Un serveur multiplexeur

Ce serveur doit gérer autant de connexions qu'il y a de clients (sur des sockets de service), diffuser les nouveaux messages aux clients connectés et être en permanence en attente de nouvelles connexions (sur la socket d'écoute).

La méthode traditionnelle pour construire une application serveur reseau est de construire un bloc principae anttendant une connexion avec la primitive accept() et de créer un processus fils lorsqu'une demande survient.

La primtive select(), que l'on utilise pour faire du multiplexage permet de construire un serveur autour d'un seul processus qui *multiplexe* les demandes de connexion et les prends en charge. Il y a des avantages à avoir un serveur constitué d'un unique processus : pas besion de primitive de synchronisation ou de connexion entre processus, mais il faut gérer plusieurs connexions.

Nous allons développer une application de chat multiclient pour illustrer ceci.

Le serveur

```c
int main(int argc, char *argv[]) {

    signal(SIGCHLD, SIG_IGN);

    int nbLus;
    char message[BUFFER_SIZE];

    /* 1. Création de la socket d'écoute */

    int secoute = socket(AF_INET, SOCK_STREAM, 0);
    if(secoute == -1){
        perror("Erreur de création de la socket");
    }

    int sservice;

    /* 2. Création de la structure d'adresse */

    struct sockaddr_in saddr = {0};
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(PORT_SERVEUR);
    saddr.sin_addr.s_addr = inet_addr(ADRESSE);

    struct sockaddr_in caddr = {0};
    unsigned int caddrlen;

    /* 3. On attache l'adresse et on ouvre l'écoute */

    if(bind(secoute, (struct sockaddr *) &saddr, sizeof(saddr)) == -1){
```

```
        perror("Erreur lors du bind");
        return 1;
    }

    if(listen(secoute, MAX_CLIENTS) == -1){
        perror("Erreur lors du listen");
        return 1;
    }

    /* 4. Création du set de descripteur */

    fd_set ensemble, temp;
    FD_ZERO(&ensemble);
    FD_SET(secoute, &ensemble);

    int max = secoute; //le descripteur max
    printf("Serveur en attente de nouveaux clients ou messages. \n");
    while(1){

        temp = ensemble;
        if(select(max+1, &temp, NULL, NULL, NULL) == -1){
            printf("Signal reçu \n");
        }

        /* 5. Vérification de l'arrivée de client */

        if(FD_ISSET(secoute, &temp)){
            caddrlen = sizeof(caddr);
            sservice = accept(secoute, (struct sockaddr *) &caddr,
&caddrlen);
            if(sservice > max){
                max = sservice;
            }
            FD_SET(sservice, &ensemble);
            printf("Un nouveau client est connecte \n");
        }

        /* 6. Gestion des clients */
        for (int fd = 0; fd < max+1; fd++){

            /* On ne lit pas sur la socket d'écoute, si la socket fd est
inactive, il n'y a rien à lire donc on saute*/
            if(!FD_ISSET(fd, &temp) || fd == secoute){
                continue;
            }

            nbLus = read(fd, message, BUFFER_SIZE-1);

            if(nbLus == -1){
                perror("Erreur de read");
                FD_CLR(fd, &ensemble); //Retire le descripteur du set
                shutdown(fd, SHUT_RDWR);
                close(fd);
                continue;
```

```
            }
            if(nbLus == 0){
                printf("Déconnexion d'un client \n");
                FD_CLR(fd, &ensemble); //Retire le descripteur du set
                shutdown(fd, SHUT_RDWR);
                close(fd);
                continue;
            }

            if(nbLus > 0){
                write(1, "Message reçu de ", strlen("Message reçu de "));
                write(1, message, nbLus);
            }

            /* Affichage du message reçu chez tout les clients*/
            for (int fdi = 3; fdi <max+1; fdi++){
                if(fdi == fd){
                    continue;
                }
                write(fdi, message, nbLus);
            }
        }
    }
    /* 7. Terminaison du serveur */
    shutdown(secoute, SHUT_RDWR);
    close(secoute);
    return 0;

}
```

1. **Création de la socket d'écoute**

   Comme d'habitude, on crée une socket d'écoute avec la primitive *socket()*. C'est sur cette socket que les clients se connecterons.

   On déclare également une socket de service sservice.

2. **Création de la structure d'adresse**

   On crée la structure d'adresse correspondante aux paramètres réseaux du serveur.

   On déclare également une structure client et une taille pour la structure cliente.

3. **On attache l'adresse et on ouvre l'écoute**

   On utilise les primitives *bind()* et *listen()* pour lié l'adresse à la socket et l'ouvrir aux demandes de connexion.

4. **Création du set de descripteur**

   Ici commence les particularités du multiplexage. On commence par créer un set de descripteur (du type *fd_set*). **Les sets de descripteurs contiennent comme leur nom l'indique une liste de descripteurs de fichiers**. On peut réaliser plusieurs actions sur ces set avec les macros suivantes :

- FD_ZERO(&ensemble) : vide le set de descripteur *ensemble*
- FD_SET(descripteur, &ensemble) : ajoute le descripteur au set *ensemble*
- FD_CLR(descripteur, &ensemble) : retire le descripteur du set *ensemble*
- FD_ISSET(descripteur, &ensemble) : retourne vrai si le descripteur est dans l'ensemble.

et on déclare un set de la façon suivante : *fd_set ensemble* .

Les set sont implémentés comme tableaux de bits, une case par descripteur. Sous Linux : FD_SETSIZE = 1024, sizeof(fd_set) = 128 octets.

La primitive *select()* est basée sur le concept de set. Son prototype est le suivant :

```
int select(int fd_max, fd_set *rds, fd_set *wrs, fd_set *exs, struct
timeval *tv);
```

Les paramètres correspondent à :

- fd_max: le numero du plus grand descripteurs des trois ensemble **+1**.
- rds : pointeur vers le set à examiner en lecture. Nullable si vide.
- wrs : pointeur vers le set à examiner en écriture. Nullable si vide.
- exs : pointeur vers le set à examiner pour un état exceptionnel. Nullable si vide.
- tv : pointeur vers une structure *timeval* donnant le délais d'attente maximum. Nullable pour une attente illimitée.

L'appel à select est bloquant jusqu'a la reception d'un signal ou la fin du temps d'attente. En cas de reception d'un signal par un processus, *select()* renvoie -1.

Les timevals

```
struct timeval {
    long tv_sec; //le nombre de seconde
    long tv_usec; //le nombre de microseconde
};
```

Dans notre cas, nous créeons deux sets : l'ensemble global et un sets temporaire sur lequel on va faire nos opérations.

On vide le set ensemble par précaution (les aléas de la mémoire), avec FD_ZERO. Et on ajoute notre socket d'écoute au set, avec FD_SET.

5. **Vérification de l'arrivée de client**

   On rentre dans la boucle générale.

   Dans cette boucle, on commence par copier le set *ensemble* dans le set temporaire *temp*. Ensuite, on attends avec la primitive *select()* la réception d'un signal indiquant qu'un client veut se connecter. *On mets max+1 (max contenant le numero de descripteurs de secoute), la socket temporaire en lecture et null pour le reste, on veut une attente illimitée donc null également pour la structure timeval.*

Quand un client se connecte, on ajoute son descripteurs au set *ensemble*. (et on adapte si nécessaire le max).

6. **Gestion des clients**

   Et on gère les clients.

   On crée une boucle qui va parcourir tout les descripteurs de notre set.

   Dans cette boucle, on commence par vérifier que l'on ne parcours pas un descripteurs qui n'est pas dans le set(et inactif) ou la socket d'écoute. Dans ce cas, on saute.

   Sinon, on lit le message avec la primitive *read()*. Dans les vérifications, si il y a une erreur ou si la liaison est terminée, on retire de l'ensemble le client (avec FD_CLR) puis on coupe la communication et on ferme la socket.

   Une fois que le message est lu, on l'affiche sur la sortie standard et on le communique à tous les clients (sauf le client qui a envoyé).

7. **Terminaison du serveur**

   On coupe la communication et on ferme la socket.

Le client

Ce client est un peu particulier, il doit à la fois pouvoir émettre à tout moment un message et à la fois recevoir à tout moment un message.

Pour pouvoir prendre en compte cette contrainte, on va crée un processus père et un processus fils.

Le processus fils : le processus fils va s'occuper de la reception des messages que le serveur envoie et va les afficher sur la sortie standard.

Le processus père : le processus père va gérer les messages saisit par le client et les enverra au serveur.

```c
int main(int argc, char *argv[]){

    int nbLus;
    char message[BUFFER_SIZE];

    char pseudo[strlen(argv[2])+1];
    strcpy(pseudo, argv[2]);

    char messageToSend[BUFFER_SIZE];


    /* 1. Création de la socket */

    int sclient = socket(AF_INET, SOCK_STREAM, 0);
    if(sclient == -1){
        perror("Erreur de création de la socket");
        return 1;
```

```
    }

    /* 2. Création de la structure d'adresse */

    struct sockaddr_in saddr = {0};
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(PORT_SERVEUR);
    saddr.sin_addr.s_addr = inet_addr(argv[1]);

    /* 3. Connexion au serveur */

    while(connect(sclient, (struct sockaddr *) &saddr, sizeof(saddr)) ==
-1);

    /* Fonctionnalités */
    /* Le fils lit les messages envoyés par le serveur */
    if(fork() == 0){
        while(1){
            int nbLus = read(sclient, message, BUFFER_SIZE-1);
            if (nbLus == -1){
                perror("Erreur lors du read depuis le serveur");
            }
            if (nbLus == 0){
                break;
            }

            //écriture sur la sortie
            write(1,"\n", strlen("\n"));
            write(1,message, nbLus);
            write(1,"> ", strlen("> "));
        }
        shutdown(sclient, SHUT_RDWR);
        close(sclient);
        exit(0);
    }

    /* Le père envoie les messages au serveur */
    while(1){
        write(1, "> ", strlen("> "));
        nbLus = read(0, message, BUFFER_SIZE);

        if(nbLus == -1){
            perror("Erreur lors du read");
            return 1;
        }

        if(nbLus == 0){
            break;
        }

        message[nbLus] = '\0';
        sprintf(messageToSend, "%s : %s", pseudo, message);

        write(sclient, messageToSend, strlen(messageToSend));
```

```
    }
    /* 4. Terminaison */
    close(sclient);
    shutdown(sclient, SHUT_RDWR);
    return 0;
}
```

On commence par les trois étapes classiques du client :

1. **Création de la socket**
2. **Création de la structure d'adresse**

   L'adresse corresponds à l'adresse sur laquelle on va joindre le serveur.

3. **Connexion au serveur**

   On essaie de se connecter tant que la connexion n'est pas établie.

4. **Fonctionnalités**

   Le fils lit les messages envoyés par le serveur

   Très classique, on attend l'envoi avec la primtive **read()** et quand on a reçu, on affiche sur la sortie standard (avec une mise en forme).

   Le père envoie les messages au serveur

   Le père affiche le symbole "> " indiquant que l'utilisateur peut écrire. Lorsque l'utilisateur appuie sur "entrée" (signifiant la fin de la chaine), le père récupère le message avec la primitive *read()*.

   Il mets en forme le message, en affichant le pseudo avant le message. **Sans oublier d'ajouter le caractère de fin de chaine**.

   Une fois le message mis en forme, il l'envoie au serveur.

5. **Terminaison**

On coupe la communication et on ferme la socket.