

Programmation Réseau en C sous Unix

Chapitre 1 : Notions de base

I. Structure d'adresse :

En programmation réseau, nous allons utiliser les sockets. C'est un moyen de communication qui se définit par un port et une adresse.

Elle est représentée sous la forme d'un descripteur de fichier (un entier).

Les fonctions permettant de manipuler des sockets prennent en argument une structure d'adresse de socket.

a) La structure d'adresse de socket IPV4 :

Aussi appelé "structure d'adresse de socket internet", elle est définie dans `<netinet/in.h>`.

```
struct sockaddr_in
{
    uint8_t sin_len; //taille de la structure
    sa_family_t sin_family; //famille de la socket
    in_port_t sin_port; //numéro de port TCP ou UDP
    struct in_addr sin_addr; //adresse IPv4
    char sin_zero[8]; //inutilisé
};
```

Avec

```
struct in_addr
{
    in_addr_t s_addr; //adresse IPv4 sur 32 bits
};
```

sin_len : utilisé uniquement avec les sockets de routage.

sin_family : vaut `AF_INET` pour un socket IPv4.

sin_port : numéro de port sur 16 bits.

sin_addr : adresse IPv4 sur 32 bits.

b) Structure d'adresse de socket générique :

Les fonctions de gestion de socket doivent fonctionner avec toutes les familles de protocoles supportées, donc avec différentes structures de socket.

Programmation Réseau en C sous Unix

- Problème : comment faire pour qu'une fonction accepte des structures différentes comme paramètres ?
- on peut utiliser un pointeur `void*` en ANSI C. Ce qui ne fonctionne pas avec les sockets.
 - On utilise une structure générique pouvant contenir toutes les autres

```
struct sockaddr
{
    uint8_t sa_len;
    sa_family_t sa_family;
    char sa_data[14];
}
```

Tout appel à une fonction de socket se fera avec un transtypage de la structure d'adresse de socket générique.

Grâce au champ `sa_family`, le compilateur saura où se trouvent les différents champs de la structure.

Exemple :

```
struct sockaddr_in servaddr;
int sockfd;
: //remplissage servaddr
: //initialization sockfd
connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
```

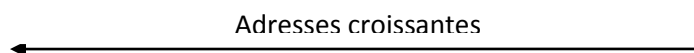
NB :

Avant d'utiliser une structure d'adresse de socket, il est préférable de l'initialiser à 0. Cela se fait avec un des 2 fonctions suivantes :

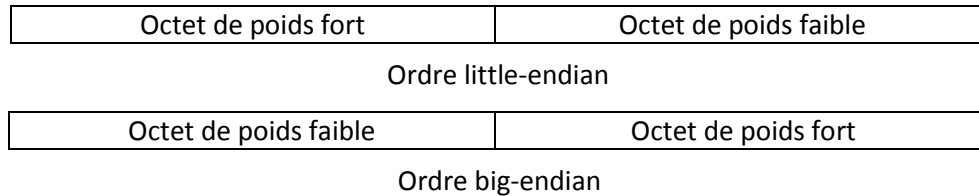
```
#include <strings.h>
void bzero(void* dest, size_t nbytes);
#include <string.h>
void *memset(void* dest, int c, size_t len);
```

II. Ordre d'octets :

Soit un entier codé sur 16 bits (2 octets). Il y a 2 façons de le stocker en mémoire :



Programmation Réseau en C sous Unix



Il n'y a pas de norme sur la façon dont sont stockées les valeurs multi-octets en mémoire. Ce qui peut poser problème :

Exemple :

259 en little-endian :
poids fort | poids faible
0000 0001 | 0000 0011
poids faible | poids fort
Lu sur un système en big-endian :
→ 769

Cette différence de stockage est importante en programmation réseau car les protocoles réseaux spécifient tous un ordre réseau d'octets. Par exemple, les protocoles internet utilisent les big-endian.

Les champs port et adresse des structures de socket doivent être en ordre réseau.

Si l'ordre de stockage de la machine n'est pas le même que l'ordre réseau, il faut convertir les valeurs.

```
#include <netinet/in.h>
uint16_t htons(uint16_t val);
uint32_t htonl(uint32_t val);
    retournent val en ordre réseau sur 16 et 32 bits
uint16_t ntohs(uint16_t val);
uint32_t ntohl(uint32_t val);
    retournent val en ordre machine sur 16 et 32 bits

h : hôte
n : network
s : short
l : long
```

III. Conversion d'adresses :

Les adresses IP sont toujours utilisées sous la forme de 32 bits par les fonctions en programmation réseau. Or, cette forme n'est pas utilisable par un humain. On utilise généralement la forme décimale pointée X.Y.Z.T.

On a donc besoin de fonctions de conversions entre ces 2 formes.

Programmation Réseau en C sous Unix

a) Fonctions IPv4 seulement :

```
#include <arpa/inet.h>
int inet_aton(const char *str, struct in_addr *addr);
    convertit l'adresse décimale pointée str en adresse sur 32 bits
    addr (en ordre réseau)

char *inet_ntoa(struct in_addr addr)
    convertit l'adresse sur 32 bits addr et retourne la forme
    décimale pointée.
```

b) Fonctions IPv4 et IPv6 :

```
#include <arpa/inet.h>
int inet_pton(int family, char *str, void *addr);
    Transforme l'adresse décimale pointée str en adresse sur 32
    bits addr.
    Family = AF_INET pour IPv4
            = AF_INET6 pour IPv6

char *inet_ntop(int family, void *addr, char *str, size_t len);
    Convertit addr sur 32 bits en adresse sous forme décimale
    pointée str.

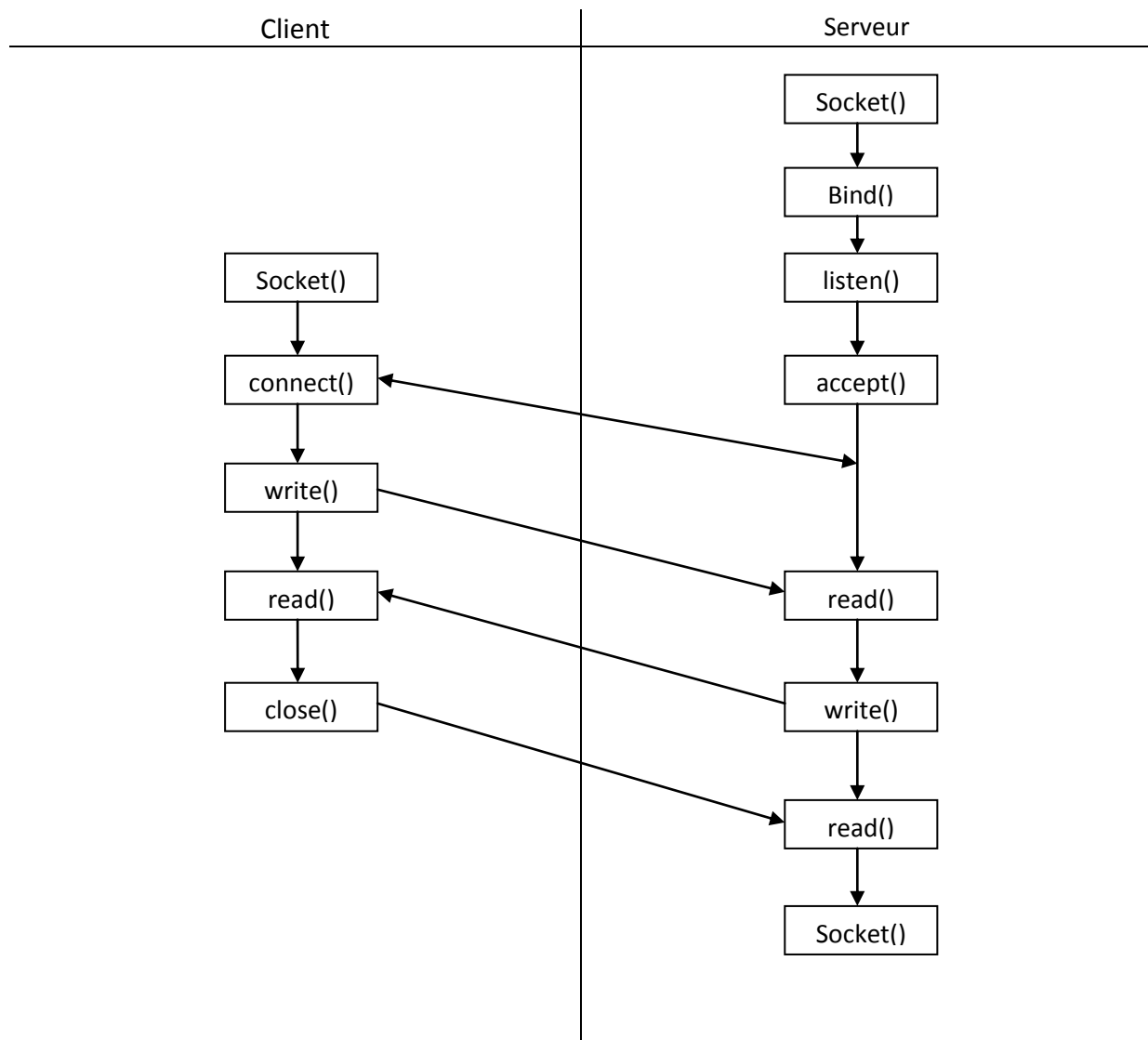
Family : cf inet_pton
Len : taille de la chaîne :
    IPv4 = 16
    IPv6 = 46
```

Programmation Réseau en C sous Unix

Chapitre 2 : les sockets TCP

I. Fonctionnement d'un client et d'un serveur TCP

Nous allons tous d'abord voir les différentes fonctions nécessaires à l'établissement, le maintien et la fin d'une connexion entre un client et un serveur TCP.



II. La fonction socket :

```
#include <sys/socket.h>
int socket (int family, int type, int protocole)
    Retourne un descripteur de socket
```

Crée un socket avec le type de protocole de communication désiré.

Programmation Réseau en C sous Unix

Family : famille de protocoles

AF_INET : protocoles IPv4

AF_INET6 : protocoles IPv6

AF_LOCAL : protocoles du domaine Unix

AF_ROUTE : sockets du router

Type : type de socket

SOCK_STREAM -> sockets TCP

SOCK_DGRAM -> sockets UDP

SOCK_RAW -> raw sockets

Protocol : protocole utilisé par le socket :

|IPPROTO_TCP

|IPPROTO_UDP

ou 0 : le système choisit le protocole selon family et type :

SOCK_STREAM + AF_INET = TCP

SOCK_STREAM + AF_INET6 = TCP

SOCK_DGRAM + AF_INET = UDP

SOCK_DGRAM + AF_INET6 = UDP

III. La fonction bind :

Elle associe à un socket une adresse locale de protocole. Pour les protocoles internet, l'adresse de protocole est composée d'une adresse IP (V6 ou V4) et d'un port.

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```

Sockfd : descripteur du socket à laquelle on veut lier l'adresse et le port

Addr : structure d'adresse contenant l'IP et le port à lier au socket.

Len : taille de cette structure.

Bind permet de spécifier par quelle adresse IP et quel port le socket va communiquer.

Le serveur doit appeler bind pour au moins spécifier le port sur lequel il écoute.

Par contre, le client n'utilise généralement pas bind. Il laisse le noyau choisir un port éphémère et la meilleure adresse IP selon le routage (si le client a plusieurs IP).

On peut laisser le choix au noyau de l'adresse IP à lier au socket en mettant l'adresse INADDR_ANY comme adresse dans la structure.

Lors de la connexion (TCP) ou de la réception de données (UDP), le noyau choisit l'IP.

Programmation Réseau en C sous Unix

IV. La fonction listen :

Elle transforme un socket actif (socket client) en un socket passif, prêt à se mettre en attente de connexion (passage de l'état CLOSED à l'état LISTEN).

On peut aussi spécifier avec listen le nombre maximum de connexion que le noyau peut mettre en file d'attente pour le socket.

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

Sockfd : descripteur de socket

Backlog : nombre maximal de connexions

Backlog = nombre de connexions complètes + nombre de connexions incomplètes (juste un SYN reçu).

V. La fonction accept :

Elle est appelée par le serveur TCP. Elle le met en attente de connexion et ne retourne une valeur qu'une fois la connexion complète.

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *clilen);
```

Retourne | un descripteur de socket connectée si OK
| -1 sinon

Sockfd : descripteur du socket passif

Cliaddr : contiendra la structure d'adresse du client connecté.

Clilen : taille de cette structure.

On utilisera le descripteur retourné par accept pour lire et/ou écrire dans le socket.

Programmation Réseau en C sous Unix

```
1 seule fois
Int listenfd, connfd ;
struct sockaddr_in serraddr, cliaddr ;
listenfd = socket(...);
//remplissage de servaddr
bind(listenfd, (struct sockaddr *)&servaddr, ...);
listen(listenfd,...);

Pour chaque
connexion
For( ; ; )
{
    connfd = accept(listenfd, (struct sockaddr
*)&cliaddr, ...) ;
    //traitement de la connexion;
    .
    .
    .
}close(connfd) ;
```

VI. La fonction connect :

Elle est utilisée par un client TCP pour établir une connexion avec un serveur.

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr* servaddr, socklen_t
addrlen);
```

Retourne | 0 si OK
| -1 sinon

Socketfd : descripteur de socket retourné par la fonction socket.

Servaddr : structure d'adresse du serveur.

Addrlen : taille de cette structure.

Connect ne retourne une valeur qu'une fois le 3way handshake terminé ou s'il y a une erreur. Les erreurs peuvent être :

- Le client ne reçoit pas la réponse à son SYN. Après 75 secondes à renvoyer le SYN régulièrement, l'erreur ETIMEDOUT est générée.
- Le client reçoit un RST en réponse à son SYN (il n'y a pas de processus serveur qui attend une connexion sur le port demandé. L'erreur ECONNREFUSED est générée.

Programmation Réseau en C sous Unix

- La réponse au SYN est un paquet ICMP "destination unreachable" envoyé par un routeur. Comme dans le premier cas le SYN va être renvoyé plusieurs fois et après 75s, l'erreur EHOSTUNREACH est générée.

VII. La fonction close :

C'est la fonction close des E/S de bas niveau :

```
#include <unistd.h>
int close(int sockfd);
```

Elle provoque la fermeture du socket. Il ne sera plus possible d'y lire ou d'y écrire, mais les données en attente dans le socket seront quand même transmises.

Close provoque la décrémentation du compteur de références du socket. Ce compteur représente les processus utilisant le descripteur du socket. C'est seulement quand ce compteur passe à 0 que la phase de déconnection TCP commence.

VIII. Serveur concurrent :

Le serveur que nous venons de voir est un serveur itératif. Il accepte une connexion, le traite et se remet en attente.

Si le traitement de la connexion est long aucun autre client ne peut se connecter. Pour gérer plusieurs clients à la fois, on va séparer la partie connexion de la partie traitement. Pour cela, une fois la connexion établie, le serveur crée un processus serveur fils qui traitera la connexion pendant qu'il se remettra en attente de client.

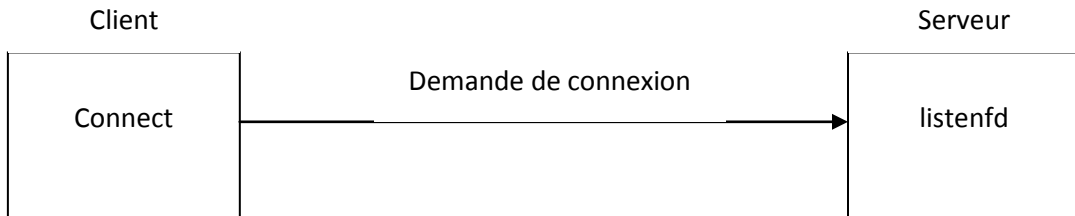
Structure d'un serveur concurrent ;

```
//phase préliminaire
for(;;)
{
    connfd = accept(listenfd, ...);
    pid = fork();
    if(pid == 0)
    {
        close(listenfd); //fenêtre de la socket passive
        traitement(connfd); //traitement de la connexion
        close(fd); //fin de la connexion
        exit(0);
    }
    close(fd); //fenêtre de la socket connectée, mais pas de la connexion
```

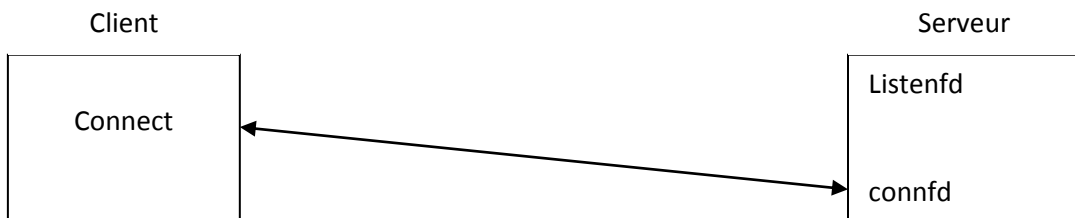
Programmation Réseau en C sous Unix

Lors du fork, les descripteurs ouverts sont hérité par le fils.

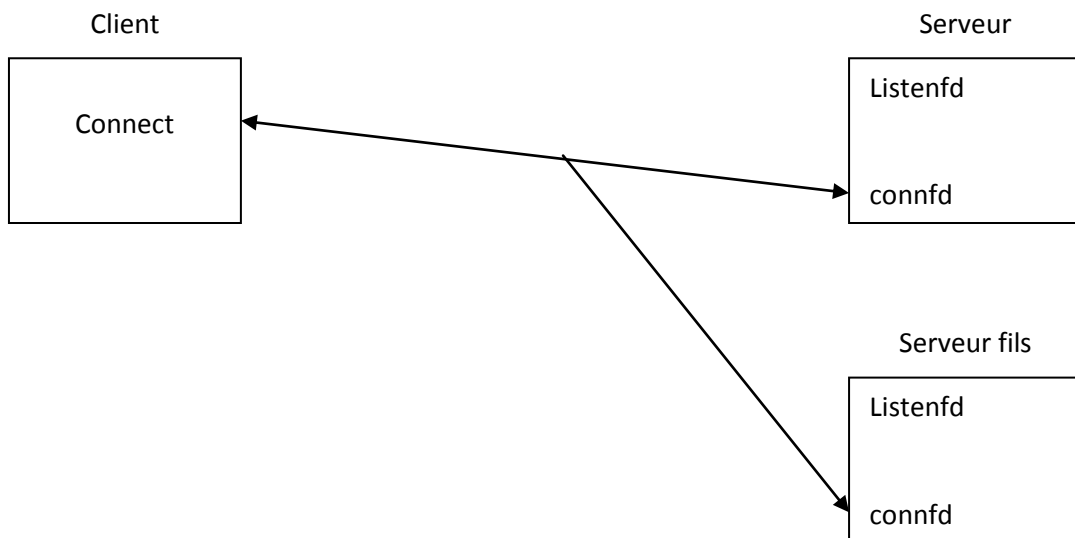
Avant la fin de accept :



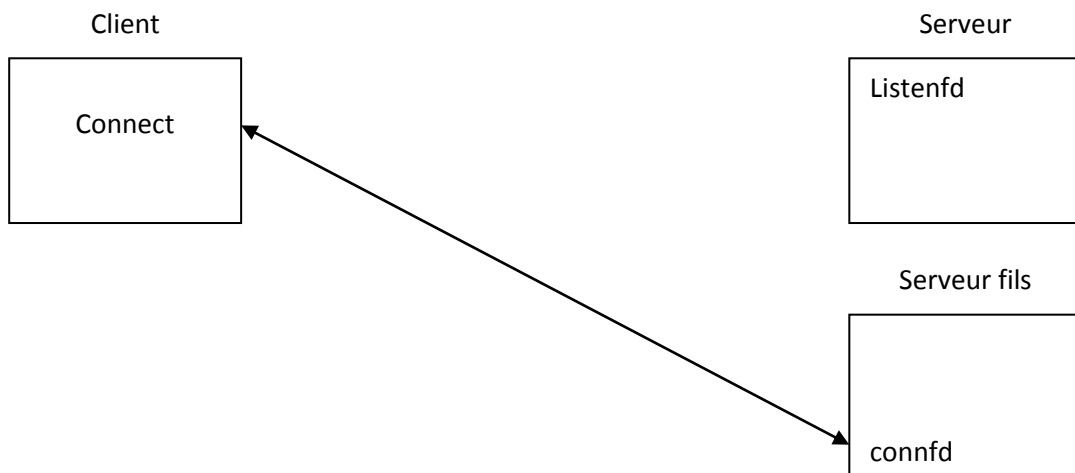
Après accept :



Après fork



Après close(listenfd) du fils et close(connfd) du père.



Programmation Réseau en C sous Unix

IX. Gestion des serveurs fils :

Quand une connexion se termine, le processus fils qui en était chargé se termine. Or, son processus père (le serveur) ne s'arrête jamais et n'a aucune procédure de récupération des codes retour de ses fils. Ceux-ci deviennent donc des zombies.

Pour des raisons d'optimisations on doit éliminer ces zombies du système. Pour cela, on va appeler une fonction de type `wait` quand un fils se termine. On va donc détourner le signal `SIGCHLD` reçu par le père lui indiquant la fin d'un de ses fils.

```
Signal(SIGCHLD, fin-fils) ;
```

Avec :

```
Void fin-fils(int signo)
{
    wait(NULL) ;
}
```

Ceci pose un problème dû au non stockage des signaux. En effet, si 2 fils se terminent en même temps, le premier `SIGCHLD` sera traité et le deuxième perdu.

Pour éviter de ne pas traiter la fin de certains fils, on va appeler une fonction de type `wait` tant qu'il y a des fils morts. Pour cela, on utilisera `waitpid`, qui peut être rendu non-bloquante et qui retourne 0 s'il n'y a pas de fils terminée à attendre.

```
Void fin-fils(int signo)
{
    While(waitpid(-1, NULL, WNOHANG)>0)
}
```

Attente de
tous les
processus

statut

option
qui rend
non-bloquant

Avec la gestion du signal `SIGCHLD`, donc l'interruption éventuelle de l'appel système `accept`, il faut implémenter le mécanisme de relance des appels systèmes bloquants interrompus (cf. cours système).

```
For( ; ; )
{
    If ((connfd = accept(...)) < 0)
    {
        If(errno = EINTR)           Si l'appel système accept a été
        Continue ;                  interrompu on revient au for
    }
}
```

Programmation Réseau en C sous Unix

```
        Else{
            Printf("l'erreur accept\n") ;
            Exit(-1) ;
        }
    }
}
```

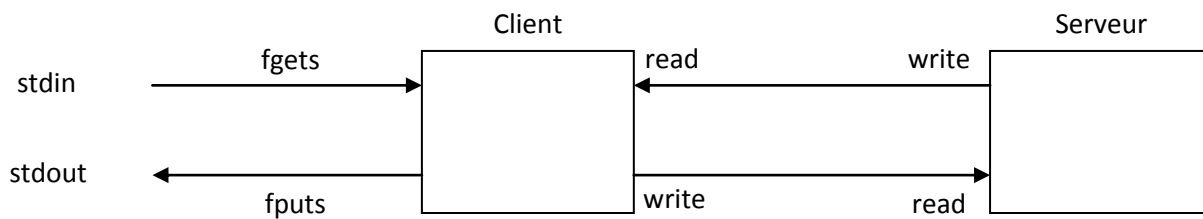
Programmation Réseau en C sous Unix

Chapitre 3 : Etude d'un Client/serveur

I. Introduction

Nous allons écrire un client echo et un serveur echo qui auront le fonctionnement suivant :

- Le client lit une chaîne sur l'entrée standard et l'envoie au serveur.
- Le serveur affiche la ligne reçue et la renvoie au client.
- Le client affiche ce qu'il a reçu sur la sortie standard.



Le serveur est un serveur concurrent.

II. Le client :

Une fois la connexion établie, le client appellera la fonction traitement client :

```
Void traitement_client (FILE *fp, int sockfd)
{
    Char sendline[MAXLINE], recvline[MAXLINE];
    While(fgets(sendline, MAXLINE, fp) != NULL)
    {
        Write(sockfd, sendline, strlen(sendline));
        If(read(sockfd, recvline, MAXLINE) == 0)
        {
            Printf("serveur terminé\n");
            Exit(-1);
        }
        Fputs(recvline, stdout);
    }
}
```

Entrée des données (généralement stdin)

socket connectée

Programmation Réseau en C sous Unix

III. Lancement du Client/serveur

On va démarrer notre serveur, puis notre client et observer ce qui se passe au niveau des sockets avec l'outil netstat et l'option -at.

1. On lance le serveur sur le port 1234 avec l'IP wildcard.
netstat nous donnera :

Proto	Recv_Q	Send_Q	Local Address	Foreign Address	State
tcp	0	0	*:1234	*:*	LISTEN

2. On lance le client qui se connecte au serveur sur le même hôte, on a :

Serveur-père	Tcp	0	0	*:1234	*:*	LISTEN
Serveur-fils	Tcp	0	0	localhost:1234	localhost:9342	ESTABLISHED
client	tcp	0	0	localhost:9342	localhost:1234	ESTABLISHED

On peut aussi afficher les raisons du blocage d'un processus avec le champ wchan de ps. Dans notre cas, on aura :

Pour le serveur père : `wait_for_connect`

Pour le serveur fils : `tcp_data_wait`

Pour le client : `read_chan`

IV. Fin normal :

Le client envoie des données saisies, le serveur les lui retourne. L'utilisateur tape CTRL+D dans le client :

1. `fgets` retourne NULL, la fonction `traitement_client` se termine.
2. Le client tout entier se termine provoquant la fermeture de tous ses descripteurs ouverts. Ce qui provoque l'envoi d'un FIN au serveur qui répond avec un ACK. (serveur en `CLOSE_WAIT` et client en `FIN_WAIT_2`)
3. Le `read` du fils retourne 0 donc le fils se termine et ferme ses descripteurs. Il envoie un FIN au client qui retourne un ACK (fin de la phase de déconnexion). Le client entre dans l'état `TIME_WAIT`.

Etat `TIME_WAIT`

C'est l'état dans lequel se trouve le pair qui a initié la fin de la connexion. La partie de la connexion qui se trouve en `TIME_WAIT` y reste un temps égal à $2 \times \text{MSL}$ (Maximum Segment Lifetime), c'est-à-dire deux fois la durée de vie maximum d'un datagramme IP sur un réseau.

Cet état empêche l'hôte qui s'y trouve de reconstruire une connexion avec la même adresse IP et le même port. Ainsi, grâce aux $2 \times \text{MSL}$, on est sûr qu'aucun paquet perdu de la connexion précédente ne sera présent sur le réseau et donc susceptible de revenir. L'attente en `TIME_WAIT` évite la présence de paquets dupliqués.

Programmation Réseau en C sous Unix

Cette persistance dans l'état TIME_WAIT permet aussi de s'assurer que la phase de connexion TCP s'est déroulée correctement. En effet, si un des 2 paquets de fin de connexion est perdu, il doit être possible de le renvoyer, notamment en ce qui concerne l'ACK final envoyé par le pair responsable de la fermeture active.

V. Scénario 1 : Fin du processus serveur

1. On lance le serveur et le client. On teste la connexion en envoyant des données du client.
2. On tue le processus du serveur. Les descripteurs du serveur sont fermés, un FIN est envoyé au client, qui répond avec un ACK.
3. Il ne se passe rien chez le client qui est bloqué dans le fgets.

Netstat nous informe que :

- Le client est en CLOSE_WAIT
 - Le serveur fils est en FIN_WAIT2 (à la moitié de la phase de connexion).
4. On tape une ligne au clavier dans le client, ce qui le débloque. Il envoie les données au serveur (le FIN du serveur indique juste qu'il n'enverra plus de données), qui lui retourne un RST.
 5. Le client, dans le read, ne voit pas le RST. Read retourne 0 et le client constate que le serveur est terminé.

VI. Scénario 2 : la machine serveur plante

C'est un arrêt brutal de la machine après établissement de la connexion. Dans ce cas, rien n'est envoyé au client pour le prévenir de l'arrêt du serveur.

Quand le client envoie les données il ne reçoit pas d'ACK et va donc retransmettre le paquet 12 fois (environ égal à 9 minutes). Finalement read retourne un erreur avec errno = ETIMEDOUT (ou EHOSTUNREACH si c'est un routeur intermédiaire qui a déclaré l'hôte inatteignable).

VII. Scénario 3 : la machine serveur plante et redémarre

On établit la connexion, le serveur plante, redémarre et le client envoie des données. Le serveur, n'ayant plus trace de la connexion précédente, retourne un paquet RST au client.

Read retournera une erreur avec errno = ECONNRESET.

Programmation Réseau en C sous Unix

Chapitre 4 : Multiplexage d'Entrée/Sorties

I. Introduction

Nous avons vu dans le chapitre précédent que notre client avait à gérer 2 entrées à la fois : l'entrée standard et le socket. Cela posait problème notamment lorsque le client bloqué dans le fgets ne voyait pas que le processus serveur était terminé.

Il faudrait que nous puissions être prévenus par le noyau si un évènement (données à lire, erreur, possibilité d'écrire, ...) se produisait sur un ou plusieurs descripteurs à la fois. C'est le multiplexage d'E/S.

II. La fonction select

Cette fonction permet à un processus d'informer le noyau de surveiller des descripteurs en attente d'événements, et de réveiller le processus si un ou des événements se sont produits.

Les événements détectables sont :

- Le descripteur est prêt en lecture
- Le descripteur est prêt en écriture
- Un certain temps s'est écoulé.

```
#include <sys/select.h>
#include <sys/time.h>

int select(int masefdp1, fd_set readset, fd_set writeset, fd_set
exceptset, struct timeval timeout);
```

Retourne :

- Le nombre de descripteur prêts
- 0 si le compteur de temps est terminé
- -1 si erreur

Masefdp1 : indique le nombre de descripteurs à surveiller. Sa valeur est égale à la valeur du plus grand descripteur à surveiller (masefd) à laquelle on ajout 1 (p1).

Readset : ensemble des descripteurs à surveiller en lecture.

Writeset : ensemble des descripteurs à surveiller en écriture.

Exceptset : ensemble des descripteurs à surveiller en exception.

Timeout : permet de gérer le temps d'attente de select.

Programmation Réseau en C sous Unix

Les ensembles de descripteurs

Un ensemble de descripteurs est un tableau d'entiers dans lequel chaque bit de chaque entier correspond à un descripteur sur 32 bits, la case 0 contient les descripteurs de 0 à 31, la case 1 de 32 à 63, etc... La présence d'un descripteur dans l'ensemble est signalée par la valeur 1 du bit correspondant. C'est le type `fd_set` qui permet de les gérer de façon transparente grâce à des macros :

- `Void FD_ZERO(fd_set *fds)`
vide l'ensemble `fds`
- `Void FD_SET(int fd, fd_set *fds)`
met le descripteur `fd` dans l'ensemble `fds`
- `Void FD_CLR(int fd, fd_set *fds)`
retire le descripteur `fd` de l'ensemble `fds`
- `Int FD_ISSET(int fd, fd_set fds)`
teste la présence du descripteur `fd` dans l'ensemble `fds`

Ex :

```
Fd_set rset;
FD_ZERO(&rset); //initialization à 0 de rset
FD_SET(1, &rset); //Les descripteur 1, 6 et 7 sont mis dans rset
FD_SET(6, &rset);
FD_SET(7, &rset);
```

Le compteur de temps timeout :

Cet argument est de type

```
Struct timeval
{
    Long tv_sec; //nombre de secondes
    Long tv_usec; //nombre de microsecondes
}
```

Il y a 3 possibilités d'attente :

- Attente infinie : `select` ne sera débloquenté que lorsqu'un des descripteurs surveillés sera prêt. Timeout doit être à `NULL`.
- Attente finie : si aucun des descripteurs surveillés n'est prêt avant le temps spécifié, alors `select` est débloquenté et retourne 0. Pour cela, `timeout.tv_sec != 0` ou `timeout.tv_usec != 0`.
- Attente nulle : `select` n'est pas bloquant et retourne 0 si aucun descripteur n'est prêt. Dans ce cas, `timeout.tv_sec = timeout.tv_usec = 0`.

Programmation Réseau en C sous Unix

La notion de descripteur prêt :

Un descripteur sera déclaré prêt en lecture si :

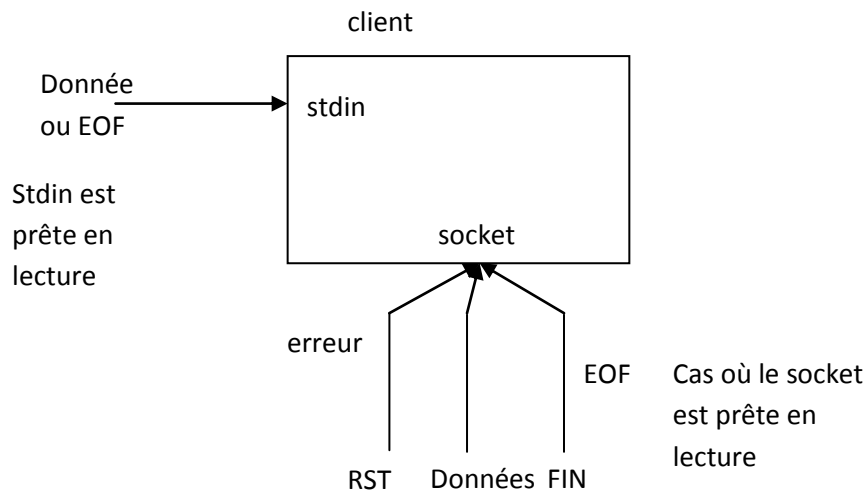
- Le nombre d'octets de données dans le buffer de réception du socket est supérieur ou égal à la taille du low_water mark du buffer de réception (= taille minimale des données reçues avant qu'on puisse les lire, par défaut 1).
- La moitié en lecture de la connexion est fermée (on a reçu un FIN) read sera alors non-bloquant et retournera 0.
- Il y a une erreur pendante sur le socket. Un read va être non-bloquant et retournera -1.
- Le socket est passif et une connexion complète existe. Un accept sur le socket passif sera donc non-bloquant.

Un descripteur sera déclaré prêt en écriture si :

- La taille de l'espace libre dans le buffer d'émission de la socket est supérieur ou égal à la taille du low_water mark du socket d'émission (2048 par défaut).
- Une erreur de socket est pendante, un write sera non-bloquant et retournera -1.

III. Amélioration du client echo

Nous allons maintenant réécrire notre client echo avec select pour gérer les scénarios qui posent problème. On peut schématiser ainsi les conditions à surveiller par select.



Algorithme du client :

rset : ensemble de descripteurs

initialisation de rset (`FD_ZERO`)

Boucle client :

- Mettre le descripteur d'entrée de données dans rset (`FD_SET`)
- Mettre le descripteur de socket dans rset (`FD_SET`)
- Calculer le paramètre maxfdp1 de select (`maxfdp1 = max(descripteur entrée, descripteur socket) + 1`)

Programmation Réseau en C sous Unix

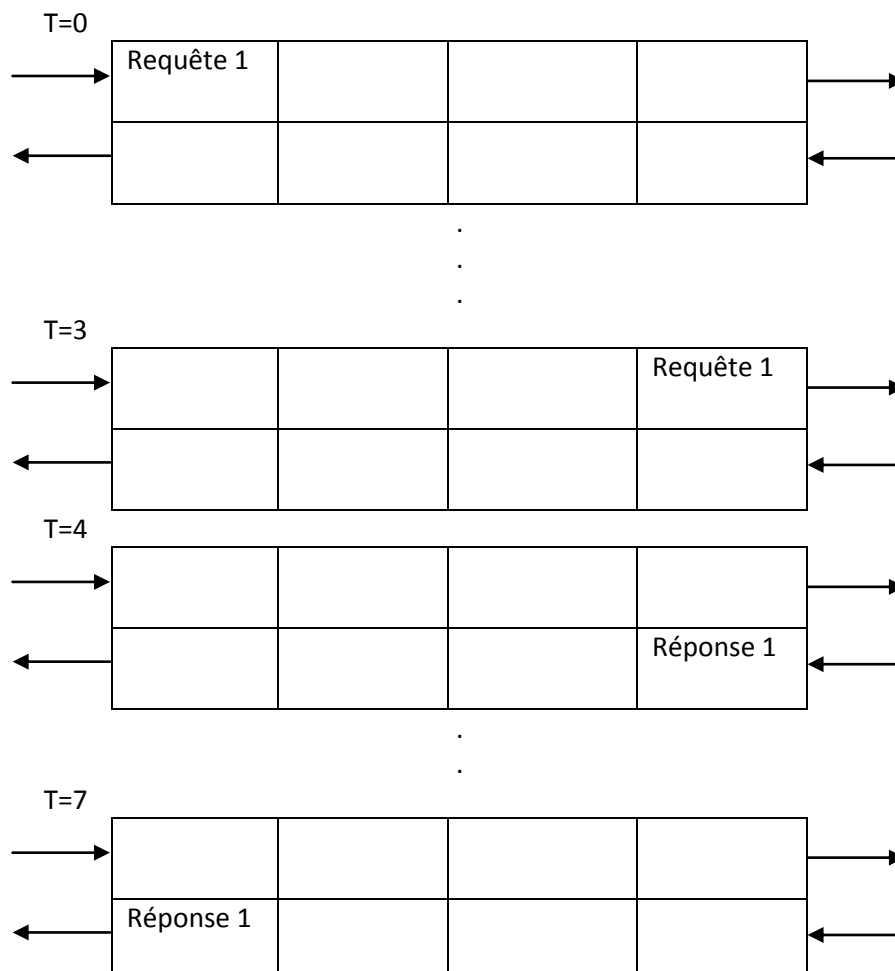
- `select` sur `rset` en lecture, avec temps d'attente infini.
- Si le descripteur de socket est prêt en lecture (`FD_ISSET`) (\Leftrightarrow descripteur de socket appartient à `rset`)
alors lecture de la socket (`read`)
si tout s'est bien passé
alors affichage de ce qui a été lu à l'écran
sinon sortie du programme
- Si le descripteur d'entrée est prêt en lecture (\Leftrightarrow descripteur d'entrée appartient à `rset`) (`FD_ISSET`)
alors lecture sur ce descripteur (`fgets`)
s'il y a quelque chose à lire ($\Leftrightarrow \neq \text{NULL}$)
alors écriture dans le socket de ce qui a été lu (`write`)
sinon fin du programme

IV. Envoie par lots

Jusqu'à présent, nous avons considéré que notre programme prenait ses données d'entrée du clavier, en mode interactif. Ce type de fonctionnement dit "stop-and-wait" ne pose pas de problème de vidage de la socket car les envois sont espacés, et la socket n'est jamais totalement "remplie".

Exple :

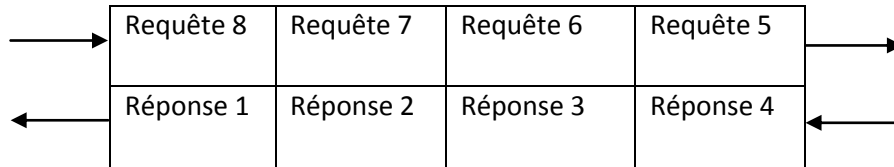
Pour un RTT (Round Trip Time) de 8 unités de temps, on a :



Programmation Réseau en C sous Unix

Le tube n'est utilisé qu'à 1/8 de sa capacité.

Supposons que l'entrée de données soit un fichier. Les données vont se succéder sans interruption jusqu'à la fin du fichier. Le tube sera entièrement rempli :



Si la requête 8 est la dernière ligne du fichier, `fgets` va retourner EOF, et le client se termine. Cela va provoquer la fermeture du socket par le client.

Or, il reste des choses à lire dans le socket. La fin du fichier ne signifie pas que le travail du client est terminé, cela signifie juste que le client n'écrira plus dans le socket.

Le client doit donc fermer sa socket en écriture et ainsi signaler au serveur qu'il ne lui enverra plus rien, mais qu'il peut encore lire.

Pour cela, quand le client aura lu EOF sur l'entrée de données, il appellera la fonction `shutdown`.

```
#include <sys/socket.h>
int shutdown(int sockfd, int howto) ;
```

`howto` : indique l'action de `shutdown`

`SHUT_RD` : fermeture du socket en lecture. Les données dans le buffer de réception sont supprimés et plus aucune opération de lecture n'est possible.

`SHUT_WR` : fermeture du socket en écriture (half-close). Les données dans le socket sont envoyées, suivis du début de la séquence de déconnexion TCP (FIN).

`SHUT_RDWR` : combinaison des 2 précédents.

Modification du client :

Dans la version précédente, plus rien à lire en entrée signifiait fin du programme. Or, ici nous allons continuer à lire dans le socket, mais sans surveiller l'entrée de données dont on sait qu'elle est terminée. On va utiliser un flag, `inotify`, pour marquer la détection d'un EOF sur l'entrée.

`inotify = 0` s'il n'y a pas eu EOF sur l'entrée

`inotify = 1` s'il y a eu EOF sur l'entrée.

Programmation Réseau en C sous Unix

Algorithme :

```
a. initeof = 0
b. Initialisation de rset
c. Boucle client
    ▪ Insertion du descripteur de socket dans rset
    ▪ Si (initedof == 0)
        alors insertion du descripteur d'entrée dans rset
d. Calcul de maxfdp1
e. Select sur rset en lecture avec temps infini
f. Si le descripteur d'entrée est prêt en lecture
    alors si la lecture retourne un EOF
        alors initedof = 1 et demi fermeture en écriture de
        la socket
        sinon écrire les données lues dans le socket
g. Si e descripteur de socket es prêt en lecture
    alorssi la lecture retourne 0
        alors si initedof == 1
            alors fin normal
            sinon erreur
        sinon écrire à l'écran ce qui a été lu.
```

V. Serveur echo et select

Nous allons réécrire notre serveur Echo sous la forme d'un seul processus capable de gérer plusieurs clients à la fois grâce à select.

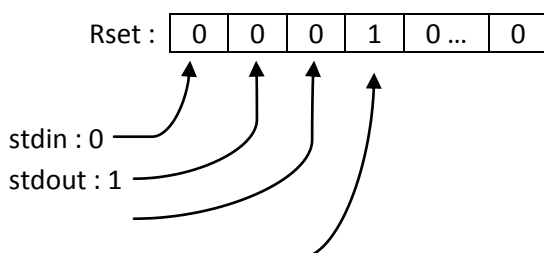
En fait, select surveillera le descripteur de socket passive ainsi que tous les descripteurs de sockets connectées. Ainsi, si une nouvelle connexion arrive, ou qu'un client connecté envoie des données (ou se deconnecte) le noyau avertira le serveur qu'il a gérer un évènement.

Pour arriver à ce résultat, il nous faudra garder une trace de tous les descripteurs de sockets connectées. En effet, une fois select débloqué, et après avoir vérifié si la socket passive en est responsable (arrivée d'une nouvelle connexion), on doit parcourir la liste des sockets connectées afin de voir la ou lesquelles sont responsables du déblocage. On utilisera un tableau d'entiers, clients, où l'on rangera les descripteurs des sockets connectées.

Exemple :

Soit rset l'ensemble des descripteurs à surveiller. Il est vide au lancement du programme (tout à 0), ainsi que clients (tout à -1).

Quand le serveur crée le socket passif, les descripteurs 0, 1 et 2 étant réservés (stdin, stdout et stderr), elle aura le descripteur n°3 :



Programmation Réseau en C sous Unix

stderr : 2

socket passive : 3

Client :

-1	-1	...	-1
----	----	-----	----

Un client se connecte. Le socket connecté se voit attribuer le descripteur n°4. On aura donc :

Rset :

0	0	0	1	1	0	...	0
---	---	---	---	---	---	-----	---

Clients :

4	-1	-1	...	-1
---	----	----	-----	----

Un autre client se connecte :

Rset :

0	0	0	1	1	1	0	...	0
---	---	---	---	---	---	---	-----	---

Clients :

4	5	-1	...	-1
---	---	----	-----	----

Deconnection du premier client :

Rset :

0	0	0	1	0	1	0	...	0
---	---	---	---	---	---	---	-----	---

Clients :

-1	5	-1	...	-1
----	---	----	-----	----

Algorithme :

```
Rset /*descripteur après select*/, allset /*tous les descripteurs à
surveiller*/ : ensembles de descripteurs
nready, maxi /*indice max de clients*/, maxfd /*plus grand
descripteur à surveiller*/ : entiers
.
.
.
Maxfd = listenfd;
Maxi = -1;
Initialisation des cases de clients à -1
Mise à zéro de rset et allset
Listenfd est mis dans allset
Boucle du server :
    Rset <- allset;
    Nready <- select sur rset;
Si (listenfd appartient à rset) = connexion
Alors accept(...)
    Parcours de clients pour trouver la première case libre i. On
    y range le descripteur connfd retourné par accept.
    //sauvegarde du descripteur de socket connectée dans clients.
    Ajout de ce descripteur dans allset.
    //ajout dans l'ensemble des descripteurs à surveiller.
    Si (connfd > maxfd)
        Alors maxfd <- connfd;
    Si (i > maxi)
        Alors maxi <- i;
```

Programmation Réseau en C sous Unix

```
//M.A.J. des valeurs maximum
Si (nready - 1 <= 0) //s'il n'y a aucun autre descripteur
prêt, on revient à la boucle.
    Alors continue;
Pour i de 0 à maxi
    Si (client[i] < 0)
        Alors continue;
    Sockfd <- client[i];
    Si (sockfd appartient rset)
    Alors n <- lecture donc socket;
        Si (n == 0) = fin connexion
        Alors fermeture de la socket;
            Sockfd est retiré de allset
            Client[i] <- -1;
        Sinon on écrit ce qu'on a lu dans la socket.
        Si (nready - 1 <= 0) //il n'y a plus de descripteurs
                                //prêts
            Alors break;
Fin boucle serveur
```

Ce serveur, bien que plus compliqué à écrire, évite de créer autant de processus que de clients. Mais il est vulnérable à une attaque classique, le Denial-of-Service.

Si un client se connecte et envoie un octet de données (par un retour chariot), le serveur va le lire et se bloquer dans read, en attente d'autres données. Si le client ne les lui envoie pas, il est bloqué et ne pourra répondre aux autres.

Une règle de sécurité veut qu'un serveur ne doit jamais être dans un fonction relative à un client. Une façon de rendre ce serveur imperméable à cette attaque, c'est de mettre une limite de temps sur les fonctions bloquantes.

Programmation Réseau en C sous Unix

Chapitre 5 : Option des socket

I. Getsockopt et setsockopt

Il est possible de changer le comportement par défaut des sockets. La fonction `getsockopt` permet de récupérer la valeur d'une option, `setsockopt` permet de modifier cette valeur.

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

int getsockopt( int sockfd, int level, int optname, void *optval,
socklen_t *optlen);
int setsockopt( int sockfd, int level, int optname, const void
*optval, const socklen_t *optlen);
```

`sockfd` : descripteur de la socket dont on veut récupérer/modifier une option.

`level` : niveau de la socket où se situe l'option.

`optname` : nom de l'option.

`optval` : nouvelle valeur attribuée à l'option ou pointeur où sera stockée la valeur de l'option.

`optlen` : taille de l'argument `optval`.

(`optval` est un `void *` car les options peuvent prendre différents types de valeurs. C'est aussi pour cela qu'on spécifiera sa taille avec `optlen`).

II. Etat des sockets

Certaines options nécessitant d'être activées/désactivées à un moment précis selon l'état du socket. Les options suivantes `SO_DEBUG`, `SO_LINGER`, `SO_KEEPAIVE`, `SO_RCVBUF`, `SO_RCVLOWAT`, `SO_SNDBUF`, `SO_SNDLOWAT`, `SO_DONTROUTE` sont hérités par la socket connectée TCP de la socket passive. Pour être sûr qu'une de ces options soit dans l'état voulu dès la phase de connexion terminée, il faut lui donner sa valeur dès la socket passive.

III.Options génériques

Elles sont indépendantes du protocole de la socket, même si elles ne s'appliquent pas forcément à toutes les sockets.

1. `SO_BROADCAST`

Elle permet ou interdit au processus d'envoyer des paquets en diffusion. Cette option ne peut s'appliquer qu'aux sockets non connectés (UDP).

2. `SO_DEBUG`

Quand elle est activée (seulement en TCP), le noyau garde une trace de tous les paquets reçus ou envoyés par la socket. On peut examiner ces traces avec `trtp`.

Programmation Réseau en C sous Unix

3. SO_DONTROUTE

Elle force les paquets à outrepasser les mécanismes normaux de routage. Ils sont envoyés à l'interface locale déterminée avec la partie réseau et la partie sous-réseau de l'adresse de destination

4. SO_KEEPALIVE

Si elle est activée pour un socket TCP, et qu'aucune donnée n'a été échangée sur le socket pendant 2 heures, alors TCP envoie automatiquement une sonde keep_alive au pair. Celui-ci doit y répondre. On a 3 scénarios possibles :

- a. Le pair répond avec le ACK attendu. Tout va bien.
- b. Le pair répond avec un RST pour informer qu'il est tombé et a redémarré. Le socket est fermé et l'erreur ECONNRESET est retournée.
- c. Il n'y a pas de réponse ; TCP va envoyer jusqu'à 8 sondes, espacées de 75 secondes. S'il n'y a toujours aucune réponse après ces 11min 15s, il abandonne, le socket est fermé, l'erreur sera ETIMEDOUT.

5. SO_LINGER

Elle permet de changer le comportement par défaut de la fonction close sur le socket. Par défaut, close retourne immédiatement et s'il y a des données dans le buffer d'envoi du socket, le système va tenter de les faire parvenir au pair.

Elle fonctionne avec la structure suivante :

```
struct linger
{
    int l_onoff ;
    int l_linger ;
}
```

`l_onoff` : 0 : désactive l'option
 1 : active l'option

`l_linger` : délai en secondes

- I. `l_onoff = 0` : le comportement par défaut de classe est conservé.
- II. `l_onoff = 1` et `l_linger = 0` : la connexion est avortée dès le `close` : les données restantes sont supprimées et un RST est envoyé au pair. Cela évite l'état `TIME_WAIT`, avec les risques inhérents.
- III. `l_onoff = 1` et `l_linger > 0` : Lors de l'appel de `close`, le noyau va attendre. C'est-à-dire que s'il y a des données restantes dans le buffer d'envoi alors le processus est endormi jusqu'à ce qu'il ait reçu un ACK pour les données envoyées ou que le temps `l_linger` ait expiré. Ainsi, on peut être sûr que les dernières données envoyées ont bien été reçues (mais pas si elles ont été lues).

Programmation Réseau en C sous Unix

6. `SO_RCVBUF` et `SO_SNDBUF`

Elles permettent de changer la taille du buffer de reception et du buffer d'envoi. Ces options doivent être mise à jour avant la connexion car elles influent sur la taille de la fenêtre échangée lors des SYN de connexion (avant connect pour un client, avant listen pour un serveur). Leur taille doit être au minimum de $4 * MSS$ (maximum segment size) et elle doit être un multiple de MSS pour ne pas gaspiller d'espace.

7. `SO_RCVLOWAT` et `SO_SNDLOWAT`

Elles permettent de modifier les low-water marks en réception et émission (voir chapitre 4).

8. `SO_RCVTIMEO` et `SO_SNDTIMEO`

Elles permettent de mettre une limite de temps sur les émissions et receptions (write et read). Elles utilisent une structure timeval (cf chapitre 4).

9. `SO_REUSEADDR`

Elle a plusieurs utilités :

- Elle permet à un serveur de démarrer et de se lier à un port même s'il existe des connexions précédentes établies sur ce même port.

On rencontre cette situation dans le scénario suivant :

- o Un serveur est démarré.
- o Il crée un fils pour traiter une connexion entrante.
- o Le père meurt mais le fils continue à traiter avec le client.
- o Le serveur père est redémarré.

Par défaut, lors du redémarrage du serveur, le bind retourne une erreur, car le port est déjà utilisé.

- Elle permet à un nouveau serveur d'être démarré sur le même port qu'un serveur existant lié à l'adresse wildcard, si chaque instance se lie à une adresse IP locale différente. Ce cas de figure se présente notamment pour un site hébergeant de multiples serveurs http avec des alias IP. Il faut des IP différents car TCP interdit le "completely duplicate binding".
- Elle permet à un processus de lier le même port à plusieurs sockets à condition de spécifier un adresse IP local différente, Ceci est notamment utilisé par les serveurs UDP.
- Elle permet le "completely duplicate binding" avec les sockets UDP, ce qui est utilisé dans le multicasting.

IV. Options au niveau IPv4

Le niveau sera à `IPPROTO_IP` dans `getsockopt` et `setsockopt`.

1. `IP_OPTIONS`

Elle permet de modifier les options dans l'entête IPv4. Il faut une connaissance intime d'IPv4 pour l'utiliser.

2. `IP_TTL`

Elle permet de modifier le TIME TO LIVE d'un paquet.

V. Options au niveau TCP

Le niveau sera à `IPPROTO_TCP` dans `getsockopt` et `setsockopt`.

Programmation Réseau en C sous Unix

1. TCP_MAXSEG

Elle permet de récupérer ou modifier le MSS d'une connexion TCP. Le Maximum Segment Size est la quantité maximale de données que TCP enverra au pair. C'est une valeur inférieure ou égale au MSS envoyé par le pair dans le SYN de connexion. On ne peut augmenter la valeur du MSS.

2. TCP_NODELAY

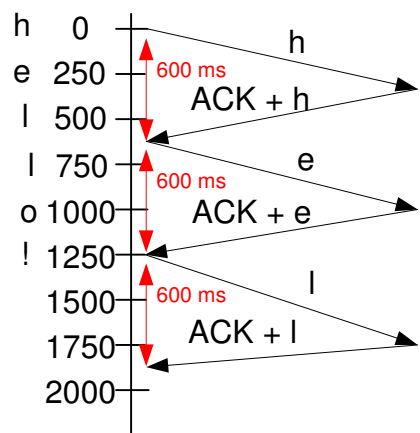
Quand elle est activée, cette option inhibe l'algorithme Nagle de TCP, qui est activé par défaut. Le but de Nagle est de réduire le nombre de paquets de petite taille sur les WAN. Le principe est que s'il y a des données en attente d'acknowledgement sur une connexion, alors aucun petit paquet ne sera envoyé en réponse à un write avant que les données existantes n'aient été accusées. Un petit paquet est un paquet de taille inférieure au MSS.

Nagle va permettre d'éviter qu'il y ait de multiples petits paquets en attente acknowledgement en même temps. Telnet génère de nombreux petits paquets car il en crée un par caractère tapés. Nagle est indécélable sur un LAN mais sur un WAN, le temps d'acknowledgement entre chaque caractère peut créer un log dans la saisie.

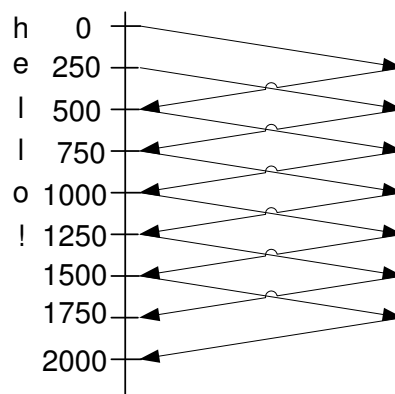
Exemple :

On tape 6 caractères "Hello!" dans telnet avec 250ms entre chaque caractère. Le RTT du serveur est 600ms. Le serveur retourne immédiatement au client le caractère reçu avec le ACK.

Avec Nagle :



Sans Nagle :



Programmation Réseau en C sous Unix

Nagle peut aussi interférer avec un autre algo, celui des ACK retardés. Ce dernier amène TCP à ne pas envoyer un ACK dès la réception de données, mais à attendre de 50 à 200ms. Ainsi, il y a des chances que dans cet intervalle, il y ait des données à joindre au ACK.

Il se pose alors un problème pour les clients auxquels le serveur ne retourne pas de données car ils ne pourront, à cause de Nagle, envoyer de données au serveur avant de recevoir son ACK retardé. Ils doivent donc désactiver Nagle.

Programmation Réseau en C sous Unix

Chapitre 6 : Threads

I. Introduction

Jusqu'à présent, pour créer un serveur concurrent, nous avons utilisé la création de processus avec `fork` afin de gérer les connexions indépendamment de leur établissement. Néanmoins, les processus peuvent poser quelques problèmes :

- La création de processus est une opération couteuse en temps machine : duplication de zones mémoires, de descripteurs, etc...
- Pour passer des informations entre un père et un fils, il faut utiliser des mécanismes spécifiques d'IPC.

Les threads (ou processus légers) permettent de résoudre ces 2 problèmes :

- La création d'un thread prend de 10 à 100 fois moins de temps que celle d'un processus.
- Les threads d'un même processus partagent le même espace mémoire global.

Un thread est un fil d'exécution dans un processus. Un processus possède au moins un thread.

II. Rappels

1) Données partagées

Les threads, en plus de l'espace mémoire globales, partagent d'autre données :

- Le code
- Les descripteurs de fichiers ouverts
- Les gestionnaire de signaux
- Le répertoire courant
- UID et GID

Chaque thread à son propre :

- Identifiant (TID)
- Compteur de programme
- Pile (variables locales et appels de fonctions)
- Errno
- Masque de signaux

2) Création et fin de thread

Un thread est créé par la fonction :

Programmation Réseau en C sous Unix

```
#include <pthread.h>
int pthread_create( pthread_t *tid, const pthread_attr_t *attr, void
*(*func) (void*), void *arg);
```

`tid` : identifiant du thread retourné si la création a fonctionné.

`attr` : attributs du thread (généralement initialisé à NULL ou à la valeur par défaut).

`func` : fonction support du thread c'est-à-dire le code exécuté par le thread. Elle prend en argument un pointeur générique `arg` et retourne un pointeur générique. On pourra ainsi lui passer les données que l'on veut et lui faire retourner ce que l'on veut.

`arg` : argument de `func`.

Exemple :

```
pthread_t tid;
pthread_create(&tid, NULL, serr-fils, (void*) connfd);
```

Un thread peut se terminer de plusieurs façons :

- La fonction support du thread appelle `return`. La valeur de retour sera alors le statut de sortie du thread.
- Si la main du processus se termine ou si un des threads du processus appelle `exit`.
- En appelant la fonction `void pthread_exit(void *status)`. La valeur de `status` sera récupérée par `pthread_join`.
ATTENTION : `status` ne doit pas pointer sur un objet local, celui-ci étant détruit avec le thread.

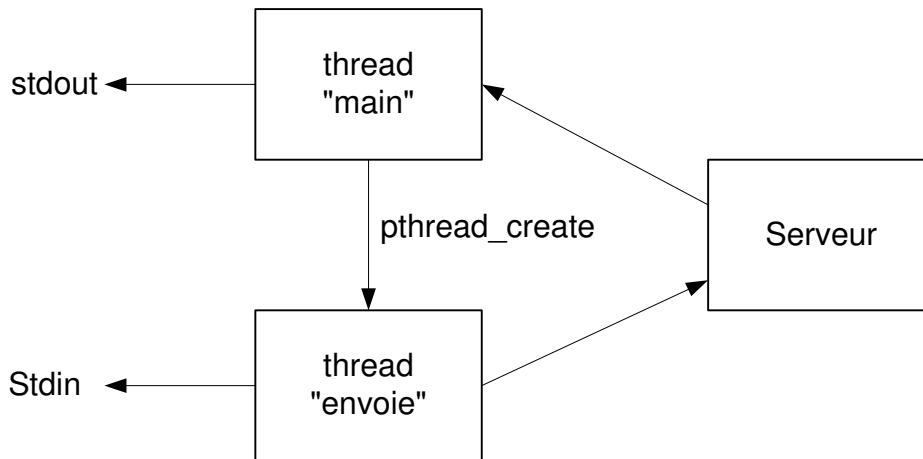
3) Gestion des threads

- Un thread peut récupérer son identifiant avec la fonction : `pthread_t pthread_self()` ?
- On peut attendre la fin d'un thread avec la fonction : `int pthread_join(pthread_t tid, void *status)`.
`tid` : identifiant du thread à attendre
`status` : si `status` est non-nul, il contiendra le statut de sortie de `return` ou de `pthread_exit`.
- Un thread peut avoir 2 états : joint (par défaut) ou détaché. Quand un thread joint se termine, son `tid` et son statut de sortie sont concernés jusqu'à ce qu'un autre thread appelle `pthread_join`. Un thread détaché agit comme un processus démon : quand il se termine, toutes ses ressources sont libérées et on ne peut l'attendre. On rend un thread détaché avec : `int pthread_detach(pthread_t tid)`.

Programmation Réseau en C sous Unix

III. Client Echo et threads

Nous allons réécrire le client Echo vu précédemment sous 2 formes (stop-and-wait et select) avec des threads. On aura un thread chargé de lire l'entrée de données et d'envoyer le résultat dans le socket et un thread chargé de lire dans le socket et d'afficher le résultat.



Il faudra que le thread "envoi" ait accès à 2 données du thread principale :

- Le pointeur de fichier d'entrée, passé en paramètre au programme.
- Le descripteur de socket connectée.

Pour cela, il y a 2 solutions :

1. Déclarer les 2 variables en global.
2. Les mettre dans une structure et la passer au thread dans le pthread_create.

Client :

```
int sockfd_g ; //En global
FILE *fp_g ; //En global

void client(FILE *fp, int sockfd)
{
    char recvline[MAXLINE] ;
    pthread_t tid;
    sockfd_g = sockfd ;
    fp_g = fp ;
    pthread_create(&tid, NULL, envoie, NULL);
    while(read(sockfd, recvline, MAXLINE)>)
        fputs(recvline, stdout);
}

void *envoie(void *arg)
{
    char sendline[MAXLINE];
    while(fgets(sendline, MAXLINE, fp_g) != NULL)
```

Programmation Réseau en C sous Unix

```
        write(sockfd_g, sendline, strlen(sendline));
        shutdown(sockfd, SHUT_WR);
        return(NULL);
    }
```

IV. Serveur Echo et threads

Nous allons réécrire notre serveur Echo concurrent en remplaçant la création de fils avec fork par la création de threads.

Code :

```
.
.
.
for( ; ; )
{
    connfd = accept(listenfd, cliaddr, &len) ;
    pthread_create(&tid, NULL, &serv, (void *) connfd) ;
}

void *serv(void *arg)
{
    pthread_detach(pthread_self());
    serveur_echo((int) arg);
    close((int) arg);
}
```

On passe connfd en paramètre à la fonction support des threads, serv, en le transtypant en void *. Dans cette fonction, chaque thread commence par se détacher du tread principal. Ainci, celui-ci n'aura pas à les attendre.

Ensuite, la fonction de traitement, serveur-echo, est appelée avec le descripteur de socket connectée passé via arg.

Enfin, chaque thread doit fermer explicitement le socket connectée (la fin d'un thread n'entraîne pas la fermeture de ses descripteur comme avec les processus).

Cette version peut passer quelques problèmes sur les systèmes où le transtypage d'un entier par un void * ne fonctionne pas (c'est le cas si sizeof(int)<=sizeof(pointeur)). Il faut donc modifier le passage d'argument pour rendre notre serveur portable.

La solution la plus simple serait de passer comme argument au thread l'adresse de connfd :

```
pthread_create(&tid, NULL, &serv, &connfd) ;

void *serv(void *arg)
{
    int connfd;
```


Programmation Réseau en C sous Unix

```
connfd = *((int *) arg);  
.  
.  
.  
}
```

Dans le thread principal la valeur de `connfd` sera écrasée à chaque nouvelle connexion, mais son adresse sera la même.

Tous les threads récupéreront la valeur de `connfd` à la même adresse, ce qui peut poser des problèmes selon l'ordonnement :

- Une connexion est établie, `accept` retourne le descripteur n° 6 pour `connfd`.
- Le thread 1 est créé et on lui passe l'adresse de `connfd`.
- Avant que le thread ne démarre, une nouvelle connexion s'établit et le thread principale récupère la main .
- `Accept` retourne `connfd = 7`. Le thread 2 est créé et on lui passe l'adresse de `connfd`.
- Quand thread 1 reprendra son exécution , il récupérera la valeur de `connfd` à l'adresse qu'on lui a passé, et on aura `connfd = 7`.
- Ce sera la même chose pour thread 2. Les 2 threads agiront sur la même connexion.

Pour éviter cela, on va stocker `connfd` à une adresse différente pour chaque connexion.

Version portable :

```
int *pconnfd ;  
for( ; ; )  
{  
    pconnfd = (int *) malloc (sizeof (int)) ;  
    *pconnfd = accept(listenfd, cliaddr, &len);  
    pthread_create(&tid, NULL, &serv, pconnfd);  
}  
  
void *serv(void *arg)  
{  
    int connfd;  
    connfd = *((int *) arg);  
    free(arg);  
    .  
    .  
    .  
}
```

Programmation Réseau en C sous Unix

Chapitre 7 : Entrées /Sorties avancées

I. Temporisation et socket

Nous allons voir comment introduire une limite de temps sur une opération d'E/S sur un socket.

Il y a 3 méthodes pour cela :

- Utiliser la fonction `alarm` qui envoie au processus appelant le signal `SIGALARM` après un nombre de secondes définies.
- Utiliser la fonction `select`, qui permet de définir un temps maximal d'attente, au lieu de `read` ou `write`.
- Utiliser les options `SO_RCVTIMEO` et `SO_SNDTIMEO`.

Ces trois méthodes fonctionnent avec les opérations de lecture et écriture. Il peut aussi être intéressant de placer une limite de temps sur le `connect`. Or, les options ne fonctionnent pas sur `connect` et `select` ne fonctionne avec `connect` que si la socket est non bloquante.

1) Limite de temps sur connect

On utilise la méthode "alarm". On va construire une fonction `connect_time` qui prendra 4 arguments : les trois de `connect` et un nombre de secondes.

Le code de cette fonction sera :

```
signal(SIGALARM, sig_connect);
alarm(nsec);
if((n = connect(sockfd, servaddr, len)) < 0) //si connect a été
{                                           //interrompu
    close(sockfd);
    if(errno == EINTR)
    {
        printf("connect timeout\n");
        errno = ETIMEDOUT;
    }
}
alarm(0);

void sig_connect(int s) //traitement du signal
{
    return;
}
```

2) Limite de temps avec select

Par exemple, pour placer une limite de temps avec `select`, sur une opération de lecture, on va écrire une fonction qui dira si un descripteur est prêt en lecture dans un intervalle de temps donné.

Programmation Réseau en C sous Unix

```
int readable_timeo(int fd, int nbsec)
{
    fd_set rset;
    struct timeval tr;
    int nb;
    FD_ZERO(&rset);           //fd à surveiller
    FD_SET(fd, rset);        //en lecture.
    tv.tv_sec = nbsec;       //Temps
    tv.tv_usec = 0;          //maximal.
    nb = select(fd + 1, &rset, NULL, NULL, &tr);
    Return(nb);
}
```

Si `readable_timeo` retourne 0, c'est que le temps s'est écoulé sans que le descripteur soit prêt, sinon, on peut effectuer l'opération de lecture.

3) Limite de temps avec `SO_RCVTIMEO` :

On utilise l'option de socket `SO_RCVTIMEO` pour mettre une limite de temps en réception. Pour cela, on définit le temps maximal dans une structure `timeval` et on appelle `setsockopt` sur le descripteur du socket.

Par rapport aux solutions précédentes, on active l'option une fois et la limite de temps s'applique pour toutes les lectures.

On ne peut par contre, avec cette méthode, ne gérer le temps que sur une lecture (`SO_RCVTIMEO`) ou une écriture (`SO_SNDTIMEO`), pas sur un connect.

Exple :

```
void fonc_cli( FILE *fp, int sockfd){
    int n;
    char sendline[MAXLINE], recvline[MAXLINE+1];
    struct timeval tv;
    tv.tv_sec = 5;           //limite de
    tv.tv_usec = 0;          //temps = 5 secondes
    setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
    //Active l'option
    while(fgets(sendline, MAXLINE, fp) != NULL){
        write(sockfd, sendline, strlen(sendline));
        n = read(sockfd, recvline, MAXLINE);
        if(n < 0){
            if(errno == EWOULDBLOCK){ //erreur indiquant le
                //dépassement de temps limite
                printf("Temps dépassé\n");
                continue;
            }else{
                printf("erreur\n");
                exit(-1);
            }
        }
        recline[n] = 0;
    }
}
```

Programmation Réseau en C sous Unix

```
}  
}
```

II. Lecture et écriture avancées :

a. recv et send :

Ce sont des fonctions équivalentes à read et write avec en argument supplémentaire.

```
#include <sys/socket.h>  
ssize_t recv(int sockfd, char *buff, ssize_t taille, int flags)  
ssize_t send(int sockfd, char *buff, ssize_t taille, int flags)
```

sockfd, buff, taille sont identique à ceux de read et write.

flags : option supportées :

- MSG_DONTROUTE : informe le noyau que la destination est locale et qu'il n'a pas à consulter les tables de routages.
- MSG_DONTWAIT : rend l'opération non bloquante.
- MSG_PEEK : permet de regarder les données disponible dans le socket sans les enlever.
- MSG_WAITALL : indique au noyau qu'il doit attendre que le nombre d'octets demandés soit effectivement lus avant que recv ne se termine.

b. readv et writev :

Similaires à read et write, elles permettent de lire ou d'écrire à partir de ou vers plusieurs buffers en un seul appel. C'est le scatter read (les données d'entrée sont dispersées dans de multiples buffers) et le gather write (les données à écrire sont réunies à partir de plusieurs buffers).

```
#include <sys/uio.h>  
ssize_t readv(int filedes, struct iovec *iov, int iovcnt)  
ssize_t writev(int filedes, const struct iovec*iov, int iovcnt)  
  
struct iovec{  
    void *iov_base;           //adresse du buffer  
    size_t iov_len;          //taille du buffer  
}
```

iov est un tableau de structures iovec. Ce sont les buffers où seront rangées les données lues. iovcnt est le nombre de cases de ce tableau.

c. recvmsg et sendmsg :

Ce sont les fonctions d'E/S les plus évoluées. Elle peuvent remplacer toutes les autres vues précédemment.

```
#include <sys/socket.h>  
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags)  
ssize_t sendmsg(int sockfd, struct msghdr *msg, int flags)
```

Les arguments sont stockés dans la structure msghdr.

Programmation Réseau en C sous Unix

```
struct msghdr{
    void *msg_name;           //Utilisés par les
    socklen_t msg_namelen;   //sockets non connectés
    struct iovec *msg_iov;   //Tableau pour le scatter/gather
    int msg_iovlen;         //Nombre de cases de ce tableau
    void *msg_control;      //Pour les données
    socklen_t msg_controllen; //ancillaires (ce contrôle)
    int msg_flags;         //Option : utilisé uniquement
                          //par recvmsg
}
```

L'argument flags est utilisé comme dans recv et send.

Programmation Réseau en C sous Unix

CHAPITRE 8 : Etude de différents C/S

I. Introduction :

Il y a plusieurs possibilités pour écrire un serveur. Les solutions vues jusqu'à maintenant sont :

- Le serveur itératif : 1 seul processus.
- Le serveur concurrent multi-processus.
- Le serveur concurrent multi-threads.

Nous allons décrire deux nouvelles solutions :

- Le serveur "préforké" : au démarrage, le serveur crée un ensemble de processus qu'il distribuera au fur et à mesure des connexions.
- Le serveur "préthreadé" : idem avec des threads.

Nous testerons ensuite toutes ces possibilités afin de les comparer.

II. le client de test :

Afin de faire une comparaison équitable, nous utiliserons le même client pour tous les serveurs. Ce client initiera un grand nombre de connexions simultanées demandant un nombre d'octets définis.

la ligne de commande du client sera :

```
client IPdest Port nbclient nbconnexions nboctets
```

`nbclient` : nombre de client simultané

`nbconnexions` : nombre de connexions par client

`nboctets` : nombre d'octets demandés

III. le client de test :

Pour comparer de manière significative les performances de nos serveurs, nous allons utiliser la fonction `getrusage` qui retourne l'utilisation de ressources d'un processus, et éventuellement de ses fils. On aura à la fois le temps utilisateur total et le temps système total.

Nous allons donc utiliser une fonction qui calculera ces 2 temps. Elle sera appelée quand l'utilisation interrompera le serveur avec le signal `SIGINT` (CTRL + C), ce qu'il fera quand le client aura terminé ses requêtes.

La fonction de calcul de temps sera :

```
void pr_cpu_time() {
    double user, sys;
    struct rusage myusage, childusage;
    getrusage(RUSAGE_SELF, &myusage); //récupération du temps du
```

Programmation Réseau en C sous Unix

```
processus.  
    getrusage(RUSAGE_CHILDREN, &childusage); //récupération du  
temps de ses fils.  
    user = (double)myusage.ru_utime.tv_sec +  
myusage.ru_utime.tv_usec;  
    user += (double)childusage.ru_utime.tv_sec +  
childusage.ru_utime.tv_usec;  
    sys = (double)myusage.ru_stime.tv_sec +  
myusage.ru_stime.tv_usec;  
    sys += (double)childusage.ru_stime.tv_sec +  
childusage.ru_stime.tv_usec;  
    printf("temps utilisateur = %g, temps système = %g\n", user,  
sys);
```

Le serveur devra donc appeler cette fonction quand CTRL + C sera pressé par l'utilisateur.

IV. Serveur pré-forké :

Au lieu de créer un fils à chaque connexion demandé, le serveur va créer un ensemble de fils dès son démarrage. Ainsi, il passera chaque connexion à un de ses fils existants.

L'avantage principal, c'est que les clients n'ont pas à attendre le temps d'un fork pour échanger avec le serveur.

Le problème, c'est celui du nombre de fils à créer initialement. Si le nombre de clients égale celui des fils, alors les clients suivants devront attendre la fin d'une connexion avant d'être servis.

Une solution serait de surveiller le nombre de fils disponibles et s'il descend sous une limite fixées, on en crée de nouveaux. De même, si trop de fils sont inutilisés, on en supprime quelques-uns.

Le père aura le fonctionnement suivant :

- Création de la socket passive.
- Création de n fils et sauvegarde de leurs PID dans un tableau.
- Interception du signal SIGINT :
 - mettre fin à tous les fils.
 - calculer les temps d'exécutions.
- Mise en pause.

Les fils auront la structure d'un serveur classique :

boucle infinie :

- accept
- traitement de la connexion.

Tous les N fils partagent la même socket passive. De plus, ils sont tous bloqués dans l'appel à accept. Lorsque la première connexion arrive, tous les fils sont réveillés, un récupère la connexion et N-1 se rendorment.

Programmation Réseau en C sous Unix

S'il y a trop de fils, cela peut affecter les performances. C'est le "Thundering herd" problem.

V. Serveur préthreadé (un accept par thread) :

Nous allons créer un certain nombre de threads au démarrage du processus. Chacun des threads appellera la fonction `accept`, mais, pour éviter le "thundering herd" problem, on va donner un accès en exclusion mutuelle à la fonction `accept`.

Ainsi, seul un thread sera endormi par le `accept` et donc sera réveillé par l'arrivée d'une connexion. Les autres seront endormis dans le mutex.

Le thread principal fonctionnera comme le serveur père précédent. Chaque thread aura le comportement suivant :

boucle infinie :

- acquisition du mutex
- `accept`
- libération du mutex
- gestion de la connexion

Même si cette exclusion mutuelle n'est pas obligatoire, les tests démontrent que cette version est moins gourmande que sans exclusion mutuelle.

VI. Serveur préthreadé (un accept dans le main)

Notre serveur va créer un ensemble de threads à son démarrage mais nous n'aurons qu'un `accept` dans le thread principal. Le problème qui se pose, c'est comment passer le descripteur de socket connectée à un des threads disponibles.

Il n'est pas nécessaire de passer le descripteur, il suffit que le thread obtienne le numéro du descripteur qui lui est attribué. On va créer un tableau appelé `clifd` où le thread principal stockera les descripteurs de sockets connectées. Pour se localiser dans le tableau, on utilisera 2 indices : `iput` qui indiquera où stocker le prochain descripteur et `iget` qui indiquera où se trouve le prochain descripteur disponible pour les threads. Les accès à ce tableau de vront se faire en exclusion mutuelle. `iput`, `iget` et `difd` seront en effet partagés par tous les threads.

Dans le thread principal, il faudra, à chaque nouvelle connexion, vérifier que `iput` n'a pas rejoint `iget`. Si c'est le cas, c'est que le tableau est trop petit.

Dans les threads secondaires, si `iget == iput`, elles n'ont rien à faire, si ce n'est attendre que cela change. Elles s'endormiront dans avec une variable conditionnelle. C'est le thread principal qui les réveillera.

Thread principal :

```
//création socket
for( ; ; ){
    connfd = accept(...);
    pthread_mutex_lock(&clifd_mutex);
```


Programmation Réseau en C sous Unix

```
    clifd[iput] = connfd;
    if(++iput == MAXNCLI) iput = 0; //iput est arrivé à la fin du
    tableau
    if(iput == iget){
        printf("erreur : iput = iget = %d\n",iput);
        exit(-1) ;
    }
    pthread_cond_signal(&clifd_cond); //on signale que le tableau
    a été modifié
    pthread_mutex_unlock(&clifd_mutex);
}
```

Thread secondaire :

```
for( ; ; ){
    pthread_mutex_lock(&difd_mutex);
    while(iget == iput)
        pthread_cond_wait(&clifd_cond); //plus de connexion à
    traiter donc pause.
    connfd = clifd[iget];
    if(++iget == MAXNCLI) iget = 0; //iget atteint la fin du
    tableau
    pthread_mutex_unlock(&clifd_mutex);
    serv(connfd); //fonction de traitement de la connexion
    close(connfd);
}
```

VII. Conclusion

Les résultats des mesures, pour 5 clients simultanés réclamant 4000 octets de données au serveur, et ce, 500 fois chacun (soit 2500 connexions TCP) sont, du plus lent au plus rapide :

- Serveur concurrent avec création d'un fils par connexion
- Serveur préthreadé avec un accept dans le main.
- Serveur préthreadé avec un accpet par thread.
- Serveur préforké
- Serveur multithread.

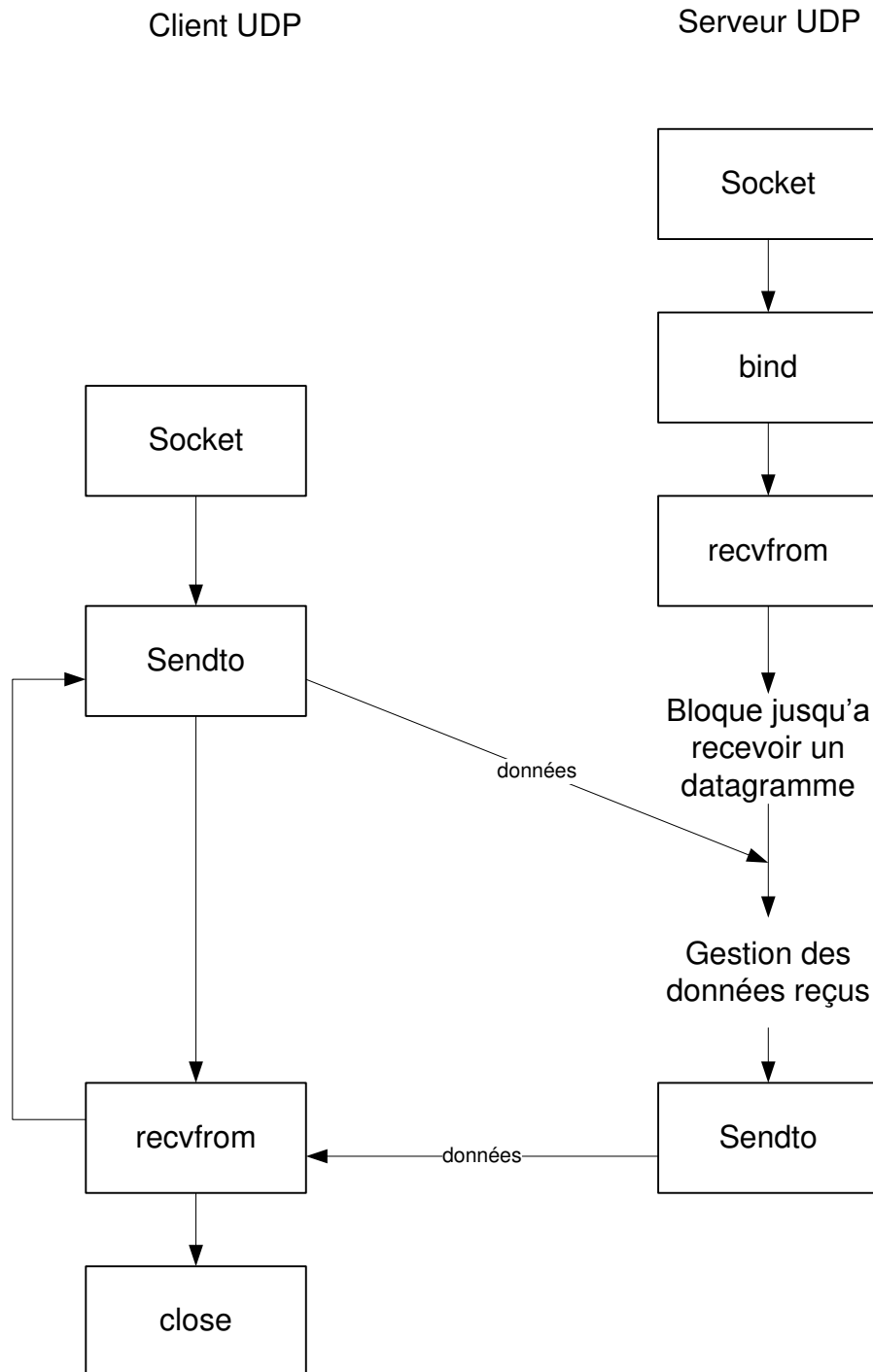
Les 4 derniers sont environ 10 fois plus rapide quele serveur concurrent.

Programmation Réseau en C sous Unix

Chapitre 9 : Sockets UDP

I. Introduction :

Il y a des différences fondamentales entre UDP et TCP, qui se retrouvent lorsqu'on écrit des applications : UDP est non-connectés et non-fiable. Les fonctions à appeler pour un client/serveur UDP sont :



Programmation Réseau en C sous Unix

II. Les fonctions `recvfrom` et `sendto` :

Elles sont similaires à `read` et `write` mais requièrent 3 arguments supplémentaires :

```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
struct sockaddr *from, socklen_t *addrlen)
```

Les 3 premiers arguments sont les mêmes que ceux de `read`.

`flags` : voir cours « E/S avancées ».

`from` : contiendra les « coordonnées » de la provenance des données reçues.

`addrlen` : contient la taille de la structure d'adresse de l'expéditeur.

```
ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int
flags, const struct sockaddr *to, socklen_t addrlen)
```

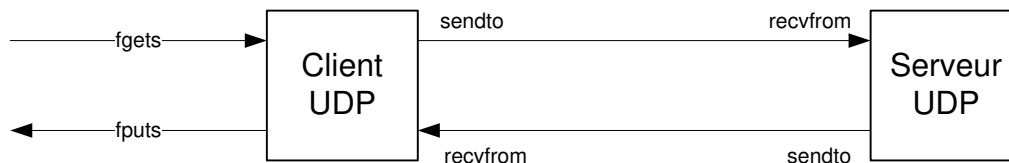
Les 3 premiers arguments sont les mêmes que ceux de `write`.

`to` : coordonnées du destinataire du datagramme

`addrlen` : taille de `to`

les 2 fonctions retournent la taille des données lues ou écrites. Il est possible d'envoyer un datagramme de taille 0 : il y aura 20 octets d'entête IP, 8 octets d'entête UDP et 0 octets de données. Un `recvfrom` qui retourne 0 ne signifie pas que le pair a fermé la "connexion".

III. Serveur Echo UDP :



Les différences avec le serveur Echo TCP sont :

- Le second argument de la fonction `socket` vaudra `SOCK_DGRAM`.
- Après le `bind`, plus de `accept`, le serveur se met immédiatement en `listen`.

Ce serveur, comme la plupart des serveurs UDP, sera itératif. Il y a donc un seul `socket`, et les datagrammes reçus seront stockés dans une file d'attente FIFO, quelle que soit leur provenance.

IV. Client Echo UDP :

La aussi le 2^{ème} argument de `socket` vaudra `SOCK_DGRAM`. On n'utilise plus la fonction `connect`, mais directement `sendto`. Le port éphémère qu'utilisera le client, déterminé en TCP lors d'un `connect`, sera, en UDP, donné par le premier appel à `sendto`.

Programmation Réseau en C sous Unix

V. Authentification de la réponse :

Tout hôte connaissant le port d'écoute du client peut lui envoyer des réponses, qui seront mélangés aux vraies réponses du serveur. Pour authentifier la réponse, il faut récupérer avec `recvfrom` l'adresse de provenance de chaque datagramme et le comparer avec celle de destination.

Exple :

```
...
sendto(sockfd, sendline, strlen(sendline), 0, &servaddr, servlen);
len = servlen;
n = recvfrom(sockfd, recvline, MAXLINE, 0, &reply, &len);
if(len != servlen || memcmp(&servaddr, &reply, len) != 0){
    printf("réponse de provenance inconnue\n");
    continue;
}
...
```

Cette méthode fonctionne parfaitement avec un serveur qui a une seule adresse IP, mais peut poser des problèmes avec un serveur à plusieurs interfaces. Si ce type de serveur se lie à l'adresse wildcard, ce sont les fonctions de routage du noyau qui choisiront l'adresse de sortie pour répondre au client. Ainsi, si la réponse ne provient pas de la même interface contactée par le client, le datagramme sera refusé de façon erronée.

Pour résoudre cela, il faut créer un socket par adresse IP du serveur et le lier explicitement à cette adresse. On les surveillera avec un `select`. Les réponses seront bien envoyées de la socket qui les a reçues.

VI. Erreur de serveur :

Nous allons examiner ce qui se passe si le client envoie des données alors que le serveur n'est pas démarré. Le client appelle `sendto` et se bloque dans `recvfrom`.

`Sendto`, même s'il n'a pas pu fonctionner correctement, ne retourne pas d'erreur.

Avec `tcpdump`, on s'aperçoit qu'une erreur ICMP "port unreachable" a été générée lors de l'envoi.

Cette erreur, dite asynchrone, n'est pas retournée au client. En effet, l'appel à `recvfrom` ne permet pas de récupérer le paquet ICMP. Ce type d'erreur est signalé uniquement lorsqu'on utilise `connect` en UDP.

VII. Connect et UDP :

On peut utiliser la fonction `connect` avec un socket UDP. Cela n'entraîne pas une connexion comme avec TCP, mais permet de récupérer les erreurs et d'enregistrer l'adresse IP et le port du pair.

Il y a 3 différences entre les sockets UDP connectés et non-connectés :

- On ne peut plus spécifier pour chaque envoi le destinataire. On n'utilisera plus `sendto` mais `write` et `send`.

Programmation Réseau en C sous Unix

- Il n'est plus nécessaire de récupérer les coordonnées de l'expéditeur lors d'une réception. On utilisera read et recv à la place de recvfrom.
- Les erreurs asynchrones seront transmises au processus.

Un client ou serveur UDP utilisera connect sur un socket UDP seulement s'il utilise cette socket pour communiquer avec un hôte unique. (Exple : un client DNS configuré pour contacter qu'un serveur).

VIII. Serveur TCP et UDP :

Nous allons écrire un serveur capable de gérer des requêtes TCP et des requêtes UDP. Nous combinerons un serveur concurrent TCP et un serveur itératif UDP. Pour cela, on va créer 2 sockets, une UDP et une TCP, et on va surveiller les 2 descripteurs avec la fonction select. Une fois les 2 sockets créées, on aura :

```
FD_ZERO(&rset);
maxfdp1 = max(udpfd, listenfd)+1;
for( ; ; ){
    FD_SET(listenfd, &rset);
    FD_SET(udpfd, &rset);
    if((nready=select(maxfdp1, &rset, NULL, NULL, NULL))<0){
        if(errno == EINTR){
            Continue;
        }
        else{
            printf("erreur select\n");
            exit(-1);
        }
    }
    if(FD_ISSET(listenfd, &rset)){
        len = sizeof(cliaddr);
        connfd = accept(...);
        if((pid = fork()) == 0){
            close(listenfd);
            traitementserveur(connfd);
            exit(0);
        }
        Close(connfd);
    }
    if(FD_ISSET(udpfd, &rset)){
        len = sizeof(cliaddr);
        n = recvfrom(...);
        sendto(...);
    }
}
}
```

IX. Choix TCP/UDP :

Même en connaissant les différences entre TCP et UDP (la fiabilité), on peut se poser la question : quand utiliser UDP à la place de TCP ?

Les avantages d'UDP sont les suivants :

Programmation Réseau en C sous Unix

- Il permet le multicasting et le broadcasting, pas TCP.
- Pour un simple échange requête/réponse, il ne demande que 2 paquets, contre 11 pour TCP si c'est une nouvelle connexion ;

De son coté, TCP apporte des fonctionnalités par défaut qui devront être ajoutées à UDP en cas de besoin :

- Accusés de réception, retransmission des paquets perdus, détection des doublons et séquençage des paquets réordonnés.
- Contrôle du flux de données par le receveur (fenêtrage).
- Contrôle du flux par l'expéditeur (congestion avoidance).

Pour résumer :

- UDP doit être utiliser pour :
 - o Multicasting et broadcasting : L'ajout de contrôle d'erreur en multicasting est compensé par le gain de performances entre envoyer le paquet à N destinataires et envoyer N paquets via N connexion TCP.
- UDP peut être utilisé pour :
 - o Des applications du type requête/réponse. Néanmoins, il faut ajouter le contrôle d'erreur, soit, au minimum, accusés de réception, délais et retransmission. Le contrôle de flux est généralement inutile si les requêtes et réponses ne sont pas trop grandes.
- UDP ne devrait pas être utilisé pour :
 - o Des transferts important de données . En effet, l'ajout de toutes les fonctionnalités de base de TCP revient à réinventer TCP.
- Une exception à ce dernier point est TFTP. Son utilisation courante est le chargement du bootstrap sur des terminaux. Or, le code requis en UDP est 5 fois moins long qu'en TCP et les transferts ne se font que sur un LAN, pas sur un WAN.

X. Fiabilité et UDP :

Comme on vient de le voir, si on veut créer une application requête/réponse en UDP, il faut implémenter 2 fonctionnalités supplémentaires à notre client/

- Les numéros de séquences pour vérifier qu'une réponse correspond bien à une requête.
- Les délais et retransmissions pour gérer les paquets perdus.

La plupart des applications UDP existants utilisent ces 2 fonctionnalités : DNS, TFTP, SNMP et RPC. L'ajout des numéros de séquences est facile. Il suffit, dans chaque requête, d'accoler un numéro de séquences que le serveur retournera avec la réponse.

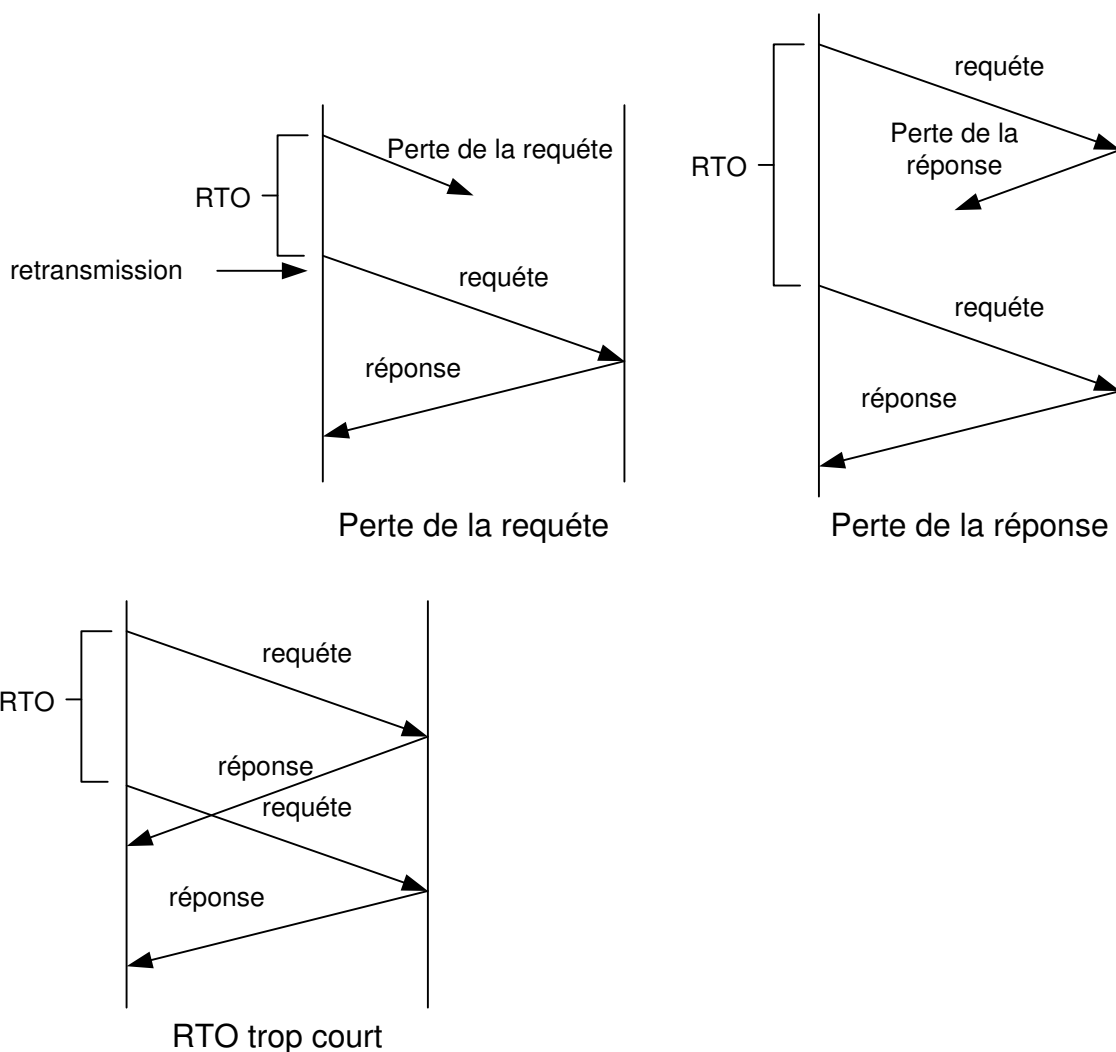
Programmation Réseau en C sous Unix

Le problème qui se pose, c'est pour le délai avant retransmission. En effet, le temps requis pour un datagramme pour effectuer un aller-retour varie énormément selon le réseau, la distance et l'encombrement, et varie dans le temps.

On utilise donc pas directement le RTT pour estimer le délai. On utilisera le RTO (Recovery Time Objective) calculé à partir des RTT.

Il se calcule avec l'algorithme de jacobson, qui introduit aussi la notion d'attente exponentielle : si un RTO de 2s ne suffit pas à recevoir la réponse, le prochain sera de 4s, etc...

Néanmoins, il subsiste un problème : l'ambiguïté de retransmission. 3 scénarios peuvent se produire :



Quand le client reçoit une réponse pour une requête retransmise, il ne peut savoir à quelle requête elle correspond.

Programmation Réseau en C sous Unix

Chapitre 10 : Les raw sockets

I. Introduction

Elles apportent des fonctionnalités supplémentaires par rapport aux sockets classiques TCP et UDP :

- Elles permettent d'écrire des paquets ICMP et IGMP.
- Elles permettent à un processus de pouvoir recevoir et émettre des datagrammes IPv4 avec un champ protocole non supporté par le noyau. (La plupart des noyaux ne supportent que les valeurs suivants pour ce champ : 1 (ICMP), 2 (IGMP), 6 (TCP), 17 (UDP)). Par exemple un champ protocole égal à 89, le programme qui l'implémente utilise donc une raw socket.
- Elles permettent à un processus de créer sa propre entête IPv4, grâce à l'option IP_HDRINCL.

II. Création d'une socket RAW

Les étapes permettent de créer une raw socket sont :

- La fonction socket prend comme second argument la valeur SOCK_RAW :

```
int sockfd;  
sockfd = socket(AF_INET, SOCK_RAW, protocole);
```

Seul le super-utilisateur peut créer des sockets raw.

- Si on veut créer sa propre entête IP, on doit activer l'option IP_HDRINCL :

```
const int on = 1;  
setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
```

- Bind est très rarement utilisé sur une raw socket. Il n'y a pas de notion de port. Si bind est utilisé, c'est uniquement pour donner l'adresse de sortie des datagrammes.
- Connect est rarement utilisé sur une raw socket et permet d'utiliser write au lieu de sendto, l'adresse de destination étant donnée.

III. Envoi sur une socket raw

Il suit les règles suivantes :

- L'envoi se fait avec sendto ou sendmsg en spécifiant l'adresse de destination. Si la socket est connectée, ou utilise write, writev ou send.
- Si l'option IP_HDRINCL n'est pas mise, l'adresse de départ des données à envoyer sera celle du premier octet suivant l'entête IP.
- Si l'option IP_HDRINCL est activée, l'adresse de début des données à envoyer par le noyau sera celle de l'entête IP.

La taille des données à écrire devra inclure celle de l'entête.

Le processus doit réécrire toute l'entête IP sauf :

- o Le champ d'identification IPv4 qui peut être mise à 0 pour laisser le noyau le remplir.

Programmation Réseau en C sous Unix

- Le checksum qui est calculé par le noyau.
- Les options IP.
- Le noyau fragmente les paquets raw qui dépassent le MTU de l'interface de sortie.
- Avec IPv4, c'est à l'utilisateur de calculer les checksums éventuels contenus après l'entête IPv4.

IV. Réception sur une socket raw

La première chose à savoir, c'est quels sont les paquets reçus que le noyau transmet à une socket raw ?

On a les règles suivantes :

- Aucun paquet TCP ou UDP n'est passé par une socket raw.
- La plupart des paquets ICMP sont passés à une socket raw, excepté les paquets echo request, timestamp request et address mask request.
- Tous les paquets IGMP sont passés à une socket raw, après que le noyau ait traité le message IGMP.
- Tous les paquets dont le champ protocole n'est pas compréhensible par le noyau sont passés à une socket raw. Le noyau n'effectue que des vérifications minimales sur l'entête IP : version IP, le checksum IPv4, la taille de l'entête et l'adresse IP de destination.

Quand le noyau doit passer un paquet IP à une raw socket, il examine toutes les raw socket du système et transmet le paquet à toutes les raw sockets correspondantes. Si tous les tests suivants sont concluants, le paquet sera transmis à la raw socket :

- Si un protocole non-nul est spécifié à la création de la socket raw, elle ne recevra que les paquets avec ce protocole.
- Si une adresse locale est liée à la socket avec bind, selon les datagrammes avec cette adresse comme destinataire seront passés.
- Si une adresse distante a été spécifiée avec connect, alors seuls les datagrammes provenant de cette adresse seront transmis.

Si une socket raw est créée avec un protocole de 0, et qu'elle n'appelle ni bind, ni connect, alors elle recevra tous les datagrammes que le noyau passe aux raw sockets.