenia jest to, żebyś przy wywoływaniu `bind()` nie schodził za bardzo z numerem portu. Wszystkie porty poniżej 1024 są ZAREZERWOWANE (chyba, że jesteś superużytkownikiem)! Możesz używać tylko portów powyżej tej liczby, aż do 65535 (zakładając, że nie są one już zajęte przez inny program).

Czasami możesz zauważyć, że przy próbie ponownego uruchomienia serwera `bind()` zwraca błąd stwierdzając "Podany adres jest już w użyciu" ("Address already in use.") Co to oznacza? Część gniazda, która była połączona ciągle się szlaja w jądrze i zajmuje ten port. Możesz poczekać, żeby zasoby się zwolniły (minutę lub trochę dłużej), albo dodać kawałek kodu do twojego programu, sprawiając, że ten ponownie wykorzysta dany port. Przykład:

```
    int yes=1;
 //char yes='1'; // programiści Solaris używają tego

    // zgub denerwujący komunikat błędu "Address already in use"
    if (setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
        perror("setsockopt");
        exit(1);
    }
```

Jeszcze jedna mała uwaga na koniec o `bind()`: są sytuacje, w których kompletnie nie będziesz potrzebował tej funkcji wywoływać. Jeśli łączysz się ze zdalnym serwerem (`connect()`) i nie obchodzi cię z jakiego portu będziesz nadawał (tak jak to ma miejsce w programie **telnet**, gdzie dbasz tylko o numer portu zdalnego), możesz po prostu wywołać `connect()`. Ta funkcja sama sprawdzi, czy gniazdo nie jest przypisane, i użyje nieużywanego portu jeśli potrzeba.

4.3. `connect()` -- Ej, ty!

Poudawajny przez chwilę, że jesteś programem telnet. Twój użytkownik rozkazuje ci (dokładnie tak samo jak w filmie *TRON*), żebyś wziął deksryptor gniazda. Spełniasz prośbę i wywołujesz `socket()`. Następnie, użytkownik mówi ci, żebyś nawiązał połączenie z "`10.12.110.57`" na porcie "`23` (jest to standardowo port telneta). I co teraz robisz?

Na szczęście dla ciebie, programie, właśnie dokładnie przeglądasz sekcję o `connect()` -- jak połączyćsię ze zdalnym hostem. Więc czytaj dalej! Nie ma czasu do stracenia!

Funkcja `connect()` ma następujący prototyp:

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

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

`sockfd` to nasz przyjazny sąsiad - deksryptor gniazda zwrócony przez wywołanie `socket()`. *`serv_addr`* to `struct sockaddr` zawierający docelowy port i adres IP, a *`addrlen`* może mieć wartość `sizeof(struct sockaddr)`.

Czyż nie zaczyna to nabierać sensu? Pozwól, że przytoczę następujący przykład:

```
#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;   // tu będzie przechowywany adres docelowy

    sockfd = socket(AF_INET, SOCK_STREAM, 0); // zrób sprawdzanie błędów!

    dest_addr.sin_family = AF_INET;          // host byte order
    dest_addr.sin_port = htons(DEST_PORT);   // short, network byte order
    dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
    memset(&(dest_addr.sin_zero), '\0', 8);  // wyzeruj resztę struktury

    // nie zapomnij o sprawdzeniu błędów dla connect()!
    connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
    .
    .
    .
```

Ponownie, pamiętaj o sprawdzeniu wartości zwracanej przez `connect()` -- zwróci ona `-1` w przypadku błędu i odpowienio ustawi zmienną *`errno`*.

Zauważ również, że nie wywoływaliśmy funkcji `bind()`. Po prostu nie musimy się troszczyć o numer portu lokalnego: obchodzi nas tylko to gdzie idziemy (zdalny port). Jądro wybierza za nas port lokalny, a strona, z którą się łączymy pobierze tą informację automatycznie. Bez obaw.

4.4. `listen()` -- Proszę dzwonić...

No dobra, czas odwórcić role. Co jeśli nie chcesz się łączyć z innymi hostami. Powiedzmy, że tak dla jaj, chcesz czekać na połączenia przychodzące i obsługiwać je w jakiś sposób. Ten proces jest dwustopniowy: najpierw nasłuchujesz (`listen()`, następnie przyjmujesz połączenia (`accept()`) (patrz niżej.)

Ropoczęcie nasłuchiwania jest bardzo proste, jednak wymagą odrobiny tłumaczenia:

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

`sockfd` jak zwykle jest deksryptorem gniazda zwróconym przez funkcję `socket()`. *`backlog`* jest ilością dozwolonych połączeń w kolejce połączeń przychodzących. Co to oznacza? Połączenia przychodzące będą czekały

w tej kolejce, dopóiki każdego z osobna nie zaakceptujesz (`accept()`), a *`backlog`* określa ile może być oczekujących połączeń. Większość systemów po cichu ogranicza tą liczbę do 20. Prawdopodobnie i tak będziesz ustawiał tą wartość na `5` lub `10`.

Jak zwykle, `listen()` zwraca `-1` i ustawia *`errno`* w przypadku błędu.

Tak jak się zapewne domyślasz, musimy wywołać `bind()` przed `listen()` - w przeciwnym razie jądro wybierze dla nas losowy port. Fuj! Więc jeśli zamierzasz nasłuchiwać na połączenia przychodzące, prawidłową kolejnością wywołań systemowych jest:

```
socket();
bind();
listen();
/* tutaj accept() wkracza do akcji */
```

Powyższe pozostawiam jako przykładowy kod źródłowy, ponieważ mówi on sam za siebie (kod z sekcji o `accept()`, poniżej, jest bardziej kompletny). Pewne trudności z całego tego bałagnu może sprawiać jedynie wywołanie `accept()`.

4.5. `accept()` -- "Dziękujemy za wybranie portu 3490."

Przygotuj się -- wywołanie `accept()` jest trochę dziwne! Oto co się zaraz stanie: ktoś z bardzo daleka spróbuje połączyć się (`connect()`) z twoją maszyną na port, na którym nasłuchujesz (`listen()`). Jego połączenie zostanie umieszone w kolejci i bedzie czekało na zaakceptowanie. Wywołujesz `accept()` i mówisz jądru, żeby przyjąć jedno połączenie oczekujące. Zostanie ci zwrócony *całkiem nowy* deskryptor gniazda dla tego jednego połączenia! Zgadza się, nagle masz *dwa deskryptory gniazd* za cenę jednego! Ten oryginalny ciągle nasłuchuje na twoim porcie, a nowoutworzony jest w końcu gotowy do wysyłania (`send()`) i odbierania dancyh (`recv()`). W końcu tu dotarliśmy!

Wywołanie jest następujące:

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

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

`sockfd` jest deskryptorem nasłuchującego gniazda. Całkiem proste, nie? *`addr`* jest zazwyczaj wskaźnikiem do lokalnej struktury `struct sockaddr_in`. To właśnie tutaj się znajdą informacje o oczekującym połączeniu (i dzięki temu możesz stwierdzić, który host się do ciebie dobija i z jakiego portu). *`addrlen`* jest lokalną zmienną całkowitą, która powinna być ustawiona na `sizeof(struct sockaddr_in)` zanim adres jest przekazany do `accept()`. Ta funkcja nie umieści więcej bajtów danych w *`addr`* niż określa to zmienna *`addrlen`*. Jeśli umieści mniej bajtów, zmienna *`addrlen`* zostanie odpowiednio zmieniowa, tak by odpowiadała rozmiarowi struktury.

Zganij co? `accept()` zwraca `-1` i ustawia zmienną *`errno`*, jeśli wystąpi błąd. Założę się, że nie wieziałeś.

Tak jak przedtem, jest tego trochę dużo do wchłonięcia na raz, więc teraz pokaże przykładowy kawałek kodu, żebyś to lepiej zrowumiał:

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

```
#define MYPORT 3490    // port, z którym użytkownicy będą się łączyli

#define BACKLOG 10     // jak dużo może być połączeń oczekujących

main()
{
    int sockfd, new_fd;  // nasłuchuj na sock_fd, nowe połączenia na new_fd
    struct sockaddr_in my_addr;    // informacja o moim adresie
    struct sockaddr_in their_addr; // informacja o adresie osoby łączącej się
    int sin_size;

    sockfd = socket(AF_INET, SOCK_STREAM, 0); // zrób sprawdzanie błędów!

    my_addr.sin_family = AF_INET;         // host byte order
    my_addr.sin_port = htons(MYPORT);     // short, network byte order
    my_addr.sin_addr.s_addr = INADDR_ANY; // uzupełnij moim adresem IP
    memset(&(my_addr.sin_zero), '\0', 8); // wyzeruj resztę struktury

    // nie zapomnij o sprawdzeniu błędów dla poniższych wywołań!
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

    listen(sockfd, BACKLOG);

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

Ponownie zauważ, że będziemy używali deksryptora gniazda *`new_fd`* dla wszystkich funkcji `send()` oraz `recv()`. Jeśli chcesz odebrać tylko jedno połączenie, możesz zamknąć (`close()`) nasłuchujące gniazdo *`sockfd`*, żeby zapobiec kolejnym połączeniom przychodzącym na ten sam port, jeśli sobie tylko tego życzysz.

4.6. `send()` i `recv()` -- Mów do mnie, Misiu!

Te dwie funkcje służa do komunikacji przez gniazda strumieniowe lub połączone gniazda datagramowe. Jeśli chcesz użyć normalnych, niepołączonych gniazd datagramowych, będzie musiał się bliżej przyjżeć sekcji o `sendto()` i `recvfrom()` poniżej.

Wywołanie `sendto()`:

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

`sockfd` jest deskryptorem gniazda, do którego chcesz wysłać dane (niezależnie od tego czy jest to gniazdo zwrócone przez `socket()` czy też `accept()`). *`msg`* jest wskaźnikiem do danych, które chcesz wysłać, a *`len`* jest długością tych danych w bajtach. *`flags`* po prostu ustaw na `0` (zobacz stronę podręcznika systemowego o `send()`, jeśli chcesz wiedzieć więcej o flagach).

Przykładowym kodem może być:

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

`send()` zwraca ilość bajtów, które udało się wysłać -- *może to być mniejsza wartość od tej, którą podałeś funkcji!* No bo widzisz, czasami chcesz wysłać ogromną ilość danych i jądro po prostu nie może sobie z tym poradzić. Spróbuje wysłać najwięcej ile może, ufając, że wyślesz resztę później. Pamiętaj, jeśli wartosć zwrócona przez `send()` nie zgadza się z wartością znajdującą się w zmiennej *`len`*, wszystko zależy od ciebie, czy wyślesz resztę danych. Dobra wiadomość: jeśli pakiet jest mały (mniej niż 1KB) *prawdopodobnie* zostanie on wysłany w całości. Znowu, `-1` jest zwracane w przypadku wystąpienia błedu, a zmienna *`errno`* jest ustawiana na wartość tego błedu.

Wywołanie `recv()` jest podobne pod wieloma aspektami:

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

`sockfd` jest deksryptorem gniazda, z którego dane maja być odczytane. *`buf`* jest buforem, w którym zostaną umieszczone odczytane dane o długości podanej za pomocą parametru *`len`* -- jest to maksymalna długość bufora. *`flags`* znowu może być ustawiona na `0` (zobacz stronę podręcznika systemowego o `recv()`, żeby uzyskac informacje o flagach).

`recv()` zwraca ilość odczytanych danych, lub `-1` w przypadku błędu (i odpowiednio ustawia *`errno`*).

Ale czekaj! `recv()` może zwrócić `0`. To może oznaczać tylko jedno: druga strona zamknęła połączenie! Zwrócona wartość `0` jest sposobem mówienia `recv()`, że to właśnie się stało.

To było łatwe, prawda? Możesz teraz wysyłać i odbierać dane używając gniazd strumieniowych! Faaajnie! Jesteś Uniksowym Programistą Sieciowym!

4.7. `sendto()` i `recvfrom()` -- Mów do mnie w stylu DGRAM

To wszystko jest cacy," już słyszę jak mówicie, "ale co mam zrobić z niepołączonymi gniazdami datagramowymi?" No problemo, amigo. Jesteś we właściwym miejscu.

Poniważ gniazda datagramowe nie są połączone ze zdalnym hostem, zgadnij jaką informację musimy podać zanim możemy wysłać pakiet? Dokładnie! Adres docelowy! Oto przepis:

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

Jak już pewnie zauważyłeś, to wywołanie jest praktycznie takie samo jak wywołanie `send()` z dwoma dodatakowymi parametrami. *`to`* jest wskaźnikiem na strukturę `struct sockaddr` (którą i tak pewnie będziesz przechowywał jako `struct sockaddr_in` i wykonywał odpowiednie rzutowanie w ostatniej chwili), któr zawiera docelowy adres IP i port. *`tolen`* może być po prostu ustawione na `sizeof(struct sockaddr)`.

Tak jak to było z `send()`, `sendto()` zwraca ilość wysłanych danych (ta ilość może być inna od tej, którą podałeś!), lub `-1` w przypadku błędu.

Odpowiednio podobne są `recv()` i `recvfrom()`. Wywołanie `recvfrom()` jest następujące:

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

Ponownie, jest to praktycznie to samo co `recv()` z dodatkiem pewnych pozycji. *`from`* jest wskaźnikiem do lokalnej struktury `struct sockaddr`, która będzie wypełniona adresem IP i portem maszyny, od której pochodzi pakiet. *`fromlen`* jest wskaźnikiem do lokalnej zmiennej typu `int`, która przed wywołaniem tej funkcji powinna być ustawiona na `sizeof(struct sockaddr)`. Gdy funkcja powróci, *`fromlen`* będzie zawierało długość adresu przechowywanego w *`from`*.

`recvfrom()` zwraca ilość odebranych danych lub `-1` w przypadku błędu (ustawiając odpowiednio zmienną *`errno`*).

Pamiętaj, że jesli połączysz (`connect()`) gniazdo datagramowe, możesz używać po prostu funkcji `send()` i `recv()` dla wszystkich operacji. Gniazdo samo w sobie jest nadal gniazdem datagramowych, a pakiety nadal korzystają z protokołu UDP, ale interfejs gniazda automatycznie doda adres docelowy i informacje o źródle za ciebie.

4.8. `close()` i `shutdown()` -- Wynocha!

Ale jazda! Wysyłałeś (`send()`) i odbierałeś (`recv()`) dane przez cały dzień i już masz dość. Jesteś gotowy, by zamknąć połączenie na twoim deksryptorze gniazda. To jest calkiem proste. Możesz po prostu użyć standardowej funkcji Uniksowej do zamykania deskryptorów plików - funkcji `close()`:

```
close(sockfd);
```

To zapobiegnie jakimkolwiek dalszym odczytom bądź zapisom do tego gniazda. Ktokolwiek próbujący odczytać bądź zapisać do gniazda po drugiej stronie otrzyma błąd.

W przypadku gdybyś chciał mieć większą kontrolę nad zamykaniem gniazda, możesz użyć funkcji `shutdown()`. Ta funkcja pozwala ci uciąc połączenie w określonym kierunku lub w obu kierunkach (tak jak to robi `close()`). Wywołanie:

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

`sockfd` jest deksryptorem gniazda, które chcesz zamknąć, a *`how`* jest jedną z poniższych wartości (od tłumacza: dla zachowania przenośności używaj stałych podanych w nawiasach zamiast podanych tu wartości):

- `0` (`SHUT_RD`)-- dalszy odbiór danych jest zabroniony
- `1` (`SHUT_WR`) -- dalszy zapis danych jest zabroniony
- `2` (`SHUT_RDWR`) -- dalszy odbiór i zapis danych jest zabroniony (tak jak przy `close()`)

`shutdown()` zwraca `0`, gdy wywołane się powiedzie lub `-1` w przypadku błędu (ustawiając odpowiednio zmienną *`errno`*).

Jeśli użyjesz `shutdown()` na niepołączonym gnieździe datagramowym, sprawisz, że gniazdo będzie niedostępne dla dalszym wywołań `send()` i `recv()` (pamiętaj, że możesz ich używać, jeśli połączysz (`connect()`) twoje gniazdo datagramowe).

Należy tu zauważyć, że `shutdown()` nie zamyka deskryptora pliku, ale zmienia jego stan używalności. Żeby zwolnić deksryptor gniazda, musisz użyć `close()`.

Nic więcej.

4.9. `getpeername()` -- Kim jesteś?

Ta funkcja jest baardzo prosta.

Jest tak prosta, że prawie nie dałem jej własnej sekcji. Ale jednak ją ma.

Funkcja `getpeername()` powie ci, kto jest na drugim końcu połączonego gniazda strumieniowego. Wywołanie:

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

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

`sockfd` jest deskryptorem połączonego gniazda strumieniowego, *`addr`* jest wskaźnikeim do struktury `struct sockaddr` (lub `struct sockaddr_in`), która będzie przechowywała informacje o drugiej stronie połączenia, a *`addrlen`* jest wskaźnikeim do zmiennej typu `int`, która przed wywołaniem funkcji powinna mieć wartość `sizeof(struct sockaddr)`.

Funkcja zwraca `-1` w przypadku błędu i odpowiednio ustawia zmienną *`errno`*.

Jak już masz ten adres, możesz użyć funkcji `inet_ntoa()` lub `gethostbyaddr()`, by wyświetlić lub pobrać więcej informacji. Nie, nie możesz pobrać nazwy użytkownika (no dobra, jeśli na tamtym komputerze działa demon ident, to jest możliwe. Jednak to wykracza poza ramy tego dokumentu. Patrz RFC-1413[10] po więcej informacji).

4.10. `gethostname()` -- Kim ja jestem?

Jeszcze prostszą funckją od `getpeername()` jest funckja `gethostname()`. Zwraca ona nazwę komputera, na którym działa twój program. Ta nazwa może być później użyta przez funkcję `gethostbyname()`, poniżej, w celu ustalenia adresu IP twojej maszyny lokalnej.

Co mogłoby nam sprawić większą przyjemność? Przychodzi mi na myśl wiele rzeczy, ale nie mają one związku z programowaniem gniazd. W każdym razie oto wywołanie funkcji:

```
#include <unistd.h>

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

Argumenty są proste: *`hostname`* jest wskaźnikiem do ciągu znaków, w którym będzie przechowywana nazwa hosta przy powrocie z funkcji, a *`size`* jest długością w bajtach danych zawartych w ciągu znaków *`hostname`*.

Funkcja zwraca `0` przy pomyślnym zakończeniu oraz `-1` w przypadku błedu (jak zwykle odpowiednio ustawiając zmienną *`errno`*).

4.11. DNS -- Mówisz "whitehouse.gov", odpowiadam "198.137.240.92"

W przypadku gdybyś nie wiedział co to jest DNS - jest to "Domain Name Service" Krótko mówiąc, mówisz mu jaki jest czytelny dla człowieka adres dla danego serwisu, a on ci poda jego adres IP (którego możesz używać z `bind()`, `connect()`, `sendto()` lub czegokolwiek innego, do czego jest ci to potrzebne). Tym sposobem, gdy ktoś wpisze:

```
$ telnet whitehouse.gov
```

telnet wie, że potrzebuje połączyć się z adresem "198.137.240.92".

Ale jak to działa? Będziesz używał funkcji `gethostbyname()`:

```
#include <netdb.h>

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

Jak widzisz, ta funkcja zwraca wskaźnik do struktury `struct hostent`, której zawartość jest następująca:

```
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]
```

A tu są opisy poszczególnych pól w `struct hostent`:

- *`h_name`* -- oficjalna nazwa hosta
- *`h_aliases`* -- ciąg znaków, zakończony znakiem NULL, zawierający alternatywne nazwy dla hosta
- *`h_addrtype`* -- typ zwróconego adresu - zazwyczaj *`AF_INET`*.
- *`h_length`* -- długosc adresu w bajtach
- *`h_addr_list`* -- ciąg znaków, zakończony znakiem NULL, określający adresy sieciowe dla hosta. Adresy hosta są w sieciowej kolejności bajtów.
- *`h_addr`* -- pierwszy adres z *`h_addr_list`*.

`gethostbyname()` zwraca wskaźnik do wypełnionej struktury `struct hostent`, lub NULL w przypadku błędu (w tym przypadku *`errno`* *nie* jest ustawiane -- *`h_errno`* jest ustawiana w zamian. Zobacz `herror()`, poniżej.)

Ale jak się tego używa? Czasami (czego możemy się dowiedzieć czytając podręczniki komputerowe), zasypanie czytelnika informacja nie wystarcza. Ta funkcja jest z pewnością łatwiejsza w użytcie niż się wydaje.

Tu jest przykładowy program[11]:

```
/*
** getip.c -- jak rozwiązywać nazwy hostów
*/

#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) {  // sprawdź poprawność lini poleceń!
        fprintf(stderr,"usage: getip address\n");
        exit(1);
    }

    if ((h=gethostbyname(argv[1])) == NULL) {  // pobierz informacje o hoście
        herror("gethostbyname");
        exit(1);
    }

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

   return 0;
}
```

Korzystając z `gethostbyname()` nie możesz używać `perror()` do wyświetlania komunikatów o błędach (ponieważ *`errno`* nie jest używane). Zamiast tego wywołuj `herror()`.

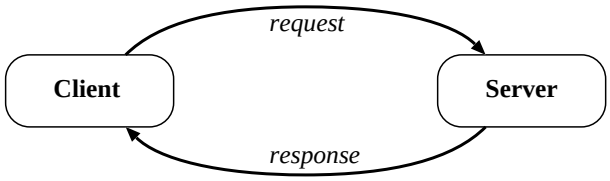
Jest to całkiem proste. Po prostu podajesz tekst, który zawiera nazwę maszyny ("whitehouse.gov") funkcji `gethostbyname()`, a później zbierasz informacje ze zwróconej struktury `struct hostent`.

Jedyną dziwną rzeczą może być wyświetlanie adresu IP pobrane powyższym sposobem. *`h->h_addr`* jest typu `char *`, natomiast `inet_ntoa()` chce typu `struct in_addr`. W tym celu robię odpowiednie rzutowanie *`h->h_addr`* na `struct in_addr*`, a później stosuję dereferencję, żeby dostać dane.

5. Tło Klient-Serwer

To świat klientów i serwerów, chłopcze. Praktycznie wszystkie operacje w sieci to procesy klientów rozmawiające z procesami serwera i vice-versa. Weźmy za przykład **telnet**. Kiedy łączysz się ze zdalnym hostem na porcie 23

Rysunek 2. Interakcja Klient-Serwer.



używając telneta (klient), program na tamtym hoście (nazywany **telnetd**, serwer) budzi się do życia. Zajmuje się przychącymi połączeniami telnetowymi, wyświetla ci prośbę o zalogowanie, itd.

Wymiana informacji pomiędzy klientem i serwerem jest przedstawiona w Figure 2.

Zauważ, że para klient-serwer może rozmawiać używając `SOCK_STREAM`, `SOCK_DGRAM`, lub czegokolwiek innego (dopóki rozmawiają tym samym językiem). Dobrymi przykładami par klient-serwer są **telnet**/**telnetd**, **ftp**/**ftpd**, lub **bootp**/**bootpd**. Za każdym razem, gdy używasz programu **ftp**, po drugiej stronie jest zdalny program, **ftpd**, który ci służy.

Często będzie tylko jeden serwer na maszynie, i ten serwer bedzie obsługiwał wiele klientów używając `fork()`. Najprostszym sposobem jest: serwer czeka na połączenia, akceptuje je (`accept()`) i tworzy proces potomny (`fork()`), który obsłuży to połączenie. To jest właśnie to, co nasz przykładowy serwer robi w następnej sekcji.

5.1. Prosty strumieniowy serwer

Wszystko co robi ten serwer to wysłanie tekstu "`Hello, World\n`" przez połączenie strumieniowe. Wszystko co musisz zrobić, by przetestować ten serwer, to uruchomić go w jednym oknie, i połączyć się za pomocą telneta z drugiego okna:

```
$ telnet remotehostname 3490
```

gdzie `remotehostname` jest nazwą maszyny, na której właśnie pracujesz.

Kod serwera[12]: (Zauważ: znak '\' kończący linię oznacza, że linia jest kontynuowana w następnym wierszu.)

```
/*
** server.c -- serwer używający gniazd strumieniowych
*/

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

```
#define MYPORT 3490    // port, z którym będą się łączyli użytkownicy

#define BACKLOG 10     // jak dużo może być oczekujących połączeń w kolejce

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

int main(void)
{
    int sockfd, new_fd;  // nasłuchuj na sock_fd, nowe połaczenia na new_fd
    struct sockaddr_in my_addr;    // informacja o moim adresie
    struct sockaddr_in their_addr; // informacja o adresie osoby łączącej się
    int sin_size;
    struct sigaction sa;
    int yes=1;

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

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

    my_addr.sin_family = AF_INET;         // host byte order
    my_addr.sin_port = htons(MYPORT);     // short, network byte order
    my_addr.sin_addr.s_addr = INADDR_ANY; // uzupełnij moim adresem IP
    memset(&(my_addr.sin_zero), '\0', 8); // wyzeruj resztę struktury

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

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

    sa.sa_handler = sigchld_handler; // zbierz martwe procesy
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    while(1) {  // głowna pętla accept()
        sin_size = sizeof(struct sockaddr_in);
```

```
        if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
                                                        &sin_size)) == -1) {
            perror("accept");
            continue;
        }
        printf("server: got connection from %s\n",
                                          inet_ntoa(their_addr.sin_addr));
        if (!fork()) { // to jest proces-dziecko
            close(sockfd); // dziecko nie potrzebuje gniazda nasłuchującego
            if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
                perror("send");
            close(new_fd);
            exit(0);
        }
        close(new_fd);  // rodzic nie potrzebuje tego
    }

    return 0;
}
```

Jeśli jesteś tego ciekaw, to umieściłem ten kod w jednej dużej funkcji `main()` dla przejrzystości. Możesz spokojnie rozbić go na mniejsze funkcje jeśli to ci poprawi humor.

(Również ten cały `sigaction()` może być nowy dla ciebie -- nic nie szkodzi. Ten kod jest odpowiedzialny za zbieranie martwych procesów, które się pojawią, gdy procesy-dzieci zakończą działanie. Jeśli zrobisz dużo zombie i nie zbierzesz ich, twój administrator systemu będzie trochę wzburzony.)

Dane z serwera możesz pobrać korzystając z klienta umieszczonego w następnej sekcji.

5.2. Prosty strumieniowy klient

Ten gościu jest nawet prostszy niż serwer. Wszystko co robi ten klient, to łączenie się z podanych w linii komend hostem, na port 3490. Pobiera tekst, który serwer wysyła.

Kod klienta[13]:

```
/*
** client.c -- klient używający gniazd strumieniowych
*/

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

#define PORT 3490 // port, z którym klient będzie się łączył
```

```
#define MAXDATASIZE 100 // maksymalna ilość dancyh, jaką możemy otrzymać na raz

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in their_addr; // informacja o adresie osoby łączącej się

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

    if ((he=gethostbyname(argv[1])) == NULL) {  // pobierz informacje o hoście
        perror("gethostbyname");
        exit(1);
    }

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

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

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

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

    buf[numbytes] = '\0';

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

    close(sockfd);

    return 0;
}
```

Zauważ, że jeśli nie uruchomisz serwera przed klientem, `connect()` zwróci "Connection refused" ("Połączenie odrzucone"). Bardzo użytecznie.

5.3. Gniazda datagramowe

Naprawdę nie trzeba tutaj tak dużo mówić, więc po prostu przedstawię parę przykładowych programów: `talker.c` i `listener.c`.

listener siedzi na maszynie i czeka na pakiety przychodzące na port 4950. **talker** wysyła pakiet na ten port, na do podanej maszyny, który zawiera cokolwiek użytkownik wprowadzi w lini poleceń.

Tu jest źródło dla `listener.c`[14]:

```
/*
** listener.c -- serwer używający gniazd datagramowych
*/

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

#define MYPORT 4950    // port, z którym użytkownicy będą się łączyli

#define MAXBUFLEN 100

int main(void)
{
    int sockfd;
    struct sockaddr_in my_addr;    // informacja o moim adresie
    struct sockaddr_in their_addr; // informacja o adresie osoby łączącej się
    int addr_len, numbytes;
    char buf[MAXBUFLEN];

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

    my_addr.sin_family = AF_INET;         // host byte order
    my_addr.sin_port = htons(MYPORT);     // short, network byte order
    my_addr.sin_addr.s_addr = INADDR_ANY; // uzupełnij moim adresem IP
    memset(&(my_addr.sin_zero), '\0', 8); // wyzeruj resztę struktury

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

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

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

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

    close(sockfd);

    return 0;
}
```

Zauważ, że w naszym wywołaniu `socket()` w końcu używamy `SOCK_DGRAM`. Zauważ również, że nie ma potrzeby korzystania z funkcji `listen()` oraz `accept()`. Jest to jedna z zalet korzystania z niepołączonych gniazd strumieniowych!

Przyszła kolej na kod źródłowy `talker.c`[15]:

```
/*
** talker.c -- klient używający gniazd datagramowych
*/

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

#define MYPORT 4950    // port, z którym będą się łączyli użytkownicy

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; // informacja o adresie osoby łączącej się
    struct hostent *he;
    int numbytes;

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

    if ((he=gethostbyname(argv[1])) == NULL) {  // pobierz informacje o hoście
        perror("gethostbyname");
        exit(1);
    }
```

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

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

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

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

    close(sockfd);

    return 0;
}
```

I to wszystko co się tego tyczy! Uruchom **listener** na jednej maszynie, następnie uruchom **talker** na innej. Patrz jak gadają ze sobą! Fun G-rated excitement for the entire nuclear family!

Poza jednym szczegółem, o którym wspomniałem wiele razy w przeszłości: połączone gniazda datagramowe. Muszę o tym powiedzieć tutaj, ponieważ jesteśmy w sekcji o gniazdach datagramowych. Powiedzmy, że **talker** wywołuje `connect()` i podaje adres programu **listener**. Od tego momentu, **talker** może wysyłać i odbierać dane tylko z podanego funkcji `connect()` adresu. Z tego powodu nie musisz używać `sendto()` ani `recvfrom()`: możesz po prostu używać `send()` i `recv()`.

6. Trochę zaawansowane techniki

Poniższe *wcale* nie są zaawansowane, ale już wychodzą poza poziom podstawowy, który już przeszliśmy. Właściwie, jeśli doszedłeś tutaj, masz już całe podstawy programowania sieciowego w Uniksach! Gratulacje!

A więc wkraczamy w nowy, odważny świat niezwykłych rzeczy, których chcesz się nauczyć.

6.1. Blokowanie

Blokowanie. Już o tym gdzieś słyszałeś -- ale co to do licha jest? W skrócie, "blokowanie" w technicznym żargonie oznacza "usypianie". Zapewne już to zauważyłeś, że jak uruchamiasz **listener**, ten po prostu siedzi i czeka aż nadejdzie pakiet. Wszystko co się stało, to wywołanie `recvfrom()`, a że nie było żadnych danych, to `recvfrom()` się zablokował (usnął), dopóki nie nadejdą dane.

Wiele funkcji się blokuje.`accept()` się blokuje. Wszystkie funkcje `recv()` się blokują. Robią to bo mogą. Gdy utworzysz deskryptor gniazda funkcją `socket()`, jądro pozwala mu się blokować. Jeśli nie chcesz, by gniazdo się blokowało, musisz wywołać `fcntl()`:

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

Ustawiając gniazdo jako nieblokujące, możesz efektywnie wyciągnąć informacje o gnieździe. Jeśli próbujesz czytaj z nieblokującego gniazda i w tym czasie nie ma danych, gniazdo nie ma prawa się zablokować--zwróci ono `-1`, a *`errno`* zostanie ustawione na `EWOULDBLOCK`.

Mówiąc ogólnie, taki sposób wyciągania informacji o stanie jest złym pomysłem. Jeśli zostawisz swój program w pętli sprawdzającej w ten sposób czy nie nadeszły dane, zajmniejsz cały czas procesora. Bardziej eleganckie rozwiącanie sprawdzania czy są dane oczekujące do odczytania jest opisane w sekcji o `select()`.

6.2. `select()` -- Zsynchronizowane wielokrotne operacje I/O

Ta funkcja jest w pewnym stopniu dziwna, ale za to bardzo użyteczna. Weźmy za przykład następującą sytuację: jesteś serwerem i chcesz nasłuchiwać na przychodzące połączenia jak również odczytywać dane z połączeń, które już masz.

Nie ma problemu, pewnie powiesz, wystarczy `accept()` i kilka wywołań `recv()`. Nie tak szybko, cwaniaczku! Co jeśli się zablokujesz na wywołaniu `accept()`? Jak zamierzasz w tym samym czasie odczytywać dane? "Użyj nieblokujących gniazd!" Nie ma mowy! Nie chcesz być świnią dla procesora. Co w takim razie?

`select()` pozwala ci monitorować wiele gniazd w tym samym czasie. Powie ci, które są gotowe do odczytu, które są gotowe do zapisu, oraz które wywołały wyjątki, jeśli naprawdę musisz to wiedzieć.

Bez dalszego mieszania, pokażę ci sposób wywołania `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);
```

Funkcja monitoruje zestawy deksryptorów plików; dokładniej *`readfds`*, *`writefds`*, i *`exceptfds`*. Jeśli chcesz zobaczyć czy możesz czytać ze standardowego wejścia i jakiegoś deskryptora gniazda, *`sockfd`*, po prostu dodaj deksryptory plików `0` i *`sockfd`* do zestawu *`readfds`*. Parametr *`numfds`* powinien być ustawiony na wartość o jeden większą od największego deskryptora pliku. W tym przykładzie, powinieneś ustawić to na *`sockfd+1`*, ponieważ jest to na pewno większe od standardowego wejścia (`0`).

Gdy `select()` powróci, *`readfds`* będzie zmodyfikowane tak, by odpowiadało tym deksryptorom plików, które są gotowe do czytania. Możesz je po kolei sprawdzić makrem `FD_ISSET()`, poniżej.

Przed dalszym zagłębianiem się, powiem jak obsługiwać te zestawy. Każdy zestaw jest typu `fd_set`. Poniższe makra operują na tym typie:

- `FD_ZERO(fd_set *set)` -- czyści zestaw
- `FD_SET(int fd, fd_set *set)` -- dodaje *`fd`* do zestawu
- `FD_CLR(int fd, fd_set *set)` -- usuwa *`fd`* z zestawut
- `FD_ISSET(int fd, fd_set *set)` -- sprawsza, czy *`fd`* jest w zestawie

W końcu, co to jest to dziwne `struct timeval`? Czasami nie chcesz czekać w nieskończoność aż ktoś ci wyśle trochę danych. Może co 96 sekund chcesz wyświetlić "Ciągle działam..." na terminalu nawet jeśli nic się nie stało. Ta struktura czasowa pozwala ci sprecyzować czas przeterminowania. Jeśli czas jest przekroczony a `select()` nadal nie wykrył żadnego gotowego deskryptora pliku, powróci, tak byś mógł dalej przetwarzać dane.

`struct timeval` ma następujące pola:

```
struct timeval {
    int tv_sec;     // sekundy
    int tv_usec;    // mikrosekundy
};
```

Po prostu ustaw *`tv_sec`* na ilośc sekund, jaką mamy odczekać, a *`v_usec`* na ilość mikrosekund, jaką mamy odczekać. Tak, napisałem *mikro*sekund, a nie milisekund. Jest 1000 mikrosekund w jednej milisekundzie, i 1000 milisekund w jednej sekundzie. Stąd jest 1000000 mikrosekund w sekundzie. Dlaczego więc nazywa się to ""usec"? "u" ma wyglądać jak grecka litera Îź (Mu), której używamy jako "mikro". Również gdy funkcja powraca, *`timeout`* *może* być uaktualnione tak, by pokazać ile jeszcze zostało czasu. To zależy od rodzaju Uniksa, którego używasz.

Faaajoowo! Mamy zegar z dokładnością do mikrosekund! W zasadzie, nie licz na to. Standardowy Uniksowy kwant czasu wynosi około 100 milisekund, więc możesz czekać trochę tak długo, niezależnie od tego jak mała dałeś wartość w `struct timeval`.

Inne pola zainsteresowania: jeśli ustawisz pola w swojej strukturze `struct timeval` na `0`, `select()` powróci natychmiast, efektywnie pobierając informacje o wszystkich deskryptorach plików w twoim zestawie. Jeśli ustawisz parametr *`timeout`* na NULL, funkcja nigdy nie powróci z powodu przeterminowania, a więc będzie czekała, dopóki nie odkryje, że przynajmniej jeden deskryptor pliku jest gotowy. Jeśli nie zależy ci na czekaniu na dany zestaw, może go po prostu wstawić ja NULL w wyołaniu `select()`.

Poniższy kawałek kodu[16] czeka 2,5 sekundy na cokolwiek, żeby się pojawiło na standardowym wejściu:

```
/*
** select.c -- pokaz użycia select()
*/

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

#define STDIN 0  // deskryptor pliku dla standardowego wejścia

int main(void)
```

```
{
    struct timeval tv;
    fd_set readfds;

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

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

    // nie martw się o writefds i exceptfds:
    select(STDIN+1, &readfds, NULL, NULL, &tv);

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

    return 0;
}
```

Jeśli używasz terminala buforującego linię, klawisz, który powinieneś wciśnąć to RETURN, albo zostanie przekroczony dozwolony czas.

Niektórzy z was mogą myśleć, że jest to świetny sposób na czekania na dane na gniazdach datagramowych -- i macie rację: *mogą*. Niektóre z Uniksów mogą korzystać z select w ten sposób, inne nie. Powienieś zobaczyć co mówi na ten temat strona podręcznika systemowego, jeśli chcesz z tego korzystać.

Niektóre Uniksy uaktualniają czas w twoim `struct timeval` by pokazać ile czasu jeszcze zostało przed przeterminowaniem. Ale inne nie. Nie bazuj na tym założeniu, jeśli chcesz być przenośny (użyj `gettimeofday()` jeśli chcesz śledzić mijający czas. Wiem, jest to pokręcone, ale tak już jest).

Co się dzieje, gdy gniazdo w stanie odczytu zamknie połączenie? W takim przypadku `select()` powraca z tym deskryptorem gniazda zaznaczonym jako "gotowe do odczytu". Gdy już wywołasz `recv()` na nim, `recv()` zwróci `0`. W taki sposób wiesz, że klient zamknął połączenie.

Jeszcze jedna uwaga o `select()`: jeśli masz gniazdo, które nasłuchuje na połączenia, możesz sprawdzić, czy są nowe połączenia, dodając deskryptor tego gniazda do zestawu *`readfds`*.

I to, moi przyjaciele, było krótkie omówienie wszechmocnego `select()`.

Ale, zgodnie z żądaniami, tu jest dogłębny przykład. Niestety, różnica pomiędzy powyższymi krótkimi, prostymi przykładami a tym tutaj jest znaczna. Ale mimo to przejrzyj ten przykład i przeczytaj opis, który po nim następuje.

Ten program[17] działa jak prosty, wieloużytkownikowy serwer chat. Uruchom go w jednym oknie, potem za**telnet**etuj się na niego ("**telnet hostname 9034**") kilkakrotnie z różnych okien. Gdy coś napiszesz w jednej sesji **telnet**a, ten tekst powinien się pojawić w każdej innej sesji.

```
/*
** selectserver.c -- fajoski, wieloosobowy serwer chat
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

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

#define PORT 9034   // port, na którym nasłuchujemy

int main(void)
{
    fd_set master;   // główna lista deskryptorów plików
    fd_set read_fds; // pomocnicza lista deskryptorów dla select()
    struct sockaddr_in myaddr;     // adres serwera
    struct sockaddr_in remoteaddr; // adres klienta
    int fdmax;        // maksymalny numer deskryptora pliku
    int listener;     // deskryptor gniazda nasłuchującego
    int newfd;        // nowozaakceptowany deskryptor gniazda
    char buf[256];    // bufor na dane pochodzące od klienta
    int nbytes;
    int yes=1;        // dla setsockopt() SO_REUSEADDR, patrz niżej
    int addrlen;
    int i, j;

    FD_ZERO(&master);    // wyczyść główny i pomocniczy zestaw
    FD_ZERO(&read_fds);

    // utwórz gniazdo nasłuchująceo
    if ((listener = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    // zgub wkurzający komunikat błędu "address already in use"
    if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes,
                                                    sizeof(int)) == -1) {
        perror("setsockopt");
        exit(1);
    }

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

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

```
// dodaj gniazdo nasłuchujące do głównego zestawu
FD_SET(listener, &master);

// śledź najwyższy numer deskryptora pliku
fdmax = listener; // póki co, ten jest największy

// pętla główna
for(;;) {
    read_fds = master; // copy it
    if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
        perror("select");
        exit(1);
    }

    // przejdź przez obecne połączenia szukając danych do odczytania
    for(i = 0; i <= fdmax; i++) {
        if (FD_ISSET(i, &read_fds)) { // mamy jednego!!
            if (i == listener) {
                // obsłuż nowe połączenie
                addrlen = sizeof(remoteaddr);
                if ((newfd = accept(listener, (struct sockaddr *)&remoteaddr,
                                                    &addrlen)) == -1) {
                    perror("accept");
                } else {
                    FD_SET(newfd, &master); // dodaj do głównego zestawu
                    if (newfd > fdmax) {    // śledź maksymalny
                        fdmax = newfd;
                    }
                    printf("selectserver: new connection from %s on "
                        "socket %d\n", inet_ntoa(remoteaddr.sin_addr), newfd);
                }
            } else {
                // obsłuż dane od klienta
                if ((nbytes = recv(i, buf, sizeof(buf), 0)) <= 0) {
                    // błąd lub połączenie zostało zerwane
                    if (nbytes == 0) {
                        // połączenie zerwera
                        printf("selectserver: socket %d hung up\n", i);
                    } else {
                        perror("recv");
                    }
                    close(i); // papa!
                    FD_CLR(i, &master); // usuń z głównego zestawu
                } else {
                    // mamy trochę danych od klienta
                    for(j = 0; j <= fdmax; j++) {
                        // wyślij do wszystkich!
                        if (FD_ISSET(j, &master)) {
                            // oprócz nas i gniazda nasłuchującego
                            if (j != listener && j != i) {
                                if (send(j, buf, nbytes, 0) == -1) {
                                    perror("send");
                                }
                            }
                        }
```

```
                    }
                }
            } // jakie to BRZYDKIE!!
        }
    }
}

    return 0;
}
```

Zauważ, że użyłem dwóch zestawów w kodzie: *`master`* i *`read_fds`*. Pierwszy, *`master`*, przechowuje wszystkie deskryptory gniazda, które są aktulanie połączone, jak również deskryptor gniazda nasłuchującego na połączenia.

Powodem użycia zestawu *`master`* jest to, że `select()` *zmienia* podany zestaw tak, by znajdowały się w nim tylko te gniazda, które są gotowe do czytania. Ponieważ muszę mieć pełną listę połączeń niezależnie od wywołania `select()`, muszą ją przechowywać w bezpiecznym miejscu. W ostatniej chwili kopiuję zawartość *`master`* do *`read_fds`*, a dopiero potem wywołuję `select()`.

Ale czy nie oznacza to, że za każdym nowym połączeniem muszę je dodać do zestawu *`master`*? Dokładnie! I za każdym razem, gdy połączenie zostanie zamknięte muszę usunąć je z zestawu *`master`*? Tak, dokładnie.

Zauważ, że sprawdzam, kiedy gniazdo *`listener`* jest gotowe do odczytu. Kiedy jest, oznacza to, że mam nowe, oczekujące połączenie, i wtedy je akceptuję (`accept()`) i dodaję je do zestawu *`master`*. Podobnie, gdy połączenie z klientem jest gotowe do odczytu, a `recv()` zwraca `0` wiem, że klient zamknął połączenie, a ja muszę usunąć je z zestawu *`master`*.

Jeśli natomiast `recv()` zwróci wartość niezerową, wiem, że pewna ilość danych została odebrana. Więc biorę ją i przechodzę przez całą listę *`master`* i wysyłam te dane do reszty połączonych klientów.

I to, moi przyjaciele, jest mniej niż prosty przegląd tajemniczej funckji `select()`.

6.3. Radzenie sobie z niepełnym wysyłaniem

Pamiętasz, jak w sekcji o `send()`, powyżej, powiedziałem, że `send()` może nie wysłać wszystkich danych, które mu podałeś? Na przykład, chcesz, że wysłał 512 bajtów, ale on zwraca 412. Co się stało z pozostałymi 100 bajtami?

Są one ciągle w twoim małym buforze i czekają na wysłanie. Z pwodów niezależnych od ciebie, jądro zdecydowało nie wysyłać wszystkich danych za jednym razem, i teraz, przyjacielu, wszystko zależy od ciebie czy wyświesz resztę danych.

Mógłbyś napisać funckcję jak ta poniżej, która zrobi to za ciebie:

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

int sendall(int s, char *buf, int *len)
{
    int total = 0;        // ile bajtów już wysłaliśmy
    int bytesleft = *len; // ile jeszcze zostało do wysłania
    int n;

    while(total < *len) {
```

```
        n = send(s, buf+total, bytesleft, 0);
        if (n == -1) { break; }
        total += n;
        bytesleft -= n;
    }

    *len = total; // zwróć ilość faktycznie wysłanych bajtów

    return n==-1?-1:0; // zwróć -1 w przypadku błędu, 0, gdy się powiedzie
}
```

W tym przykładzie *`s`* jest gniazdem, do którego chcesz wysłać dane. *`buf`* jest buforem zawierającym dane, a *`len`* jest wskaźnikiem do zmiennej typu `int`, zawierającej ilość bajtów, jaką zajmują dane w buforze.

Funkcja zwraca `-1` w przypadku błędu (a *`errno`* jest ciągle ustawione z wywołania `send()`). Także ilość faktycznie wysłanych danych jest zwracana w zmiennej *`len`*. Będzie to taka sama liczba, jaką podałeś przy wowoływaniu funkcji, chyba że wystąpi błąd. `sendall()` zrobi co może, sąpiąc i dysząc, by wysłać dane, ale jeśli wystąpi błąd, funkcja natychmiast powraca.

Dla zachowania całości, oto przykładowe użycie tej funkcji:

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

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

Co się dzieje na drugim, odbierającym dane, końcu gdy część pakietu dochodzi? Jeśli pakiety są zmiennej długości, skąd odbiorca wie, gdzie się kończy jeden pakiet, a zaczyna drugi? Yes, real-world scenarios are a royal pain in the donkeys (od tłumacza: niech to ktoś przetłumaczy!!). Prawdopodobnie będziesz musiał *owinąć* te dane w odpowiednie nagłówki (pamiętasz to z sekcji o enkapsulacji danych na samiusieńkim początku?). Przeczytaj, jeśli chcesz znać szczegóły!!

6.4. Enkapsulacja danych

Co to właściwie znaczy enkapsulować dane? W najprostszym przypadku oznacza to, że dodajesz nagłówek do nich z jakąś informacją identyfikującą dane lub długość pakietu, lub obie te informacje.

Jak powinien wyglądać twój nagłówek? W zasadzie są to binarne dane, które posiadają wszystko to, co powinno się tam znaleść byś mógł dokończyć projekt.

Nieźle. A jakie niejasne.

No dobra. Weźmy za przykład taką sytuację: masz wieloużytkownikowy program chat, który użycia `SOCK_STREAM`. Gdy użytkownik coś napisze (powie), dwa kawałki informacji muszą być przesłane do serwera: to co było powiedziane oraz kto to powiedział.

Dotąd wszystko pasuje? "Co za problem?" pytasz.

Problem jest taki, że wiadomości mogą być różnych długości. Jedna osoba o nazwie "tom" może powiedzieć "Hi", a inna zwana "Benjamin" może powiedzieć "Hey guys what is up?".

A więc wysyłasz (`send()`) cały ten towar do klientów tak jak on przychodzi. Strumień wyjściowy będzie wyglądał tak:

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

I tak dalej. Jakim cudem klient będzie wiedział, gdzie się jedna wiadomość zaczyna, a inna kończy? Mógłbyś, gdybyś chciał, sprawić by wszystkie wiadomości były tego samego rozmiaru i wtedy dopiero wywołujesz `sendall()`, który zakodowaliśmy powyżej. Ale to marnuje przepustowość! Nie chcemy wysyłać 1024 bajtów tylko po to, by "tom" mógł powiedzieć "Hi".

Tak więc *owijamy* dane w mały nagłówek i strukturę pakietu. Zarówno klient jak i serwer wiedzą jak zapakować i rozpakować (czasami jest to określane jako "marshal" i "unmarshal") te dane. Nie patrz teraz, ale zaczynamy definiować *protokół*, który określi jak klient rozmawia z serwerem!

W tym przypadku załóżmy, że nazwa użytkownika jest stałej długości i składa się z ośmiu znaków, dopełnionych znakiem `'\0'`. Następnie przyjmijmy, że dane są zmiennej długości, składające się maksymalnie ze 128 znaków. Spójrzmy teraz na przykładową strukturę pakietu, której możemy użyć w tej sytuacji:

1. `len` (1 bajt, bez znaku) -- całkowita długość pakietu, licząc ośmiobajtową nazwę użytkownika i dane wiadomości.
2. `name` (8 bajtów) -- nazwa użytkownika, dopełniona znakiem NULL, jeśli potrzeba.
3. `chatdata` (*n*-bajtów) -- same dane, nie więcej niż 128 bajtów. Długość pakietu powinna być obliczana jako długość tych danych plus 8 (długość pola nazwy użytkownika, powyżej).

Dlaczego wybrałem 8-bajtowe i 128-bajtowe limity dla powyższych pól? Wziąłem je z powietrza, zakładając, że będą odpowiednio długie. Możliwe, że 8 bajtów jest zbyt dużą restrykcją dla twoich potrzeb, ale możesz mieć 30-bajtowe nazwy użytkownika, jeśli chcesz. Wszystko zależy od ciebie.

Używając powyższe definicji pakietu, pierwszy pakiet składałby się z następujących informacji (w heksach i ASCII):

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

A drugi byłby podobny:

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

(Długość jest przechowywana w Network Byte Order, oczywiście. W tym przypadku jest to tylko jeden bajt, więc nie robi to różnicy, ale mówiąc ogólnie będziesz chciał przechowywać wszystkie binarne liczby w Network Byte Order w twoich pakietach.)

Gdy wysyłasz te dane, powinieneś być bezpieczny i używać komendy podobnej do <`sendall()`, powyżej, dzięki czemu wiesz, że wszystkie dane zostały wysłane, nawet jeśli potrzeba wykonać wiele wywołań do `send()`, by wszystkie dane wysłać.

Podobnie, gdy odbierasz dane, musisz włożyć w to trochę więcej pracy. By czuć się bezpiecznym, powinieneś założyć, że możesz odebrać tylko część pakietu (może to być "`00 14 42 65 6E`" od Bejnamina, ale to wszystko co otrzymamy w tym wywołaniu `recv()`). Musimy wywoływać `recv()` i wywoływać, dopóki cały pakiet nie jest odebrany.

Ale jak? Znamy ilość bajtów, które musimy odebrać, by pakiet był kompletny, ponieważ ta informacja jest umieszczona na początku pakietu. Znamy również maksymalny rozmiar pakietu wynoszący 1+8+128, czyli 137 bajtów (ponieważ tak zdefiniowaliśmy pakiet).

Możesz zadeklarować wystarczająco duży ciąg dla dwóch pakietów. Jest to twój roboczy ciąg, w którym będzie rozpakowywał pakiety, podczas gdy one nadchodzą.

Za każdym razem, gdy odbierasz (`recv()`) dane, musisz je umieścić w buforze roboczym i sprawdzić czy pakiet jest kompletny. Dokładniej, czy ilość bajtów w buforze jest większa niż lub równa długości podanej w nagłówku (+1, ponieważ długość w nagłówku nie wlicza siebie). Jeśli ilość bajtów w buforze jest mniejsza niż 1, pakiet nie jest kompletny, oczywiście. Musisz zrobić specjalny warunek dla tego, ponieważ pierwszy bajt jest śmieciem i nie możesz na nim polegać, by określić długość pakietu.

Gdy już pakiet jest kompletny, możesz z nim zrobić co tylko zechcesz. Użyj go, i usuń z twojego bufora roboczego.

Czy powyższe nie skacze ci jeszcze po głowie? Bo jest też druga możliwość: możesz odebrać za jednym pakietem kawałek kolejnego w jednym wywołaniu `recv()`. Stąd, masz w buforze roboczym jeden cały pakiet i część pakietu następnego. Potworna sprawa (ale to właśnie dlatego zrobiłeś swój bufor roboczy odpowiednio duży, by przechować *dwa* pakiety -- w przypadku gdyby się to zdarzyło!).

Ponieważ znasz długość pierwszego pakietu z nagłówna i śledziłeś ilość bajtów w buforze roboczym, możesz odjąć i obliczyć ile bajtów bufora roboczego należy do drugiego, niekompletnego pakietu. Gdy uporasz się z pierwszym, możesz wyczyścić bufor roboczy i przenieść część drugiego pakietu na początek bufora, dzięki czemu będzie on gotowy na kolejne wywołania `recv()`.

(Niektórzy czytelnicy zauważą, że przenoszenie części drugiego pakietu na poczatek bufora roboczego zabiera czas, a program może być tak napisany, że nie będzie tego wymagał przez zaimplementowanie bufora kołowego. Niestety dla innych, dyskusja o buforach kołowych wychodzi poza ramy tego arytkułu. Jeśli nadal jesteś tego ciekaw, dobierz się do książki o strukturach danych.)

Nigdy nie powiedział, że będzie łatwo. No dobra, powiedziałem. I jest: musisz po prostu potrenować i wkrótce będzie to dla ciebie naturalne. Przysięgam na moc Excalibura!

7. Więcej informacji

Doszedłes aż dotąd i jeszcze ci mało! Gdzie jeszcze możesz iść, by się nauczyć więcej o wszystkich tych rzeczach?

7.1. man Pages

Sprawdź poniższe strony mana, dla początkujących:

- `htonl()`[18]
- `htons()`[19]

- `ntohl()`[20]
- `ntohs()`[21]
- `inet_aton()`[22]
- `inet_addr()`[23]
- `inet_ntoa()`[24]
- `socket()`[25]
- `socket options`[26]
- `bind()`[27]
- `connect()`[28]
- `listen()`[29]
- `accept()`[30]
- `send()`[31]
- `recv()`[32]
- `sendto()`[33]
- `recvfrom()`[34]
- `close()`[35]
- `shutdown()`[36]
- `getpeername()`[37]
- `getsockname()`[38]
- `gethostbyname()`[39]
- `gethostbyaddr()`[40]
- `getprotobyname()`[41]
- `fcntl()`[42]
- `select()`[43]
- `perror()`[44]
- `gettimeofday()`[45]

7.2. Książki

Jeśli jesteś zwolennikiem starodawnych, "trzymanych w rękach", miękich, papierowych książek, sprawdź poniższe wspaniałe przewodniki. Zauważ wybitne logo Amazon.com. Cały ten bezwstydny komercjalizm jest dlatego, że ja na tym zyskuje (upusty w Amazon.com) za sprzedawanie tych książek za pośrednictwem tego przewodnika. Więc

jeśli zamierzasz zamówić jedną z nich, dlaczego nie wyrazisz specjalnego podziękowania, poprzez użycie poniższych linków?

Dodatkoweo, więcej książek dla mnie prowadzi do większej ilości przewodników dla ciebie. ;-)

[46]

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

Internetworking with TCP/IP, volumes I-III by Douglas E. Comer and David L. Stevens. Published by Prentice Hall. ISBNs for volumes I, II, and III: 0130183806[49], 0139738436[50], 013848714 6[51].

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

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

Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by Addison Wesley. ISBN 0201563177[56].

Using C on the UNIX System by David A. Curry. Published by O'Reilly & Associates, Inc. ISBN 0937175234. *Out of print.*

7.3. W sieci

W sieci:

BSD Sockets: A Quick And Dirty Primer[57] (posiada również informacje na temat programowania systemowego w Uniksach!)

The Unix Socket FAQ[58]

Client-Server Computing[59]

Intro to TCP/IP[60] (gopher)

Internet Protocol Frequently Asked Questions[61]

The Winsock FAQ[62]

7.4. Dokumenty RFC

RFCs[63]--prawdziwy brud:

RFC-768[64]--The User Datagram Protocol (UDP)

RFC-791[65]--The Internet Protocol (IP)

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

RFC-854[67]--The Telnet Protocol

RFC-951[68]--The Bootstrap Protocol (BOOTP)

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

8. FAQ

Pyt: Gdzie mogę dostać te pliki nagłówkowe?

Odp: Jeśli nie masz ich jeszcze w swoim systemie, prawdopodobnie ich nie potrzebujesz. Sprawdź podręcznik dla twojej platformy. Jeśli budujesz dla Windows, musisz tylko dodać `#include <winsock.h>`.

Pyt: Co mam zrobić, gdy `bind()` zgłasza "Address already in use"?

Odp: Musisz użyć `setsockopt()` z opcją `SO_REUSEADDR` na nasłuchującym gnieździe. Przejrzyj sekcję o `bind()` oraz sekcję o `select()` dla przykładów.

Pyt: Gdzie mogę znaleźć listę otwartch gniazd w systemie?Gdzie mogę znaleźć listę otwartch gniazd w systemie?

Odp: Użyj **netstat**. Przejrzyj strony **man**a po szczegóły, ale powinieneś otrzymać dobry wynik wpisując:

```
$ netstat
```

Jedną trudnością może być określenie, które gniazdo należy do którego programu. `:-)`

Pyt: Jak mogę obejrzeć tabelę routingu?

Odp: Uruchom program **route** (znajduje się w `/sbin` w większości dystrybucji Linksa) lub **netstat -r**>.

Pyt: Jak mogę uruchomić programy klienta i serwera, jeśli mam tylko jeden komputer? Czy nie potrzebuję sieci, by pisać programy sieciowe?

Odp: Na szczęście dla ciebie, praktycznie wszystkie maszyny implementują sieciowe urządzenie zwrotne (tzw. loopback), które siedzi w jądrze i udaje prawdziwą kartę sieciową (to urządzenie widnieje jako "`lo`" w tabeli routingu).

Załózmy, że jesteś zalogowany na maszynie nazwanej "`goat`". Uruchom klienta w jednym oknie, a serwer w drugim. Lub wystartuj serwer w tle ("**server &**") i uruchom klienta w tym samym oknie. Dzięki urządzeniu loopback możesz wywołać zarówno **client goat** jak i **client localhost** (ponieważ "`localhost`" jest najprawdopodobniej zdefiniowany w twoim pliku `/etc/hosts`) i otrzymasz klienta rozmawiającego z serwerem bez pośrednictwa sieci!

W skrócie, żadne zmiany w kodzie nie są wymagane, by programy sieciowe chodziły na maszynach nie podłączonych do sieci! Hurraa!

Pyt: Jak mogę stwierdzić, że druga strona zamknęłą połączenie?

Odp: Możesz to stwierdzić, ponieważMożesz to stwierdzić, ponieważ `recv()` zwróci `0`.

Pyt: Jak zaimplementować narzędzie "ping"? Co to jest ICMP? Gdzie mogę się dowiedzieć więcej o surowych ("raw") gniazdach oraz `SOCK_RAW`?

Odp: Odpowiedzi na wszystkie twoje pytania odnośnie surowych gniazd znajdziesz w książkach W. Richard Stevensa z serii "UNIX Network Programming". Patrz sekcję o książkach w tym przewodniku.

Pyt: Jak budować dla Windows?

Odp: Najpierw usuń Windows i zainstaluj Linuksa lub BSD. `};-)`. Nie, właściwie po prostu zobacz sekcję o budowaniu dla Windows we wprowadzeniu.

Pyt: Jak budować dla Solaris/SunOS? Ciągle otrzymuję błędu łączenia, gdy próbuję kompilować!

Odp: Błędy łączenia zdarzają sie, poniważ maszyny Sun nie wkompilowują automatycznie biblioteki gniazd. Patrz sekcję o budowaniu dla Solaris/SunOS we wprowadzniu dla przykładu jak to zrobić.

Pyt: Dlaczego `select()` ciągle powraca, gdy otrzyma sygnał?

Odp: Sygnały sprawiają, że blokujące wywołania systemowe zwracają `-1` z *`errno`* ustawionym na `EINTR`. Gdy ustawisz funkcję obsługi sygnału za pomocą `sigaction()`, możesz ustawić flagę `SA_RESTART`, która powinna zrestartować wywołanie systemowe, po tym jak zostało przerwane.

Naturalnie to nie zawsze działa.

Moim ulubionym rozwiązaniem jest użycie konstrukcji `goto`. Wiesz, że to irytuje profesorów, więc używaj jej!

```
select_restart:
    if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
        if (errno == EINTR) {
            // jakiś sygnał właśnie nam przerwał robotę, więc restartujemy
            goto select_restart;
        }
        // obsłuż prawdziwy błąd tutaj:
        perror("select");
    }
```

Oczywiście nie *musisz* używać `goto` w tym przypadkue; możesz użyć innych struktur, by to kontrolować. Ale myślę, że wyrażenie `goto` jest czytelniejsze.

Pyt: Jak zaimplementować przedawnienie ("timeout") funkcji `recv()`?

Odp: Użyj `select()`! To ci pozwoli podać parametr przedawnenie dla deskryptora gniazda, z którego chcesz odczytywać. Możesz też owinąc całą tą funkcjonalność w jedną funkcję, taką jak ta:

```
#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;

    // ustaw zestaw deskryptorów plików
    FD_ZERO(&fds);
    FD_SET(s, &fds);

    // ustaw strukturę timeval dla przedawnienia
    tv.tv_sec = timeout;
    tv.tv_usec = 0;

    // czekaj na przedawnienie lub na odczytanie danych
    n = select(s+1, &fds, NULL, NULL, &tv);
    if (n == 0) return -2; // przedawnienie!
    if (n == -1) return -1; // błąd

    // danę już muszą być, więc wywołaj normalnie recv()
    return recv(s, buf, len, 0);
}

// Przykładowe wywołanie recvtimeout():
    .
    .
    n = recvtimeout(s, buf, sizeof(buf), 10); // przedawnienie po 10 sekundach

    if (n == -1) {
        // wystąpił błąd
        perror("recvtimeout");
    }
    else if (n == -2) {
        // wystąpiło przedawnienie
    } else {
        // mamy jakieś dane w buforze
    }
    .
    .
```

Zauważ, że `recvtimeout()` zwraca `-2` w przypadku przedawnienia. Dlaczego nie zwraca `0`? Jeśli sobie przypomnisz, to zwrócona wartość `0` przez wywołanie `recv()` oznacza, że druga strona zamknęła połączenie. Więc ta wartość jest już zajęta, a `-1` oznacza "błąd", więc wybrałem `-2` jako moja informacja o przedawnieniu.

Pyt: Jak mam szyfrować lub kompresować dane przed ich wysłaniem do gniazda?

Odp: Jedną z prostych dróg szyfrowania byłoby skorzystanie z SSL (secure sockets layer - bezpieczna wartswa gniazd), ale wychodzi to poza ramy tego przewodnka.

Ale zakładając, że chcesz dołączyć lub zaimplementować swój własny system kompresujący lub szyfrujący, jest to po prostu kwestia przemyślenia sposobu przechodzenia danych przez poszczególne kroki pomiędzy dwoma końcami. Każdy krok zmienia dane w jakiś sposób.

1. serwer odczytuje dane z pliku (lub czegokolwiek)
2. serwer szyfruje dane (dodajesz tą część)
3. serwer wysyła zaszyfrowane dane

Teraz w drugą stronę:

4. klient odbiera zaszyfrowane dane
5. klient odszyfrowuje dane (dodajesz tą część)
6. klient zapisuje dane do pliku (lub czegokolwiek)

Możesz również zrobić kompresję zamiast punktu, w którym robisz szyforwanie. Lub możesz wykonać te dwie czynności! Pamiętaj tylko o tym, że najpierw się kompresuje, potem szyfruje. `:)`

Dopóki klient poprawnie odtworzy to, co stworzył serwer, dane będą dobre, nieważne ile dodasz kroków pośrednich.

Tak więc, wszystko co musisz zrobić, by wykorzystać mój kod, to znalezieni miejsca, między którym dane są odczytywane i wywsyłane (używając `send()`) przez sieć), i dodanie kawałka kodu w tym miejscu, który będzie szyfrował.

Pyt: Co to jest "`PF_INET`", które ciągle widzę? Czy jest to powiązane z `AF_INET`?

Odp: Tak, jest powiązane. Przejrzyj sekcję o `socket()` jeśli interesują cię szczegóły.

Pyt: Jak mam napisać serwer, który przyjmuje komendy shella od klienta i je wykonuje?

Odp: Dla prostoty, załóżmy, że klient łączy się (`connect()`), wysyła dane (`send()`), i zamyka (`close()`) połączenie (tak więc nie ma kolejnych wywołań bez ponownego połączenia).

Klient działą wg. następującego kolejności:

1. `connect()` - połączenie z serwerm
2. `send("/sbin/ls > /tmp/client.out")`
3. `close()` - zamknięcie połączenia

Tymczasem, serwer przetwarza dane i je wykonuje:

1. `accept()` - zaakcpetowanie połączenia od klienta
2. `recv(str)` - wczytanie tekstu komendy
3. `close()` - zamknięcie połączenia
4. `system(str)` - wywołanie komendy

Strzeż się! Pozwalanie serwerowi wykonywać wszystko co każe klient to jak danie zdalnego dostępu do shella i pozwolenie ludziom robienia różnych rzeczy na twoim końcie, za każdym razem gdy się łączą z serwerem. Na przykład, w powyższym przykładzie, co będzie, gdy klient wyśle "**rm -rf ~**"? Skasuje to wszystkie pliki z twojego konta, ot co się stanie!

Tak więc jesteś już mądry i zapobiegasz wykonywania dowolnej komedny poza kilkoma wybranymi narzędziamy, o których wiesz, że są bezpieczne, jak np. narzędzie **foobar**:

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

Ale nadal nie jesteś bezpieczny, niestety: co jeśli klient wyśle "**foobar; rm -rf ~**"? Najbezpieczniejszą rzeczą jest napisanie funkcji, która wstawia znak uciczki ("\") przez każdym niealfanumerycznym znakiem (włączając w to spacje, jeśli zachodzi taka potrzeba) w argumentach dla komendy.

Jak widzisz, bezpieczeństwo jest dużą sprawą, gdy serwer zaczyna uruchamiać rzeczy, które podsyła klient.

Pyt: Wysyłam duże porcje danych, ale gdy wywołuję `recv()`, funkcja zwraca tylko 536 bajtów lub 1460 bajtów na raz. Ale jeśli uruchomię to na mojej lokalnej maszynie, funkcja zwraca wszystkie dane na raz. Co się dzieje?

Odp: Jest to wina MTU -- maksymalny rozmiar pakietu, z jakim sobie poradzi fizyczne medium. Na maszynie lokalnej, używasz urządzenia loopback, które obsłuży 8 KB lub więcej bez problemu. Ale na ethernet, które może obsłużyć tylko 1500 bajtów razem z nagłówkiem, trafiasz na limit. Przez modem, z MTU wynoszącym 576 bajtów (również z nagłówkiem), trafiasz na jeszcze mniejszy limit.

Po pierwsze, musisz się upewnić, że wszystkie dane są wysyłane (zobacz implementację funkcji `sendall()` po szczegóły). Jak już jesteś tego pewien, musisz wywoływać `recv()` w pętli, dopóki wszystkie dane nie są odczytane.

Przeczytaj sekcję Enkapsulacja danych po szczegóły o odbieraniu pełnych pakieków używając wielokrotnych wywołań `recv()`.

Pyt: Siedzę na maszynie z Windows i nie mam wywołania `fork()` jak również żadnego z `struct sigaction`. Co robić?

Odp: Jeśli mają gdzieś być, to będą w bibliotekach POSIX, które możesz mieć dostarczone z kompilatorem. Ponieważ nie mam maszyny z Windows, nie mogę odpowiedzieć, ale wydaje mi się, że Microsoft ma warstwę kompatybilności z POSIX i tam właśnie może być `fork()` (i pewnie też `sigaction`).

Przeszukaj pomoc, która przyszła z VC++ za słowem "fork" lub "POSIX" i zobacz, czy daje ci to jakieś wskazówki.

Jeśli to wogóle nie działa, zapomnij o `fork()`/`sigaction` i zastąp je odpowiednikami Win32: `CreateProcess()`. Nie wiem jak używać `CreateProcess()` -- pobiera ona miliony argumentów, ale powinno to byc ujęte w pomocy VC++.

Od tłumacza: jest projekt Cygwin[70], którego celem jest przeniesienie bibliotek POSIX jak również programów Uniksowych na system Windows. Tam też znajdziesz wywołania `fork()` czy `sigaction`.

Pyt: Jak bezpiecznie wysyłać dane przez TCP/IP przy użyciu szyfrowania?

Odp: Sprawdź projekt OpenSSL[71].

Pyt: Jestem za firewallem -- jaki mam podać adres IP ludziom z zewnątrz firewalla, by mogli połączyć się z moją maszyą?

Odp: Niestety, celem firewalla jest zabronienie ludziom z zewnątrz łączenia się z maszynami z wewnątrz, więc pzowolenie ludziom z zewnątrz na połaczenie się z maszyną lokalną jest naruszeniem bezpieczeństwa.

Ale nie wszystko stracone. Nadal możesz się łączyć przez firewall, jeśli ten wykonuje maskaradę lub NAT lub coś takiego. Po prostu zaprojektuj swoje programy tak, by to one zawsze nawiązywały połaczenie i będzie dobrze.

Jeśli to cię nie satysfakcjonuje, możesz poprosić swoich adminów, by zrobili dziurę w firewallu tak, żeby ludzie mogli się z tobą łączyć. Firewall może przekazywać pakiety do ciebie przez NAT lub serwer proxy.

Ale pamiętaj, że dziura w firewallu nie jest czymś, wobec czego można przejść obojętnie. Musisz się upewnić, że nie dasz złym ludziom dostępu do sieci wewnętrznej; jeśli jesteś początkujący, jest dużo trudniej uczynić programy bezpiecznymi niż może ci się zdawać.

I nie pozwól swoim adminom być wściekłymi na mnie. `;-)`

9. Wołanie o pomoc

Trochę dużo tego było. Mam nadzieję, że przynajmniej część zamieszczonych tu informacji była dokładna i szczerze wierzę, że nie ma tu żadnych poważnych błądów. Zgoba, oczywiste jest, że wszędzie są błedy.

Tak więc, niech to będzie dla ciebie ostrzeżeniem! Przepraszam, jeśli pewne nieścisłosci zawrate tu i ówdzie wywołały u ciebie smutek, ale nie możesz mnie czynić za to odpowiedzialny. Widzisz, nie odpowiadam za nawet pojedyncze słowo z tego dokumentu, bowiem wszystko to może być komplenie i całkowicie złe.

Ale prawdopodobnie nie jest. Przecież spędziłem wiele, wiele godzin babrając się tymi rzeczami, zaimplementowałem liczne narzędzia sieciowe TCP/IP korzystając z tych informacji, napisałem silniki do gier dla wielu użytkowników, i tak dalej. Ale nie jestem Bogiem, jeśli chodzi o gniazda; jestem po prostu człowiekiem.

Przy okazji, jeśli ktokolwiek ma konstruktywną (lub nie) krytykę odnośnie tego dokumentu, proszę o wysłanie maila na adres<`beej@piratehaven.org`> a ja spróbuję zrobić co mogę, by spostować informacje.

Jeśli się zastanawiasz dlaczego to zrobiłem, to powiem, że zrobiłem to dla pieniędzi. Ha! Prawdę mówiąc, nie. Zrobiłem to, ponieważ wiele ludzi zadawało mi pytania związane z gniazdami i kiedy mówiłem im, że myślałem nad zebraniem tego wszystkiego do kupy na jednej stronie o gniazdach, oni mówili "Faaaajnie!". Poza tym, myślę, że cała ta ciężko zdobyta wiedza się zmarnuję, jeśli jej nie podzielę z innymi. Sieć jest akurat doskonałym nośnikiem wiedzy. Zachęcam innych, by dostarczali takich informacji, jeśli jest to tylko możliwe.

Dobra, koniec tego -- wracaj do kodowania! `;-)`

Przypisy

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

22. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_aton.3.inc
23. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_addr.3.inc
24. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/inet_ntoa.3.inc
25. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/socket.2.inc
26. http://linux.com.hk/man/showman.cgi?manpath=/man/man7/socket.7.inc
27. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/bind.2.inc
28. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/connect.2.inc
29. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/listen.2.inc
30. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/accept.2.inc
31. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/send.2.inc
32. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recv.2.inc
33. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/sendto.2.inc
34. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/recvfrom.2.inc
35. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/close.2.inc
36. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/shutdown.2.inc
37. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getpeername.2.inc
38. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/getsockname.2.inc
39. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyname.3.inc
40. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/gethostbyaddr.3.inc
41. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/getprotobyname.3.inc
42. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/fcntl.2.inc
43. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/select.2.inc
44. http://linux.com.hk/man/showman.cgi?manpath=/man/man3/perror.3.inc
45. http://linux.com.hk/man/showman.cgi?manpath=/man/man2/gettimeofday.2.inc
46. http://www.amazon.com/exec/obidos/redirect-home/beejsguides-20
47. http://www.amazon.com/exec/obidos/ASIN/013490012X/beejsguides-20
48. http://www.amazon.com/exec/obidos/ASIN/0130810819/beejsguides-20
49. http://www.amazon.com/exec/obidos/ASIN/0130183806/beejsguides-20
50. http://www.amazon.com/exec/obidos/ASIN/0139738436/beejsguides-20
51. http://www.amazon.com/exec/obidos/ASIN/0138487146/beejsguides-20
52. http://www.amazon.com/exec/obidos/ASIN/0201633469/beejsguides-20

53. http://www.amazon.com/exec/obidos/ASIN/020163354X/beejsguides-20
54. http://www.amazon.com/exec/obidos/ASIN/0201634953/beejsguides-20
55. http://www.amazon.com/exec/obidos/ASIN/1565923227/beejsguides-20
56. http://www.amazon.com/exec/obidos/ASIN/0201563177/beejsguides-20
57. http://www.cs.umn.edu/~bentlema/unix/
58. http://www.ibrado.com/sock-faq/
59. http://pandonia.canberra.edu.au/ClientServer/
60. gopher://gopher-chem.ucdavis.edu/11/Index/Internet_aw/Intro_the_Internet/intro.to.ip/
61. http://www-iso8859-5.stack.net/pages/faqs/tcpip/tcpipfaq.html
62. http://tangentsoft.net/wskfaq/
63. http://www.rfc-editor.org/
64. http://www.rfc-editor.org/rfc/rfc768.txt
65. http://www.rfc-editor.org/rfc/rfc791.txt
66. http://www.rfc-editor.org/rfc/rfc793.txt
67. http://www.rfc-editor.org/rfc/rfc854.txt
68. http://www.rfc-editor.org/rfc/rfc951.txt
69. http://www.rfc-editor.org/rfc/rfc1350.txt
70. http://www.cygwin.com/
71. http://www.openssl.org/