

limit: coredumpsize: Can't set limit

Guide pour la programmation réseaux de Beej's

Utilisation des sockets Internet

Version 1.5.5 (13-Jan-1999)

[<http://www.ecst.csuchico.edu/~beej/guide/net>]

Intro

La programmation des sockets vous décourage? Est-ce que c'est un travail trop difficile à appréhender à partir des pages du manuel man? Vous voulez faire des programmes intéressants exploitants Internet, mais vous n'avez pas le temps d'examiner en détail une pagaille de structures pour savoir si vous devez appeler `bind()` avant `connect()`, etc., etc.

Hé bien devinez quoi! J'ai déjà fait une partie de ce sale boulot, et je souhaite le partager avec tout le monde! Vous êtes tombé au bon endroit. Ce document devrait offrir à tout programmeur standard en C les bases dont il ou elle a besoin pour comprendre tout ce bruit à propos des réseaux.

Audience

Ce document a été écrit comme un tutoriel, non comme une référence. C'est probablement le meilleur compromis pour des individus qui débutent avec la programmation des sockets et qui recherchent un guide. Ce n'est certainement pas un guide *complet* sur la programmation des sockets.

Heureusement, il devrait être juste suffisant pour donner un minimum de sens à ces pages du manuel... :-)

Plate-forme et Compilateur

La plupart du code source contenu dans ce document a été compilé sous Linux avec le compilateur Gnu'sgcc. Il a été aussi compilé avec succès en utilisant gcc sous HPUNIX. Notez que les extraits de code n'ont pas été testés individuellement.

Sommaire:

- [Qu'est-ce qu'une socket ?](#)
- [Deux types de socket Internet](#)
- [Inepties de bas niveau et théorie Réseau](#)
- [structs](#)--Connaissez les! ou bien les aliens détruiront la planète!
- [Convertissez les Natifs!](#)
- [Les adresses IP et comment les utiliser](#)
- [socket\(\)](#)--Récupérer un descripteur de fichier!
- [bind\(\)](#)--Sur quel port suis-je ?
- [connect\(\)](#)--Hé, vous!
- [listen\(\)](#)--Quelqu'un m'appellera t'il?
- [accept\(\)](#)--"Merci pour votre appel sur le port 3490."
- [send\(\) and recv\(\)](#)--Parle moi, chérie!
- [sendto\(\) and recvfrom\(\)](#)--Parle moi, style DGRAM

- [close\(\) and shutdown\(\)](#)--Hors de ma vue!
 - [getpeername\(\)](#)--Qui est tu?
 - [gethostname\(\)](#)--Qui je suis ?
 - [DNS](#)--Tu dis "whitehouse.gov", je dis "198.137.240.100"
 - [Client-Serveur en arrière plan](#)
 - [Un Serveur de Flux simple](#)
 - [Un Client de Flux Simple](#)
 - [Datagram Sockets](#)
 - [Blocage](#)
 - [select\(\)](#)--Entrées/sorties à Multiplexage Synchrone. Cool!
 - [Encore des références](#)
 - [Déclaration et sources d'aide](#)
-

Qu'est ce qu'un socket?

Si vous entendez parler de socket tout le temps et vous aimeriez savoir réellement ce que c'est. En utilisant des termes de la terminologie d'Unix c'est le moyen de parler à d'autres programmes en utilisant un descripteur standard de fichier.

Quoi ?

D'accord--vous avez peut être entendu des hackers Unix dire, "Simple, *tout* est fichier sous Unix!". Ce que voulait dire cette personne est que des qu'un programme sous Unix fait une opération d'entrée/sortie en fait il va lire ou écrire dans un descripteur de fichier. Un descripteur fichier est un simple entier associé à un fichier ouvert. Mais (et c'est là l'astuce) ce fichier peut être une connexion réseau, un FIFO, un tube, un terminal, un vrai fichier sur le disque, ou juste quelque chose d'autre. Tous sous Unix **est** un fichier! Alors lorsque vous voulez communiquer avec un autre programme par Internet, vous allez le faire par un descripteur de fichier, vous devez en être persuadé.

"Où est -ce que je trouve ce descripteur de fichier pour la communication réseau, Mr Jesaistout?" est probablement la dernière question que vous ayez à l'esprit maintenant, je vais y répondre : Vous faites un appel système appelé `socket()`. Cela vous retourne un descripteur de socket et vous communiquez grâce à celui-ci et grâce aux fonctions `send()` et `recv()`. ("[man send](#)", "[man recv](#)") appel système des sockets .

"Mais alors", devriez vous me rétorquer. "Si c'est un descripteur de fichier pourquoi diable je ne peux pas utiliser les appels systèmes `read()` et `write()` pour communiquer grâce à des sockets?" La réponse courte est "vous pouvez!", la longue est, "vous pouvez , mais `send()` et `recv()` offrent un meilleur contrôle en ce qui concerne la transmission de vos données."

Quoi d'autre? Que pensez vous de: il y a toutes sortes de sockets. Il y a les sockets avec DARPA Internet adresses (Sockets Internet), les noms de chemin vers un noeud (Sockets Unix), les adresses CCITT X.25 (les sockets X.25 sur lesquels vous pouvez tranquillement faire l'impasse) et probablement beaucoup d'autres qui dépendent de la version d'Unix que vous utilisez. Ce document traite seulement de la première sorte: Les sockets Internet.

Deux types de sockets Internet

Qu'est que c'est encore que ce truc? Il y a deux types de sockets Internet? Oui. Enfin non. Je suis en train de mentir. Il y en a bien plus, et je ne veux pas vous effrayer. Je vais uniquement vous parler de deux types ici. A l'exception de cette phrase, ou je vais vous parler des Sockets brutes (NDT: "*Raw Sockets*") qui sont très puissantes et sur lesquelles vous devriez jeter un oeil.

Bien, quels sont ces deux types ? L'un s'appelle les socket de flux (NDT: "*Stream Sockets*") et l'autre les sockets de paquets (NDT: "*Datagram Sockets*"). Elles seront référencées respectivement par `SOCK_STREAM` et `SOCK_DGRAM` dans la suite de ce document. Les "sockets de paquets" sont parfois appelées les "sockets sans

connections", (bien qu'elles puissent être connectées si vraiment vous insistez. Voir [connect\(.\)](#) pour plus d'info.)

Les sockets de flux sont deux voies de communications bi-directionnelles et fiables. Si vous envoyez deux éléments dans la socket dans l'ordre "1, 2", ces deux éléments arriveront dans le même ordre "1, 2" à l'autre bout de la connexion. Ils seront aussi sans erreurs. Toutes les erreurs que vous rencontrerez seront des prolongements de votre esprit dérangé et n'ont pas leur place dans cette discussion.

Qu'est ce qui utilise les sockets de flux?". Vous avez déjà sûrement entendu parler de l'application telnet, oui? Elle utilise les sockets de flux. Tous les caractères que vous tapez doivent arriver dans le même ordre que celui dans lequel ils ont été tapés, n'est ce pas? De même les navigateurs Web et les serveurs exploitant le protocole HTTP utilisent les sockets de flux pour recevoir des pages. Cela se vérifie si vous exécutez telnet sur un serveur WWW en spécifiant le port 80 et que vous tapez "GET pagename", alors vous aurez en retour la page HTML!

Comment les stream sockets atteignent elles ce haut niveau de qualité relatif à de la transmission de données? Elles utilisent un protocole appelé "The Transmission Control Protocol" plus connu sous le nom de "TCP". (Voir [RFC-793](#) pour (beaucoup) plus de détails sur TCP.). TCP s'assure que vos données arrivent séquentiellement et sans erreurs. Vous avez sûrement entendu "TCP" dans la première partie de "TCP/IP" où "IP" signifie "Internet Protocol" (Voir [RFC-791](#).) IP traite uniquement le routage Internet.

Chouette. Mais qu'en est il des des sockets de paquets? Pourquoi les appelle-t-on "sans connections"? Où est l'intérêt? Pourquoi ne sont-elles pas fiables? Voici quelques faits: si vous envoyez un paquet il pourrait arriver. Il pourrait arriver dans le désordre. Si il arrive les données dans le paquet seront exemptes d'erreurs.


Les sockets de paquets utilisent le protocole IP pour le routage, mais elles n'utilise pas le protocole TCP. Elles utilisent le protocole UDP (User Datagram Protoco) (Voir [RFC-768](#) pour plus de détails)

Pourquoi elles sont sans connections? D'une manière simpliste, c'est parce que vous n'avez pas à maintenir la connexion réseau ouverte comme avec les sockets de flux. Vous construisez seulement votre paquet, lui mettez une entête IP avec les informations pour le destinataire et envoyez le tout. Il n'y a pas besoin de connections. Ils sont généralement employés pour du transfert d'information par paquet. Des exemples d'applications: tftp, bootp, etc.

"Vous pourriez crier "Assez!". "Comment marchent ces programmes si des paquets peuvent être perdus?". La réponse est: chacun a son propre protocole au dessus d'UDP. Par exemple, le protocole de tftp indique que pour chaque paquet envoyé, le destinataire doit renvoyer un paquet qui indique, "je l'ai eu!" (un paquet "ACK".) Si l'expéditeur du paquet original n'obtient aucune réponse en par exemple cinq secondes il retransmettra le paquet jusqu'à ce qu'il obtienne finalement un ACK. Cette procédure d'acquisition est très importante en mettant en application des applications avec SOK_DGRAM.

Inepties de bas niveau et théorie Réseau

Puisque j'ai juste mentionné l'organisation multicouche des protocoles, il est temps de parler de comment les réseaux fonctionnent vraiment, et de montrer quelques exemples de la façon dont des paquets de SOK_DGRAM sont construits. En pratique, vous pouvez probablement sauter cette section. Cependant c'est une bonne culture de base.

 [\[L'encapsulation des Protocoles\]](#) Bon, les enfants,, il est temps d'apprendre ce qu'est l'**Encapsulation des données** C'est hyper important; à tel point que vous pourriez l'apprendre si vous suivez le cours réseaux ici à Chico State :-). A la base, la définition est: un paquet naît, ce paquet est emballé (encapsulé) avec une entête (et peut être une queue), par le premier protocole (disons le protocole TFTP), alors l'ensemble (entête TFTP inclut) est encapsulé de nouveau par le protocole suivant(disons UDP), idem avec le protocole IP, et de nouveau et finalement par la couche du protocole matériel (ou physique) (disons Ethernet).

Quand un autre ordinateur reçoit le paquet, le matériel enlève l'entête Ethernet, le noyau enlève les entêtes IP et UDP, le programme TFTP enlève l'entête TFTP et récupère finalement les données.

Je peux maintenant parler de l'horrible **Modèle de Réseaux Multicouches**. Ce modèle de réseau décrit un système de fonctionnalités réseaux qui a de nombreux avantages sur d'autres modèles. Par exemple, on peut écrire des programmes de sockets qui sont exactement les mêmes sans se soucier de la façon dont les données sont physiquement transportées (port série, Ethernet fin, AUI, quelconque) parce que les programmes de niveau inférieur s'en occupent pour vous. L'architecture matérielle et l'organisation du réseau est transparente pour le programmeur de sockets.

Sans plus de préambule, voici toute la structure éclatée du modèle. Rappelez vous en pour vos examens de cours de réseaux.

- Application
- Présentation
- Session
- Transport
- Réseau
- Lien de données
- Physique

La couche physique est la couche matériel (série, Ethernet, etc.). La couche application est aussi loin de la couche physique que vous pouvez l'imaginer-- c'est là que l'utilisateur interagit avec le réseau.

Maintenant, ce modèle est tellement général et vous pouvez l'utiliser comme un guide de réparation automobile si vous le vouliez vraiment. Un modèle multicouche plus proche d'Unix pourrait être:

- Couche Application(*telnet, ftp, etc.*)
- Couche de Transport Hôte à hôte (*TCP, UDP*)
- Couche Internet(*IP et routage*)
- Couche d'accès réseau(*Auparavant: Réseau, lien de données, et physique*)

Arrivés à ce stade, nous pouvons voir à quoi correspondent les couches lors de l'encapsulation des données originales.

Imaginez la quantité de travail nécessaire à la construction d'un seul paquet? Pfiou! et en plus, vous devez taper vous même les entêtes de paquet avec "cat"! Sans rire, tout ce que vous devez faire pour des sockets de flux est d'envoyer les données avec `send()`. Tout ce que vous devez faire pour des sockets de paquets est d'encapsuler le paquet avec la méthode de votre choix et d'appeler `sendto()`. Le noyau appelle alors la couche Transport et Internet à partir de vos données et le matériel appelle la couche d'accès réseau. Ah, la technologie moderne.

Voici la fin de la théorie réseau. Ah oui, j'ai oublié de vous parler de ce que je voulais vous dire sur la routage: Rien du tout C'est exact, je ne vais pas vous en parler du tout. Le routage consulte l'entête du paquet IP, consulte ses tables de routage, blablabla.... Regardez [IP RFC](#) si vous voulez en savoir plus. Cependant vous continuerez à vivre sans savoir comment ça marche.

structs

Nous voici arrivés, il est temps de parler un peu programmation. Dans cette section, je parlerai de divers types de données utilisées par les interfaces de sockets, puisque certaines sont des vraies saloperies à deviner.

D'abord le plus facile: le descripteur de socket. Un descripteur de socket est du type:

```
int
```

Juste un int standard.

Les choses deviennent bizarres à ce point, donc, lisez le tout et croyez en moi. Rappelez vous ceci: Il existe deux façons d'arranger les octets: octet de poids fort en premier ou bien octet de poids faible en premier. Le premier est appelé "Ordre d'Octets Réseau" (*NDT: "Network Byte Order"*). Certaines machines stockent leurs

nombre en interne dans l'ordre réseau d'autres non. Quand je dis que quelque chose doit être dans l'ordre NBO ("*Network Byte Order*") vous aurez à appeler une fonction (comme `htons()`) pour le transformer en "Ordre d'Octets Hôte" (NDT: "*Host Byte Order*"). Si je ne dis pas "NBO", alors il faut laisser la valeur dans l'ordre "HBO".

Ma première `Struct(TM)`--`struct sockaddr`. Cette structure contient les informations d'adresse de socket pour beaucoup de types de sockets:

```
struct sockaddr {
    unsigned short    sa_family;    /* famille d'adresse, AF_xxx      */
    char              sa_data[14]; /* 14 octets d'adresse de protocole */
};
```

`sa_family` peut être beaucoup de choses, mais ce sera "`AF_INET`" pour tout ce que nous faisons dans ce document. `sa_data` contient une adresse de destination et un numéro de port pour la socket. C'est plutôt lourd et gauche.

Pour utiliser la `struct sockaddr`, les développeurs ont créé une structure parallèle: `struct sockaddr_in` ("in" pour "Internet".)

```
struct sockaddr_in {
    short int          sin_family; /* Famille d'adresse              */
    unsigned short int sin_port;  /* Numéro de Port                 */
    struct in_addr     sin_addr;  /* Adresse Internet               */
    unsigned char      sin_zero[8]; /* Même taille que struct sockaddr */
};
```

Cette structure rend facile les références à des éléments de l'adresse de la socket. Noter que `sin_zero` (qui est inclus pour compléter la structure à la longueur d'un `struct sockaddr`) doit être initialiser avec des zéros à l'aide des fonctions `bzero()` ou `memset()`. De plus, et c'est la partie **importante**, un pointeur vers une `struct sockaddr_in` qui peut être instancié (cast) en un pointeur vers une `struct sockaddr` et vice-versa. Ainsi, même si `socket()` veut une `struct sockaddr *`, vous pouvez toujours utiliser une `struct sockaddr_in` et l'instancier à la dernière minute! Notez aussi que `sin_family` correspond à `sa_family` dans une `struct sockaddr` et doit être initialisé à "`AF_INET`". Finalement, les `sin_port` et `sin_addr` doivent être en **Network Byte Order**!

"Mais," dites vous, "comment une structure complète, `struct in_addr sin_addr`, peut-elle être en "Network Byte Order?" Cette question nécessite un examen détaillé de la structure `struct in_addr`, une des pires unions encore en vie:

```
/* Internet adresse (une structure pour des raisons historique) */
struct in_addr {
    unsigned long s_addr;
};
```

Auparavant *c'était* une union, mais tout cela est du passé. Bon débarras! Ainsi, si vous avez déclaré "`ina`" de type `struct sockaddr_in`, alors "`ina.sin_addr.s_addr`" référence les 4 octets de l'adresse IP (en "Network Byte Order"). Noter que si votre système utilise encore la bonne vieille et horrible union pour `struct in_addr`, vous pouvez toujours référencer les 4 octets de l'adresse IP exactement de la même manière que ci-dessus (ceci est du à des `#defines`.)

Convertissez les Natifs!

Nous allons maintenant tout droit dans la section suivante. Il y eu bien trop de discussion à propos de cet conversion d'ordre d'octets entre "réseau" ou "hôte"--l'heure de l'action est arrivée!

Il y a deux types que l'on peut convertir: `short` (deux octets) et `long` (quatre octets). Ces fonctions marchent aussi pour les version `unsigned`. Disons que l'on doit convertir un `short` de "Host Byte Order" en "Network Byte Order". Commencez avec "`h`" pour "hôte", suivi de "`to`", puis "`n`" pour "network", et "`s`" pour "short": `h-to-n-s`, ou `htons()` (Lire en anglais: "Host to Network Short").

C'est presque trop simple...

Vous pouvez utiliser n'importe quelle combinaison "n", "h", "s", et "l", en excluant celles qui sont vraiment idiotes. Par exemple, il n'y a PAS de fonction `stolh()` ("Short to Long Host")--pas dans cette soirée, en tous cas. Mais existent:

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

Maintenant que la chose s'éclaircie, on peut penser: "Que faire si l'ordre des octets doit être changé sur un char?" Puis on se dit, "Bah, inutile." On peut aussi se dire que si l'on a une machine à base de 68000 qui utilise déjà le "network byte order", il est inutile d'appeler `htonl()` pour les adresses IP. Le raisonnement est juste, MAIS les tentatives de portage vers une machine qui utilise un "network byte order" inverse, ferons capoter le programme. Soyez portable! C'est le monde Unix! Rappelez vous: Toujours mettre les octets dans l'ordre "Network Order" avant de les envoyer sur le réseau.

Un dernier point: pourquoi `sin_addr` et `sin_port` doivent être en "Network Byte Order" dans une `struct sockaddr_in`, et non `sin_family`? Réponse: `sin_addr` et `sin_port` sont encapsulés dans le paquet des couches IP et UDP, respectivement. Ainsi, ils doivent être en "Network Byte Order". Cependant, le champ `sin_family` est uniquement utilisé par le noyau pour déterminer quel type d'adresse la structure contient, il doit donc être en "Host Byte Order". De plus, puisque `sin_family` n'est PAS envoyé sur le réseau, il peut être en "Host Byte Order".

Les adresses IP et comment les utiliser

Heureusement pour vous, il y a plein de fonctions qui permettent de manipuler les adresses IP. Nul besoin de les retrouver manuellement et de les emballer dans un long avec l'opérateur `<<`.

D'abord, disons que vous avez une structure `struct sockaddr_in` et avez une adresse IP "132.241.5.10" que vous voulez mettre dedans. La fonction que vous devez utiliser est `inet_addr()`, qui convertit une adresse IP de la forme "chiffres-et-points" en un entier long non signé. L'assignation peut être faite de la manière suivante:

```
ina.sin_addr.s_addr = inet_addr("132.241.5.10");
```

Notez que `inet_addr()` retourne déjà l'adresse en "Network Byte Order"--vous n'avez pas à appeler `htonl()`. Malin!

Maintenant, le bout de code ci-dessus n'est pas très robuste parce qu'il ne contient pas de détection d'erreurs. `inet_addr()` retourne -1 en cas d'erreur. Vous rappelez vous des nombres binaires? (unsigned)-1 correspond justement à l'adresse IP 255.255.255.255! C'est l'adresse de "broadcast"! Mauvais plan. Il est donc important de vérifier proprement les erreurs.

Vous savez maintenant convertir les chaînes d'adresses IP en longs. Comment le faire dans l'autre sens? Que faire si l'on a une `struct in_addr` et que l'on veut l'imprimer sous la forme chiffres et points? Dans ce cas, il faut utiliser la fonction `inet_ntoa()` ("ntoa" signifie "network vers ascii") comme ceci:

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

Ceci imprimera l'adresse IP. Notez que `inet_ntoa()` prend une `struct in_addr` comme argument argument, et non un long. Remarquez aussi qu'elle retourne un pointeur vers un char. Il pointe vers vers un tableau statique de char dans `inet_ntoa()` de telle sorte que chaque appel vers `inet_ntoa()` écrasera la dernière adresse IP demandée. Par exemple:

```
char *a1, *a2;
.
.
```

```
a1 = inet_ntoa(ina1.sin_addr); /* soit 198.92.129.1 */
a2 = inet_ntoa(ina2.sin_addr); /* soit 132.241.5.10 */
printf("adresse 1: %s\n",a1);
printf("adresse 2: %s\n",a2);
```

will print:

```
adresse 1: 132.241.5.10
adresse 2: 132.241.5.10
```

Si vous devez sauvegarder l'adresse, utilisez `strcpy()` vers un autre tableau de caractères.

Nous avons fait le tour du sujet pour l'instant. Plus tard, vous apprendrez à convertir une chaîne comme "whitehouse.gov" en son adresse IP correspondante IP adresse (Voir [DNS](#), ci-dessous.)

socket()--Récupérer un descripteur de fichier!

Je ne peux pas laisser cela sous silence plus longtemps--Je dois maintenant vous parler de l'appel système `socket()`. Voici le détail:

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

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

Quels sont ces arguments? D'abord, `domain` doit être égal à "AF_INET", comme dans une struct `sockaddr_in` (ci-dessus.) Ensuite, l'argument `type` précise au noyau le type de socket: `SOCK_STREAM` ou `SOCK_DGRAM`. Enfin, mettre `protocol` à "0". (Notes: il y a beaucoup plus de domaines que ceux listés. Il y a beaucoup d'autres types que ceux cités. Voir la page de manuel [socket\(\)](#). Il y a aussi une meilleure méthode pour obtenir le protocol. Voir la page de manuel [getprotobyname\(\)](#).)

`socket()` retourne simplement un descripteur de socket qui peut être utilisé plus tard dans des appels systèmes, ou bien -1 en cas d'erreur. La variable globale `errno` prend la valeur de l'erreur (voir la page de manuel [perror\(\)](#).)

bind()--Sur quel port suis-je ?

Une fois que vous avez une socket, vous devez associer la socket avec un port de votre machine locale.(C'est la procédure classique si vous voulez faire un `listen()` pour avoir une connexion sur un port spécifique. MUDDS fait cela quand il vous demande de "telnet to x.y.z port 6969"). Si vous allez faire uniquement un `connect()`, il n'est pas nécessaire de faire de `bind`. Lisez quand même ce chapitre, juste pour ne pas vous endormir.

Voici le scénario pour l'appel système `bind()`:

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

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

`sockfd` est le descripteur de socket retourné par `socket()`. `my_addr` est un pointeur sur la struct `sockaddr` qui contient les informations à propos de votre adresse, des noms, du port et l'adresse IP . `addrlen` peut être mis à la valeur de `sizeof(struct sockaddr)`.

Pfou. C'est un peu gros pour une seule bouchée. Prenons un exemple:

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

```
#define MYPORT 3490

main()
{
    int sockfd;
    struct sockaddr_in my_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0); /* Contrôle d'erreur! */

    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("132.241.5.10");
    bzero(&(my_addr.sin_zero), 8); /* zéro pour le reste de la struct */

    /* ne pas oublier les test d'erreur pour bind(): */
    bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
    .
    .
    .
}
```

Il y a quelques points remarquables ici. `my_addr.sin_port` est en Network Byte Order. De même que `my_addr.sin_addr.s_addr`. Un autre point à vérifier est que le fichier d'entête peut différer d'un système à un autre. Pour en être sûr, lire les pages de manuel locales.

Finalement, à propos de `bind()`, je dois mentionner que le mécanisme pour avoir votre propre adresse IP et/ou le port peut se faire automatiquement:

```
my_addr.sin_port = 0; /* choose an unused port at random */
my_addr.sin_addr.s_addr = INADDR_ANY; /* use my IP adresse */
```

En mettant `my_addr.sin_port` à zéro, on demande à `bind()` de choisir le port pour nous. De même, en affectant la valeur `INADDR_ANY` à `my_addr.sin_addr.s_addr`, on demande que l'adresse IP de notre machine soit mise automatiquement.

Si vous êtes pointilleux, vous aurez peut être remarqué que je n'ai pas mis `INADDR_ANY` en "Network Byte Order"! Vilain garçon. Toutefois, j'ai des infos de première main: `INADDR_ANY` est vraiment à zéro! Zéro vaut toujours zéro quel que soit l'ordre des octets. Cependant, les puristes feront remarquer qu'il pourrait exister une dimension parallèle où `INADDR_ANY` est, disons, 12 et que mon code ne marchera pas là bas. Ça me convient parfaitement:

```
my_addr.sin_port = htons(0); /* choose an unused port at random */
my_addr.sin_addr.s_addr = htonl(INADDR_ANY); /* use my IP adresse */
```

Maintenant nous sommes tellement portables que vous ne voudriez pas le croire. Je voulais juste souligner cela, puisque la plupart du code que vous allez rencontrer ne s'encombrera pas d'appeler `INADDR_ANY` avec `htonl()`.

`bind()` retourne lui aussi -1 si une erreur se produit et met `errno` à la valeur de l'erreur système.

Une autre chose, fait attention quand vous appelez `bind()`: ne mettez pas n'importe quel numéro de port. Tous les ports en dessous de 1024 sont réservés. Vous pouvez utiliser n'importe quel port au dessus de 1024 jusqu'à 65535 (prenez un port qui n'est pas utilisé par un autre programme).

Une dernière remarque finale à propos de `bind()`: Il y aura des cas où vous n'aurez absolument pas besoin de l'appeler. Si vous vous connectez à une machine distante et que vous n'avez pas à faire attention au port auquel vous vous connectez (comme avec `telnet`), vous pouvez simplement appeler `connect()`, il verra si le socket n'est pas attaché et il fera un `bind()` vers un port local inutilisé.

connect()--Hé, vous!

Supposons juste quelques instants que vous êtes une application telnet. Votre utilisateur vous ordonne (comme dans le film *TRON*) d'obtenir un descripteur de fichier. Vous acceptez et appelez `socket()`. Ensuite, l'utilisateur vous demande de vous connecter à "132.241.5.10" sur le port "23" (le port standard telnet.) Oh mon dieu! Que faites vous maintenant?

Heureusement pour vous, programme, vous prenez connaissance du chapitre dédié à `connect()`--comment se connecter à un hôte distant. Vous le lisez avidement afin de ne pas décevoir votre utilisateur...

L'appel à `connect()` se fait comme suit:

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

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

`sockfd` est notre ami le voisin de descripteur de socket, tel que retourné par l'appel à `socket()`, `serv_addr` est une struct `sockaddr` contenant le port de destination et l'adresse IP, et `addrlen` sera `sizeof(struct sockaddr)`.

Est-ce que cela prend un peu plus de sens? Regardons un exemple:

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

#define DEST_IP    "132.241.5.10"
#define DEST_PORT  23

main()
{
    int sockfd;
    struct sockaddr_in dest_addr;    /* Contiendra l'adresse de destination */

    sockfd = socket(AF_INET, SOCK_STREAM, 0); /* Vérification d'erreurs! */

    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);
    bzero(&(dest_addr.sin_zero), 8);         /* zéro pour le reste de la struct */

    /* ne pas oublier les tests d'erreur pour connect()! */
    connect(sockfd, (struct sockaddr *)&dest_addr, sizeof(struct sockaddr));
    .
    .
    .
}
```

De nouveau, vérifiez la valeur de retour de `connect()`-- Cette fonction retournera -1 si une erreur arrive et mettra à jour la variable `errno`.

Remarquez aussi, que nous n'avons pas appelé `bind()`. Nous n'avons pas à nous soucier du numéro du port; La seule chose qui nous importe est où nous allons nous connecter. Le système choisira un port pour nous, et le site sur lequel nous allons être connecté recevra automatiquement cette information. Pas d'inquiétude.

listen()--Est-ce que quelqu'un va m'appeler ?

Ok, il est temps de changer d'allure. Que se passe t-il si vous ne voulez pas vous connecter à un hôte distant? Disons juste pour voir que vous souhaitez attendre des connections entrantes et les traiter d'une manière quelconque. Ce processus se fait en deux étapes: d'abord vous écoutez avec `listen()`, puis vous utilisez `accept()` (voir ci-dessous.)

L'appel système `listen()` est tres simple, cependant il nécessite une petite explication:

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

`sockfd` est l'habituel descripteur de fichier socket issu de l'appel système `socket()`. `backlog` est le nombre de connections autorisées dans la file entrante. Qu'est ce que cela signifie? Que les connections vont attendre dans cette file jusqu'à ce que vous les acceptiez avec `accept()` (voir ci-dessous) et ceci est la limite du nombre autorisé à faire la queue. La plupart des systèmes limitent en silence ce nombre à environ 20; une valeur raisonnable tournera autour de 5 ou 10.

Idem comme les autres appels systèmes, `listen()` retournera -1 et mettra à jour la variable `errno` dans le cas d'une erreur.

Bien, comme vous pouvez l'imaginer, nous devons appeler `bind()` avant d'appeler `listen()` ou sinon le système va écouter sur un port au hasard. Bleah! alors si nous allons écouter pour une connection entrante, la suite d'appel système que vous devez faire est:

```
socket();
bind();
listen();
/* accept() goes here */
```

Ceci restera juste un exemple brut puisqu'il est auto-explicatif. (Le code de la section `accept()`, ci-dessous, est plus complet.) La partie vraiment difficile de tout ce sha-bang est l'appel d'`accept()`.

`accept()`--"Merci d'appeler le port 3490."

A vos marques--l'appel d'`accept()` est vraiment dingue! Voici ce qui va se passer: quelqu'un de très très loin va essayer de se connecter avec `connect()` à votre machine sur un port que vous écoutez avec `listen()`. Sa connection va faire la queue en attendant d'être acceptée avec `accept()`. Vous appelez `accept()` et lui dites de récupérer la connection en attente. Il vous retournera un *nouveau descripteur de fichier socket* à utiliser pour cette seule connection! C'est vrai, d'un coup, vous avez *deux descripteurs de fichier socket* pour le prix d'un! L'original écoute toujours votre port et le nouveau est prêt à faire des `isend()` et `recv()`. C'est tout!

L'appel se fait comme suit:

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

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

`sockfd` est le descripteur de socket à écouter avec `listen()`. Relativement facile. `addr` est habituellement un pointeur vers une struct `sockaddr_in` locale. C'est à cet endroit que se trouve l'information concernant la connection entrante (et il est possible de déterminer quel hôte appelle sur quel port). `addrlen` est une variable entière locale qui doit contenir `sizeof(struct sockaddr_in)` avant que son adresse soit passée à `accept()`. `Accept` ne mettra pas plus d'octets dans `addr`. Si elle en met moins, elle changera la valeur de `addrlen` pour l'indiquer.

Devinez quoi? `accept()` retourne -1 et met à jour `errno` si une erreur arrive. Je vous parie que vous ne l'aviez pas deviné.

Comme précédemment, il y a beaucoup de chose à apprendre d'un coup, c'est pourquoi, voici un morceau de code pour vos études:

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

#define MYPORT 3490    /* le port de connection pour les utilisateurs */

#define BACKLOG 10     /* Le nombre maxi de connections en attente */

main()
{
```

```

int sockfd, new_fd; /* Écouter sur sockfd, nouvelle connection sur new_fd */
struct sockaddr_in my_addr; /* Informations d'adresse */
struct sockaddr_in their_addr; /* Informations d'adresse du client */
int sin_size;

sockfd = socket(AF_INET, SOCK_STREAM, 0); /* Contrôle d'erreur! */

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; /* auto-remplissage avec mon IP */
bzero(&(my_addr.sin_zero), 8); /* zero pour le reste de struct */

/* ne pas oublier les contrôles d'erreur pour ces appels: */
bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));

listen(sockfd, BACKLOG);

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

```

De nouveau, remarquez que nous allons utiliser un descripteur de socket `new_fd` pour tous les appels à `send()` et `recv()`. Si vous n'acceptez qu'une seule connection, vous pouvez fermer avec `close()` la `sockfd` originale afin d'empêcher d'autres connections sur le même port, si cela est votre souhait.

send() and recv()--Parle moi, chérie!

Ces deux fonctions servent à la communication pour les stream sockets ou les datagram sockets connectés. Si vous voulez utiliser les "unconnected" datagram sockets, vous devriez aller regarder le chapitre sur [sendto\(\)](#) et [recvfrom\(\)](#).

L'appel système `send()` :

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

`sockfd` est un descripteur de socket par lequel vous voulez envoyer des données (que ce soit celui retourné par `socket()` ou celui que vous avez obtenu avec `accept()`.) `msg` est un pointeur vers les données à envoyer, et `len` est la longueur des données en octets. Mettez juste `flags` à 0. (Voir la page de manuel [send\(\)](#) pour plus d'informations à propos des drapeaux (*NDT:flags*.)

Un code typique pourrait être:

```

char *msg = "Beej était là!";
int len, bytes_sent;
.
.
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
.
.
.

```

`send()` retourne le nombre d'octets envoyés --**Ceci peut très bien être moins que le nombre que vous lui avez demandé d'envoyer!** Parfois, vous lui demandez d'envoyer une grosse bouchée de données et il ne peut simplement pas s'en dépêtrer. Il en enverra autant que possible et se reposera sur vous pour envoyer le reste plus tard. Rappeler vous que si la longueur retournée par `send()` ne correspond pas à la valeur de `len`, il vous revient d'envoyer le reste de la chaîne. La bonne nouvelle est: si le paquet est petit (moins d'1 K à peu près) il se débrouillera *probablement* pour envoyer tout d'un coup. Là encore, -1 est retourné en cas d'erreur, et `errno` contient le numéro d'erreur.

L'appel de `recv()` est très similaire:

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

`sockfd` est le descripteur de socket sur lequel s'effectue la lecture, `buf` est le tampon où lire l'information, `len` est la longueur maximale du tampon, et `flags` peut encore être mis à 0. (Voir la page du manuel de [recv\(\)](#) [man page](#) pour des informations sur les drapeaux.)

`recv()` retourne le nombre d'octets lu dans le tampon, où -1 si il y a une erreur (avec `errno` mis à jour)

C'était facile n'est ce pas? Vous pouvez maintenant passer des données dans les deux sens sur des sockets de flux! Super! Vous êtes un programmeur Réseau Unix!

sendto() and recvfrom()--Parles moi, style DGRAM

Vous me direz "C'est parfait et propre, mais qu'en est-il des unconnected datagram sockets?" No problemo, amigo. Nous avons juste ce qu'il vous faut.

Puisque les sockets de paquets ne sont pas connectés à un hôte distant, devinez un peu quelle information nous devons donner avant d'envoyer un paquet? Exactement! L'adresse de destination! Voilà le scoop:

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

Comme vous pouvez le voir, cet appel est à la base le même que `send()` auquel on a ajouté deux autres paramètres. `to` est un pointeur vers une `struct sockaddr` (que vous avez probablement sous la forme d'une `struct sockaddr_in` et qu'il faudra instancier à la dernière minute) et qui contient l'adresse de destination et le port. `tolen` peut simplement contenir `sizeof(struct sockaddr)`.

Tout comme `send()`, `sendto()` retourne le nombre d'octets réellement envoyés (qui, je le répète, peut être inférieur au nombre d'octets que vous lui avez demandé d'envoyer), ou -1 en cas d'erreur.

Les fonctions `recv()` et `recvfrom()` ont le même comportement. L'appel de `recvfrom()` se fait comme suit:

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

Ici aussi c'est comme `recv()` agrémenté de deux paramètres. `from` est un pointeur vers une `struct sockaddr` locale qui sera remplie avec l'adresse IP et le port de la machine émettrice. `fromlen` est un pointeur vers un `int` local qui devra être initialisé à `sizeof(struct sockaddr)`. Au retour de la fonction, `fromlen` contiendra la longueur de l'adresse réellement stockée dans `from`.

`recvfrom()` retourne le nombre d'octets reçus, ou -1 en cas d'erreur (avec `errno` mis à jour en conséquence.)

Souvenez vous, si vous utilisez `connect()` pour une socket datagram, vous pouvez alors utiliser simplement `send()` et `recv()` pour toutes vos transactions réseaux. La socket restera elle même une socket datagram et le paquet utilisera le protocole UDP, mais le système ajoutera automatiquement les informations relatives à la destination et à la source pour vous.

close() and shutdown()--Hors de ma vue!

Voilà! Vous avez envoyé avec `send()` et reçu avec `recv()` des données pendant toute la journée et cela vous suffit. Vous êtes prêt à fermer la connection de votre descripteur de socket. c'est facile il suffit d'utiliser la fonction usuelle de clôture de descripteur de fichier Unix `close()`:

```
close(sockfd);
```

Ceci empêchera toutes écritures ou lectures futures sur la socket. Toute tentative de lecture ou d'écriture sur cette socket provoquera une erreur.

Au cas où vous souhaiteriez un peu plus de contrôle sur le processus de clôture de la socket, vous pouvez utiliser la fonction `shutdown()`. Elle vous permet de couper la communication dans un sens précis, ou les deux (comme le fait `close()`.) Prototype:

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

`sockfd` est le descripteur de fichier socket à fermer, et `how` est à choisir parmi:

- 0 - Réceptions interdites
- 1 - Envois interdits
- 2 - Réceptions et Envois interdits (comme `close()`)

`shutdown()` retourne 0 en cas de succès, et -1 en cas d'erreur (avec `errno` mis à jour.)

Si vous deignez utiliser `shutdown()` sur des "unconnected datagram sockets", elle rendra simplement la socket indisponible pour de futurs appels à `send()` et `recv()` (Rappelez vous que vous pouvez les utiliser si vous connectez avec `connect()` vos datagram socket.)

Rien à faire.

getpeername()--Qui êtes vous?

Cette fonction est trop facile.

C'est tellement simple, que je ne lui ai pas accordé son propre chapitre. Le voila quand même.

La fonction `getpeername()` vous dira qui est de l'autre cote de la connection. Prototype:

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

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

`sockfd` est le descripteur de la "connected stream socket", `addr` est un pointeur vers une struct `sockaddr` (ou une struct `sockaddr_in`) qui contiendra des informations sur l'autre bout de la connection, et `addrlen` est un pointeur vers un int, qui doit être initialisé à `sizeof(struct sockaddr)`.

Cette fonction retourne -1 si il y a une erreur et met à jour la variable `errno`.

Une fois que vous avez l'adresse, vous pouvez utiliser `inet_ntoa()` et `gethostbyaddr()` pour avoir plus d'informations. Non, vous ne pouvez pas connaître leur login. (Ok, ok. Si l'autre ordinateur fait tourner un démon Ident, c'est possible. Toutefois, ceci est hors sujet, ici voir [RFC-1413](https://tools.ietf.org/html/rfc1413) pour en savoir plus.)

gethostname()--Qui suis-je?

Encore plus facile que `getpeername()`: la fonction `gethostname()`. Cela vous retourne le nom de l'ordinateur sur lequel votre programme tourne. Le nom peut alors être utilisé avec `gethostbyname()`, pour déterminer l'adresse IP de votre machine.

Quoi de plus amusant? J'ai bien quelques idées mais cela ne concerne pas la programmation de sockets. En tous cas, voici le détail:

```
#include <unistd.h>

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

Les arguments sont simple: `hostname` est un pointeur vers un tableau de chars qui contiendra le hostname lors du retour de la fonction, et `size` est la longueur en octets du tableau `hostname`

Cette fonction retourne 0 en cas de succès, et -1 pour une erreur, mettant errno à jour comme d'habitude.

DNS--Vous dites "whitehouse.gov", Je dis "198.137.240.100"

Dans le cas où vous ne savez pas ce qu'est un DNS, cela veut dire un "Domain Name Service". Simplement, vous lui donnez un nom (humainement compréhensible) et il vous donne l'adresse IP, alors vous pouvez l'utiliser dans les fonctions bind(), connect(), sendto(), ou pour ce que vous voulez. De cette façon, quand quelqu'un entre:

```
$ telnet whitehouse.gov
```

telnet est capable de trouver qu'il faut faire un connect() sur "198.137.240.100".

Mais comment cela marche-t-il?, regardons l'utilisation de la fonction gethostbyname():

```
#include <netdb.h>

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

Comme vous pouvez le voir, cela renvoie un pointeur sur une struct hostent, ..., dont le prototype est comme suit:

```
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]
```

Et voici les descriptions des champs de la struct hostent:

- h_name - Nom Officiel de l'hôte.
- h_aliases - Tableau terminé par NULL de noms alternatives pour l'hôte.
- h_addrtype - Le Type d'adresse retourné; normalement AF_INET.
- h_length - La longueur de l'adresse en octets.
- h_addr_list - Tableau terminé par zéro d'adresses réseau pour l'hôte. l'adresse de l'hôte est en Network Byte Order.
- h_addr - La première adresse dans h_addr_list.

gethostbyname() retourne un pointeur sur une structure struct hostent, ou NULL si il y a erreur. (Mais errno **n'est pas mis à jour** --h_errno indique l'erreur à sa place. Voir perror(), ci-dessous.)

Mais comment cela s'utilise-t-il? Parfois (on le découvre en lisant des manuels d'ordinateurs), présenter les informations à l'utilisateur n'est pas suffisant. cette fonction est certainement plus facile à utiliser que cela en a l'air.

[Voici un programme à titre d'exemple:](#)

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>

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

    if (argc != 2) { /* Vérification d'erreurs de la ligne de commande */
```

```

    fprintf(stderr, "usage: getip adresse\n");
    exit(1);
}

if ((h=gethostbyname(argv[1])) == NULL) { /* récupérer infos de l'hôte */
    perror("gethostbyname");
    exit(1);
}

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

return 0;
}

```

Avec `gethostbyname()`, on ne peut pas utiliser `perror()` pour imprimer le message d'erreur (puisque `errno` n'est pas utilisé). Au lieu de cela, appelez `herror()`.

Finalement c'est plutôt simple. On passe simplement la chaîne qui contient le nom de la machine ("whitehouse.gov") à `gethostbyname()`, puis on récupère l'information dans la struct `hostent` retournée.

La seule source de problèmes possible est l'impression de l'adresse IP ci-dessus. `h->h_addr` est un `char *`, mais `inet_ntoa()` veut qu'on lui passe une struct `in_addr`. Il faut donc instancier (cast) `h->h_addr` en une struct `in_addr *`, puis accéder aux données.

Client-Serveur en arrière plan

Bienvenu dans le monde du client-serveur. Ce chapitre traitera de tous ce qui concerne le dialogue réseau entre le client et le serveur. Par exemple, prenez le programme `telnet`. Lorsque vous vous connectez à un site distant sur le port 23 avec `telnet` (en tant que client), un programme sur le site distant (appelé `telnetd`, le serveur) se réveillera pour traiter la demande arrivant, et enverra le prompt login etc.

 [\[Client-Server Relationship\]](#)
Figure 2. La relation Client-Serveur.

Les informations sur les échanges entre le client et le serveur sont résumées dans la figure 2.

Notez que la discours entre le client et le serveur peut se faire en `SOCK_STREAM`, `SOCK_DGRAM`, ou autre chose du moment qu'il parle le même langage. Quelques exemples de programmes client-serveur: `telnet/telnetd`, `ftp/ftpd`, or `bootp/bootpd`. A chaque fois que l'on utilise `ftp`, il y a un programme `ftpd` qui est à votre service.

Souvent, il n'y aura qu'un serveur sur la machine, et ce serveur gérera plusieurs clients grâce à `fork()`. La méthode classique est: Le serveur attends une demande de connection, l' `accept()` et `fork()` un processus fils pour traiter la demande. C'est ce que fait notre serveur dans le prochain chapitre.

Un Serveur de Flux Simple

Tout ce que fait ce serveur est d'envoyer une chaîne "Hello, world!\n" au client. Tout ce que nous avons besoin de tester est de le faire tourner dans une fenêtre et de faire un `telnet` dans une autre:

```
$ telnet remotehostname 3490
```

Ou `remotehostname` est le nom de la machine que vous utilisez.

[Le code du serveur](#): (Nota: un backslash en fin de ligne indique que la ligne se continue sur la suivante.)

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

```

```

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

#define MYPORT 3490    /* Le port où les utilisateurs se connecteront */

#define BACKLOG 10     /* Nombre maxi de connections acceptées en file */

main()
{
    int sockfd, new_fd; /* Ecouter sock_fd, nouvelle connection sur new_fd */
    struct sockaddr_in my_addr; /* Adresse */
    struct sockaddr_in their_addr; /* Adresse du connecté */
    int sin_size;

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 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; /* auto-remplissage avec mon IP */
    bzero(&(my_addr.sin_zero), 8); /* zero pour le reste de struct */

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

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

    while(1) { /* main accept() loop */
        sin_size = sizeof(struct sockaddr_in);
        if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, \
                             &sin_size)) == -1) {
            perror("accept");
            continue;
        }
        printf("serveur: Reçu connection de %s\n", \
               inet_ntoa(their_addr.sin_addr));
        if (!fork()) { /* processus fils */
            if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
                perror("send");
            close(new_fd);
            exit(0);
        }
        close(new_fd); /* Le parent n'a pas besoin de cela */

        while(waitpid(-1, NULL, WNOHANG) > 0); /* Nettoyage des processus fils */
    }
}

```

Dans le cas où vous seriez curieux, J'ai le code dans une grosse fonction `main()` pour des raisons de clarté syntaxiques (à mon avis). N'hésitez pas à le découper en fonctions plus petites si cela vous convient mieux.

Vous pouvez aussi obtenir la chaîne venant du serveur en utilisant le programme client fourni dans le chapitre suivant.

Un Client Simple de flux

Ceci est aussi simple que le serveur. Tout ce que fait ce client est de se connecter à une machine en spécifiant le port sur la ligne de commande, port 3490. Il prends alors la chaîne envoyée par le serveur.

Les sources du client:

```
#include <stdio.h>
#include <stdlib.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      /* Le port où le client se connectera */

#define MAXDATASIZE 100 /* Tampon d'entrée */

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in their_addr; /* Adresse de celui qui se connecte */

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

    if ((he=gethostbyname(argv[1])) == NULL) { /* Info de l'hôte */
        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);
    bzero(&(their_addr.sin_zero), 8); /* zero pour le reste de struct */

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

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

    buf[numbytes] = '\0';

    printf("Reçu: %s", buf);

    close(sockfd);

    return 0;
}
```

Remarquez que si vous n'exécutez pas le serveur avant d'envoyer le client, connect() retournera "Connection refused". C'est très utile.

Datagram Sockets

Je n'ai pas grand chose à dire là-dessus, je vais juste présenter deux programmes d'exemples: `talker.c` et `listener.c`.

`listener` se met à l'écoute et en attente sur le port 4950 sur une machine. `talker` envoie un paquet sur ce port, sur une machine spécifique que l'utilisateur détermine sur la ligne de commande.

Voici [le code source pour listener.c](#):

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

#define MYPORT 4950    /* Le port de connection pour l'utilisateur */

#define MAXBUFLEN 100

main()
{
    int sockfd;
    struct sockaddr_in my_addr;    /* mon adresse */
    struct sockaddr_in their_addr; /* Adresse du connecté */
    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; /* auto-fill with my IP */
    bzero(&(my_addr.sin_zero), 8);    /* zero pour le reste de struct */

    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, 0, \
        (struct sockaddr *)&their_addr, &addr_len)) == -1) {
        perror("recvfrom");
        exit(1);
    }

    printf("reçu un paquet de %s\n",inet_ntoa(their_addr.sin_addr));
    printf("le paquet fait %d octets de long\n",numbytes);
    buf[numbytes] = '\0';
    printf("Le paquet contient \"%s\"\n",buf);

    close(sockfd);
}
```

Remarquez que dans votre appel à `socket()` vous utilisez `SOCK_DGRAM`. Notez aussi qu'il n'y a nul besoin de `listen()` ou `accept()`. C'est l'un des petits profits d'utiliser les "unconnected datagram sockets"!

Voici maintenant [le code source pour talker.c](#):

```

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

#define MYPORT 4950      /* Le port de connection */

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; /* adresse du connecté */
    struct hostent *he;
    int numbytes;

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

    if ((he=gethostbyname(argv[1])) == NULL) { /* Récupérer l'info de l'hôte */
        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);
    bzero(&(their_addr.sin_zero), 8);     /* zero pour le reste de struct */

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

    printf("Envoyé %d octets à %s\n", numbytes, inet_ntoa(their_addr.sin_addr));

    close(sockfd);

    return 0;
}

```

Compilez les deux programmes, exécutez `listener` sur une machine, et `talker` sur une autre. Regardez les communiquer! Super amusement de top niveau pour toute la famille nucléaire!

Mis a part d'un petit détail que je vous avais mentionné plusieurs fois: les connected datagram sockets. Je me dois de vous en parler maintenant que nous sommes dans le chapitre datagram. Disons que `talker` appelle `connect()` et spécifie l'adresse de `listener`. A partir de ce point, `talker` ne peut plus envoyer et recevoir que depuis cette adresse spécifiée par `connect()`. Pour cette raison, vous n'avez pas à utiliser `sendto()` et `recvfrom()`; vous pouvez simplement utiliser `send()` et `recv()`.

Blocage

Vous avez déjà entendu parler de cela, mais qu'est ce qui se cache derrière? Pour faire court, "bloquer" est un jargon technique pour "dormir". Vous avez probablement remarqué que lors de l'exécution de `listener`, ci-

dessus, il attend qu'un paquet arrive. Il se passe qu'il a appelé `recvfrom()`, qu'il n'y avait pas de données et que par conséquent `recvfrom()` "bloque" (soit dors ici) jusqu'à l'arrivée de données.

Beaucoup de fonctions sont bloquant-es. `accept()` est bloquante ainsi que toutes les fonctions `recv*()`. La raison en est simple; c'est parce que ces fonctions y sont autorisées. Lors de la création de la socket grâce à l'appel `socket()`, le système la positionne en tant que descripteur bloquant. Si vous ne souhaitez pas qu'elle le soit, vous devez appeler la fonction `fcntl()`:

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

En déclarant une socket non-bloquante, vous pouvez effectivement l'interroger périodiquement. Si vous essayez de lire sur un socket non-bloquante et qu'il n'y a pas de donnée à lire, la fonction vous retournera `-1` et `errno` sera mis à `EWOULDBLOCK`.

D'une manière générale, ce type d'interrogation est plutôt une mauvaise idée. Si vous faites un programme qui attends les données sur une socket non bloquante, vous allez prendre beaucoup de ressource système inutilement. Une manière plus élégante pour voir s'il y a des données qui sont arrivées est d'utiliser la fonction `select()` que nous allons décrire dans le prochain chapitre.

`select()`--Multiplexage Synchrone d'E/S

Cette fonction a quelque chose d'étrange, mais elle est TRES UTILE. Prenez la situation suivante: vous êtes un serveur et vous voulez attendre une connexion entrante mais aussi garder la connection que vous avez déjà.

Pas de problème vous vous dites, un simple `accept()` et deux appels à `recv()`. Pas si vite! Que se passera t-il s'il y a un blocage sur un `accept()`? Comment allez vous recevoir des données avec `recv()` en même temps? Utilisez des sockets non_bloquante! Pas question! Vous ne voulez pas être un consommateur record de CPU? Quoi alors ?

`select()` vous offre le possibilité de surveiller plusieurs sockets en même temps. Cela va vous permettre de savoir laquelle est prête à la lecture ou à l'écriture et si une exception se produit sur une socket si vous voulez réellement le savoir.

Sans plus de palabres, voici le prototype de `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);
```

La fonction surveille des ensembles de descripteurs de fichiers et plus particulièrement `readfds`, `writefds`, et `exceptfds`. (On parlera d'ensemble pour parler d'ensemble de descripteur de fichier) Si vous voulez voir si vous pouvez lire à partir de l'entrée standard et sur quelques sockets, ajoutez les descripteurs de fichiers `0` et `sockfd` à l'ensemble `readfds`. Le paramètre `numfds` doit être mis à la valeur du fichier le plus haut plus un. Dans cet exemple, il doit être mis à `sockfd+1`, assurant ainsi qu'il est supérieur à l'entrée standard (`0`).

Quand `select()` se termine, `readfds` sera modifié avec la fonction réflexe disant que le descripteur de fichier que vous avez sélectionné est prêt à la lecture. Vous pouvez les tester avec les macros `FD_ISSET()`, suivantes.

Avant d'aller plus avan , je vais vous expliquer comment manipuler ces ensembles. Chaque ensemble est de type `fd_set`. Les macros suivantes agissent sur ce type:

- `FD_ZERO(fd_set *set)` - initialise l'ensemble
- `FD_SET(int fd, fd_set *set)` - Ajoute `fd` à l'ensemble
- `FD_CLR(int fd, fd_set *set)` - Elimine `fd` de l'ensemble
- `FD_ISSET(int fd, fd_set *set)` - test pour savoir si `fd` fait parti de l'ensemble

Finalement, qu'est ce que c'est que cette struct `timeval` sortie de la jungle? Bien, il y a des moment ou vous ne voulez pas attendre une éternité que quelqu'un vous envoi des données. Peut être que toutes les 96 secondes vous voulez afficher "En cours..." sur le terminal même si rien ne se passe. Cette structure dédiée au temps vous autorise à spécifier un `TIMEOUT`. Si cette période s'achève et que `select()` n'a pas trouvé de descripteurs prêt, alors la fonction prendra la main et le programme continuera.

La struct `timeval` contient les champs suivants:

```
struct timeval {
    int tv_sec;      /* secondes */
    int tv_usec;     /* microsecondes */
};
```

Affectez à `tv_sec` le nombre de secondes à attendre, et à `tv_usec` le nombre de micro-secondes. Oui, ce sont des *microsecondes*, et non des millisecondes. Il y a 1000 micro-secondes dans une millisecondes et 1 000 000 micro-secondes dans une secondes. Pour alors cela s'appelle t-il "usec" ? Le "u" est en fait la lettre grec Mu utilisée ici pour dire micro. Alors, au retour de la fonction, `timeout` *peut* être mit à jour pour indiquer combien de temps il reste. Cela dépend de la couleur de l'Unix que vous utilisez.

Yeah! Nous avons un chronomètre précis à la micro-seconde! Mais ne comptez pas trop dessus. Les Unix standards ont une résolution temporelle de 100 millisecondes, alors vous aurez probablement à attendre au moins ce temps, même si vous réglez votre struct `timeval` à une valeur plus petite.

Autre point intéressant: Si vous positionnez les champs de votre structure struct `timeval` à 0, `select()` fera un timeout immédiatement, et interrogera tous les descripteurs de fichier dans vos ensembles. Si vous mettez le paramètre `timeout` a `NULL`, il ne fera jamais de timeout, et attendra que le premier descripteur soit prêt. Finalement, si vous n'êtes pas intéressé par une attente quelconque, vous pouvez passez à `select()` , `NULL` comme paramètre.

[L'exemple suivant](#) attends 2.5 secondes que quelques choses arrive sur l'entrée standard:

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

#define STDIN 0 /* Descripteur de fichier pour entrée standard */

main()
{
    struct timeval tv;
    fd_set readfds;

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

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

    /* ignorer writefds et exceptfds: */
    select(STDIN+1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds))
        printf("Une touche à été pressée!\n");
    else
        printf("Timed out.\n");
}
```

Si vous lancez ce programme à partir d'un terminal (qui bufferise la ligne), vous devrez appuyer sur la touche RETURN ou il y aura un time out.

Maintenant, plusieurs d'entre vous peuvent penser que c'est là la bonne méthode pour attendre les données sur une socket datagram -- et vous avez raison *cela peut l'être*. Certain unix implémentent select pour cette utilisation, et d'autres non. Vous devriez regarder les pages de manuel de votre système pour savoir de quelle manière vous allez aborder le problème.

Pour conclure sur l'appel système `select()`: si vous avez une socket qui est en train d'écouter avec `listen()`, vous pouvez regarder si il y a une nouvelle connection en mettant le descripteur de socket dans l'ensemble `readfds`.

Et voilà,, mes amis, un rapide aperçu de la toute puissante fonction `select()`.

Pour plus d'informations

Vous êtes arrivé jusqu'ici, et vous en voulez encore! Ou d'autre pouvez vous trouver de nouvelles informations sur ce sujet?

Regardez les quelques pages de manuel pour commencer:

- [socket\(\)](#)
- [bind\(\)](#)
- [connect\(\)](#)
- [listen\(\)](#)
- [accept\(\)](#)
- [send\(\)](#)
- [recv\(\)](#)
- [sendto\(\)](#)
- [recvfrom\(\)](#)
- [close\(\)](#)
- [shutdown\(\)](#)
- [getpeername\(\)](#)
- [getsockname\(\)](#)
- [gethostbyname\(\)](#)
- [gethostbyaddr\(\)](#)
- [getprotobyname\(\)](#)
- [fcntl\(\)](#)
- [select\(\)](#)
- [perror\(\)](#)

Jetez un coup d'oeil à ce qui suit [livres](#):

Internetworking with TCP/IP, volumes I-III par Douglas E. Comer et David L. Stevens. Publié par Prentice Hall. Seconde édition ISBNs: 0-13-468505-9, 0-13-472242-6, 0-13-474222-2. Il y a une troisième édition de cet ensemble qui couvre IPv6 et IP sur ATM.

Using C on the UNIX System par David A. Curry. Publié par O'Reilly & Associates, Inc. ISBN 0-937175-23-4.

TCP/IP Network Administration par Craig Hunt. Publié par O'Reilly & Associates, Inc. ISBN 0-937175-82-X.

TCP/IP Illustrated, volumes 1-3 par W. Richard Stevens and Gary R. Wright. Publié par Addison Wesley. ISBNs: 0-201-63346-9, 0-201-63354-X, 0-201-63495-3.

Unix Network Programming par W. Richard Stevens. Publié par Prentice Hall. ISBN 0-13-949876-1.

Sur la toile: le web:

[Les Sockets BSD: A Quick And Dirty Primer](http://www.cs.umn.edu/~bentlema/unix/--a%20aussi%20d'autres%20infos%20de%20programmation%20super%20sur%20les%20syst%C3%A8me%20Unix!/)

([http://www.cs.umn.edu/~bentlema/unix/--a aussi d'autres infos de programmation super sur les système Unix!](http://www.cs.umn.edu/~bentlema/unix/--a%20aussi%20d'autres%20infos%20de%20programmation%20super%20sur%20les%20syst%C3%A8me%20Unix!/))

[Informatique Client-Serveur](http://pandonia.canberra.edu.au/ClientServer/socket.html)

(<http://pandonia.canberra.edu.au/ClientServer/socket.html>)

[Intro à TCP/IP](http://gopher://gopher-chem.ucdavis.edu/11/Index/Internet_aw/Intro_the_Internet/intro.to.ip/) (gopher)

(gopher://gopher-chem.ucdavis.edu/11/Index/Internet_aw/Intro_the_Internet/intro.to.ip/)

[Foire Aux Questions \(FAQ\) Protocole Internet](http://web.cnam.fr/Network/TCP-IP/) (France)

(<http://web.cnam.fr/Network/TCP-IP/>)

[La FAQ des Sockets Unix](http://www.ibrado.com/sock-faq/)

(<http://www.ibrado.com/sock-faq/>)

RFCs--Mettez vos mains dans le camboui:

[RFC-768](ftp://nic.ddn.mil/rfc/rfc768.txt) -- Le Protocole UDP (User Datagram Protocol)

(<ftp://nic.ddn.mil/rfc/rfc768.txt>)

[RFC-791](ftp://nic.ddn.mil/rfc/rfc791.txt) -- Le Protocole IP (Internet Protocol)

(<ftp://nic.ddn.mil/rfc/rfc791.txt>)

[RFC-793](ftp://nic.ddn.mil/rfc/rfc793.txt) -- Le Protocole TCP (Transmission Control Protocol)

(<ftp://nic.ddn.mil/rfc/rfc793.txt>)

[RFC-854](ftp://nic.ddn.mil/rfc/rfc854.txt) -- Le Protocole Telnet

(<ftp://nic.ddn.mil/rfc/rfc854.txt>)

[RFC-951](ftp://nic.ddn.mil/rfc/rfc951.txt) -- Le Protocole BOOTP (Bootstrap Protocol)

(<ftp://nic.ddn.mil/rfc/rfc951.txt>)

[RFC-1350](ftp://nic.ddn.mil/rfc/rfc1350.txt) -- Le Protocole TFTP (Trivial File Transfer Protocol)

(<ftp://nic.ddn.mil/rfc/rfc1350.txt>)

Déclaration et appel à l'aide

Bien, Ce document a été relu par d'autres personnes heureusement et j'espère sincèrement qu'aucune aucune erreur éclatante ne s'est glissé dans ce document. Cependant il y a en a toujours :(.

Ainsi, s'il y a, c'est dur pour vous. Je suis désolé si des inexactitudes contenues dans ce document vous ont posé des problèmes, mais vous ne pouvez pas me tenir pour responsable.

Mais peut être que non. Après tout, J'ai passé beaucoup de temps sur ce bazar, et j'ai implémenté plusieurs utilitaires réseaux TCP/IP pour Windows (incluant Telnet). Je ne suis pas un dieu de la programmation des sockets; Je suis juste une personne comme les autres.

Si quelqu'un a une remarque constructive (ou non) à propos de ce document, pouvez vous me l'envoyer à beej@ecst.csuchico.edu et j'essayerais de la prendre en compte et de mettre à jour ce document.

Au cas où vous vous demanderiez pourquoi j'ai fait ceci, bien, je l'ai fait pour l'argent Non, vraiment, je l'ai fait parce qu'un bon nombre de gens m'ont posé des questions à ce sujet et en outre, j'estime que toute cette connaissance durement gagnée va se gaspiller si je ne peux pas la partager avec d'autres. Le WWW s'avère justement être le véhicule parfait. J'encourage d'autres à fournir les informations semblables autant que possible.

Assez avec tout ça--retournons au code! ;-)

Copyright © 1995, 1996 par Brian "Beej" Hall. Ce guide peut être réimprimé par n'importe quel moyen à condition que son contenu ne soit pas changé, il est présenté en sa totalité, et cette notification de tous droits réservés reste intact. Contactez beej@ecst.csuchico.edu pour plus d'information.