Client/Serveur
Les sockets et le multiplexage E/S
CNAM 2006

Sylvain Pasquet

Sommaire

Première partie

Le client socket UNIX POSIX.

1 Description de la méthode de traitement.

Notre client fonctionnera en TCP (mode connecté), il devra afficher les messages envoyés par le serveur et envoyer les messages saisies par l'utilisateur dans la console. On devra utiliser un multiplexage d'entrées pour lire soit les messages du serveur ou soit l'entrée standard du programme (saisie clavier). Afin de procéder, nous allons utiliser la commande C POSIX Select(), qui permet de déclarer un ensemble de descripteurs de fichier (entrée clavier, et socket reseau) à surveiller afin de procéder à la méthode adéquate pour chacun de ses événements.

1-Exécution du client avec IP et port.
2-Connexion au serveur avec IP et port.
3-Mise en attente infinie pour les évenements.
3-Déclaration des entrées à surveiller.
4-Traitement de l'information en fonction de l'évenement sur une entrée.
5-Fin du programme.

Tab. 1 – Détails du traitement de l'information.

2 Exécution du client avec IP et port.

Le lancement du programme client devra être simple, voici un exemple si le serveur est exécuté sur la machine 10.0.0.2 et sur le port 5632 :

```
$./client 10.0.0.2 5632
```

3 Connexion au serveur avec IP et port.

Le programme devra récuperer les arguments passés à l'exécution afin de connaitre sur quel serveur et sur quel port il devra se connecter. Tout d'abord nous vérifions si le nombre d'argument est correct avant de continuer l'éxécution du programme, la cas contraire le programme s'arrête en expliquant le fonctionnement de celui-ci. Si le nombre d'argument est correct nous les utiliserons directement pour remplir la struture **sockaddr_in** initialisée dans `their_addr`, on devra l'utiliser pour la création du socket de connexion vers le serveur.

```
/*On compte le nombre d'argument*/
if( argc < 2 )
{
        fprintf( stderr, "usage: ./client IP PORT\n" );
        exit( 1 );
}
/*On rempli la structure hostent (he) en utilisant*/
/*le premier argument qui est l'IP du serveur*/
if( ( he = gethostbyname( argv[1] ) ) == NULL )
{
        herror( "gethostbyname" );
        exit( 1 );
}

/*Initialisation de notre socket pour la futur connexion*/
if( ( sockfd = socket( AF_INET, SOCK_STREAM, 0 ) ) == -1 )
{
        perror( "socket" );
        exit( 1 );
}

/*their_addr est la structure de type sockaddr_in*/
/*On la rempli avec les informations comme */
/*le type de socket, le port et l'ip*/

/* host byte order */
their_addr.sin_family = AF_INET;
/* short, network byte order */
their_addr.sin_port = htons(atoi(argv[2]));/
their_addr.sin_addr = *( ( struct in_addr * )he->h_addr );
```

On se connecte sur le serveur grâce au socket créé, en utilisant les informations contenus dans la structure `their_addr`, on sort du programme en cas d'erreur de la création du socket.

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

4 Mise en attente infinie pour les évenements.

Pour réaliser cette attente infinie de notre programme on va utiliser une boucle **for** sans condition. Cela va permettre d'attendre indéfiniment les futurs évenements clavier ou messages du serveur.

```
for( ;; )
        {
/*Contiendra:*/
/*-Les entrees a surveiller.*/
/*-Le code de gestion des evenements.*/
        }
```

5 Déclaration des entrées à surveiller dans notre boucle for.

Nous allons utiliser le multiplexage de la commande C POSIX **select()**, afin de pouvoir surveiller plusieurs entrées d'informations et eviter les situations de blocage. Cet ensemble d'entrées à surveiller sera declaré dans la variable `ensemble_lecture` qui sera initialisée avec la macro C **fd_set**.

```
fd_set ensemble_lecture;
```

Ensuite, on remet à zero ensemble_lecture par sécurité, et on ajoute nos deux descripteurs de fichiers à surveiller, le clavier tout d'abord avec `STDIN_FILENO` et ensuite notre socket avec son descripteur `sockfd`.Et enfin on initialise le **select()** avec ensemble_lecture, le premier argument est équivalent au plus grand descripteur de fichier +1, ce que contient `FD_SETSIZE`. Si il y a une erreur pendant la création du **select()** alors on quitte le programme.

```
FD_ZERO( &ensemble_lecture );
FD_SET( STDIN_FILENO, &ensemble_lecture );
FD_SET( sockfd, &ensemble_lecture );
if( select( FD_SETSIZE, &ensemble_lecture, NULL, NULL, NULL ) < 0 )
        {
                perror( "Select" );
        }
```

Nous allons définir maintenant ce que le programme doit faire si un événement arrive sur le descripteur de fichier `STD_FILENO`. La macro `FD_ISSET` nous permet de tester si le descripteur de fichier contient un évenement, si c'est le cas on rentre dans notre condition de traitement qui envoie le message saisis sur la socket. En cas d'erreur d'envoi du message vers le serveur on quitte le programme.

```
/*On teste notre descripteur de fichier*/
if( FD_ISSET( STDIN_FILENO, &ensemble_lecture ) )
        {
                nb_lus = read( STDIN_FILENO, buf, sizeof buf );
                /*On envoi le message saisie sur le socket*/
                /* vers le serveur*/
                if( ( numbytes = sendto( sockfd, buf, nb_lus, 0,
                                ( struct sockaddr * )&their_addr,
                                 sizeof( struct sockaddr ) ) ) == -1 )
                {
                        perror( "sendto" );
                        exit( 1 );
                }
        }
```

Nous allons définir maintenant ce que le programme doit faire si un événement arrive sur le descripteur de fichier `sockfd` qui correspond au socket de connexion sur le serveur. La macro `FD_ISSET` nous permet de tester si le descripteur de fichier contient un événement, si c'est le cas on rentre dans notre condition de traitement qui envoie le message du socket vers la sortie standard de la console. En cas d'erreur de réception du message on quitte le programme.

```
/*On teste notre descripteur de fichier*/
if( FD_ISSET( sockfd, &ensemble_lecture ) )
        {
                /*Procedure le lecture du socket*/
                if( ( numbytes = recv( sockfd, buf,
                                        MAXDATASIZE, 0 ) ) == -1 )
                {
                        perror( "recv" );
                        exit( 1 );
                }
                buf[numbytes] = '\0';
                printf( "Recu: %s \n", buf );
        }
```

Au final, si on met l'ensemble du code de traitement dans la boucle for, cela nous donne :

```
for( ;; )
{
        FD_ZERO( &ensemble_lecture );
        FD_SET( STDIN_FILENO, &ensemble_lecture );
        FD_SET( sockfd, &ensemble_lecture );

        if( select( FD_SETSIZE, &ensemble_lecture, NULL, NULL, NULL ) < 0 )
        {
                perror( "Select" );
        }
        if( FD_ISSET( STDIN_FILENO, &ensemble_lecture ) )
        {
                nb_lus = read( STDIN_FILENO, buf, sizeof buf );
                if( ( numbytes = sendto( sockfd, buf, nb_lus, 0,
                                        ( struct sockaddr * )&their_addr,
                                        sizeof( struct sockaddr ) ) ) == -1 )
                {
                        perror( "sendto" );
                        exit( 1 );
                }
        }

        if( FD_ISSET( sockfd, &ensemble_lecture ) )
        {
                /*Procedure le lecture du socket*/
                if( ( numbytes = recv( sockfd, buf,
                                        MAXDATASIZE, 0 ) ) == -1 )
                {
                        perror( "recv" );
                        exit( 1 );
                }
                buf[numbytes] = '\0';
                printf( "Recu: %s \n", buf );
        }
}
```

Deuxième partie

Le serveur socket UNIX POSIX.

6 Description de la méthode de traitement.

1-Exécution du serveur avec Nom et port.
2-Création du socket d'écoute sur le port.
4-Mise en attente infinie pour les événements.
5-Déclaration des entrées à surveiller.
6-Traitement de l'information en fonction de l'événement sur une entrée.
7-Fin du programme.

Tab. 2 – Détails du traitement de l'information.

Notre serveur fonctionnera en TCP (mode connecté), il devra afficher les messages envoyés par les clients et réenvoyer le message recu à tous les clients connectés au serveur. On devra utiliser un multiplexage d'entrées/sorties pour lire soit les messages des clients, soit réécrire le message envoyé d'un client sur tous les clients connectés par l'intermédiaire de la socket.

7 Exécution du serveur avec Nom et port.

Le lancement du programme serveur devra être simple, voici un exemple pour un serveur qui s'appellera serveur1 sur le port 5632 :

```
$./serveur serveur1 5632
```

8 Création du socket d'écoute sur le port.

Pour stocker les informations de création de la socket du serveur on déclare notre structure **sockaddr_in** que l'on nommera adr, et pour les clients on utilisera aussi une structure **sockaddr_in** que l'on appellera `client`. On utilisera la structure `client` à chaque connexion d'un client sur le serveur.

```
struct sockaddr_in adr;
struct sockaddr_in client;
```

Nous vérifions au lancement du programme que les arguments sont corrects c'est à dire que le nombre d'arguments convient et que le port fourni est compris entre 1025 et 65535. Si n'est pas le cas on sort du programme en affichant la syntaxe attendue.

```
if( (argc != 3) || ( atoi(argv[2])<1025 ) || ( atoi(argv[2])>65535 ) )
        { printf("Usage : ./serveur <Nom_du_serveur> <numero_port>\n
                        Numero de port compris entre 1025 et 65535\n");
        exit(-1);
        }
/*On stock le nom du serveur dans la variable servername*/
servername=argv[1];

/*On cree notre message de bienvenue en concatenant */
/*"Connexion etablie sur:" et le nom du serveur*/
/*Ce message sera envoye a chaque client connecte*/
strcat(bienvenue, (char*)servername);

/*On stock le numero de port dans numport*/
numport = atoi(argv[2]);
```

Nous pouvons maintenant initialiser notre futur socket TCP, éventuellement en cas d'erreur nous quittons le programme. Pour mettre en relation une socket et un port nous utilisons la commande **bind()** mais avant de procéder, nous devons initialiser la structure `adr` avec les informations du serveur comme le numéro de port et l'interface ethernet à utiliser ou même la famille.

```
/*Initialisation de la famille dans la structure adr*/
adr.sin_family=AF_INET;

/*Initialisation de notre futur socket TCP*/
if ((mySocket=socket(adr.sin_family,SOCK_STREAM,IPPROTO_TCP))==-1)
{
        perror("Erreur socket");
        exit(1);
}

/*Pour prendre l'adresse IP du systeme, htonl pour une portabilite maximum*/
adr.sin_addr.s_addr=htonl(INADDR_ANY);

/*On convertit la valeur PORT en Host to Network Short*/
/*Si 0 alors port->aleatoire*/
adr.sin_port=htons(numport);

adr_len=sizeof(adr);

/*On associe les informations de adr au socket*/
/*Afin de creer celui-ci*/
/*En cas d'erreur on sort du programme*/
if (bind(mySocket,(struct sockaddr *)&adr,adr_len))
```

```
{
        perror("Erreur : Bind impossible");
        exit(2);
}
```

Ensuite nous pouvons mettre la socket en écoute, nous limiterons le nombre de connexions en attente à 10. En cas de problème lors de la mise en écoute du socket on sort du programme.

```
/*On limite le nombre de connexions en attente a 10*/
if(listen(mySocket,10))
{
        perror("Erreur de listen");
        close(mySocket);
        exit(23);

}
```

9 Mise en attente infinie pour les événements.

Nous allons pouvoir manager les événements arrivant sur la socket du serveur. Comme pour le client nous allons utiliser la fonction **select()**, non pas pour 2 entrées mais pour l'ensemble des clients connectés. Une autre méthode de boucle infinie est utilisée, dans le client nous avions utilisé un **for()** sans condition, ici nous allons utiliser un while avec une condition 1 (TRUE), qui met aussi le processus en boucle.

Pour notre serveur nous allons utiliser deux ensembles de lecture de descripteurs de fichier, le premier ensemble représente tous les clients connectés, à chaque connexion d'un client nous ajouterons son descripteur de socket dans cet ensemble. Un deuxieme ensemble tampon `rfds` qui est une copie à chaque tour du **while** de `ensemble_lecture` servira à tester si un client a un message à transmettre, on utilise ce tampon pour éviter la déconnexion directe d'un nouveau client sur le serveur, le nouveau client sera pris en compte au prochain tour de la boucle **while**. Pour parcourir les clients afin de tester s'ils ont un message à transmettre ou pour envoyer le message recu à tous, nous utilisons une boucle **for** qui parcourt tous les descripteurs de fichier déclarés dans la tampon `rfds`.

```
/*Descripteur pour le select*/
fd_set rfds, ensemble_lecture;
```

Par mesure de sécurité nous remettons à zéro les structures `ensemble_lecture` et `rfds` avant notre boucle infinie.

```
FD_ZERO(&ensemble_lecture);
FD_ZERO(&rfds);
FD_SET(mySocket, &ensemble_lecture);
```

Voici notre boucle infinie oú se trouvera le code de traitement des informations.

```
 while( 1 )
{
/*Notre code de gestion se trouvera ici*/
}
```

Maintenant il nous faut tester si des clients sont connectés sur le serveur, si c'est le cas on affichera l'adresse IP du client et on l'ajoutera dans notre ensemble de descripteur à vérifier avec **select()**.

10 Déclaration des entrées à surveiller.

```
/* copie des listes de sockets */
/* de ensemble_lecture vers rfds*/
memcpy(&rfds, &ensemble_lecture, sizeof(rfds));

/*Initialisation du select sur l'ensemble des descripteurs +1*/
/* surveillance des descripteurs en lecture */
if( select(FD_SETSIZE, &rfds, 0, 0, 0) == -1 )
{
        fprintf(stderr, "select: %s", strerror(err));
        exit(1);
}

/*On teste mySocket pour une */
/*eventuelle connexion d'un client*/
if( FD_ISSET(mySocket, &rfds) )
{
        adr_len_cli = sizeof(client);
        if( (ClientSocket = accept(mySocket,
                        (struct sockaddr *)&client,
                        &adr_len_cli)) == -1 )
        {
                fprintf(stderr, "accept: connexion impossible\n");
                exit(1);
        }
        /*J'imprime l'adresse IP du client*/
        printf(" Connexion d'un client depuis %s \n",
                        inet_ntoa(client.sin_addr));
        fflush(stdout);

        /* ajout du client dans les sockets a surveiller */
        FD_SET(ClientSocket, &ensemble_lecture);
```

```
        /* souhaiter la bienvenue au client */
        /* Envoi du message sur le socket   */
        write(ClientSocket, bienvenue, strlen(bienvenue));

        /* lecture non bloquante */
        fcntl(ClientSocket, F_SETFL, O_NONBLOCK |
                        fcntl(ClientSocket, F_GETFL, 0));
}
```

11 Traitement de l'information en fonction de l'évenement sur une entrée.

On parcourt maintenant l'ensemble **rfds** pour tester si des clients ont des messages à transmettre au serveur.

```
/* Tester si les sockets clientes ont ete modifiees */
for( fd=0; fd<FD_SETSIZE; fd++ )
{
/*On teste si le descripteur en cours existe*/
/*On saute le socket du serveur             */
        if( fd != mySocket && FD_ISSET(fd, &rfds) )
        {
        /* Verification de la connexion d'un client */
        /* Lire client est une fonction de lecture sur le socket*/
                if( (message = lire_client(fd)) == NULL )
                {
                /*On ferme la socket*/
                /*Et on affiche un message sur le serveur*/
                        close(fd);
                        FD_CLR(fd, &ensemble_lecture);
                        fprintf(stderr, "--␣perte␣d'un␣client␣!␣--\n");
                }else{
                /* envoyer le message recu a tous les clients */
                        for( c=0; c<FD_SETSIZE; c++)
                        {
                        if( c!=mySocket && FD_ISSET(c, &ensemble_lecture) )
                        {
                                write(c, message, strlen(message)+1);
                        }
                        }
                }
        }
}
```

12 Fonction annexe de lecture du socket.

Afin de clarifier le code de traitement de l'information du serveur, la fonction de lecture d'une socket a été mise à part.

```
char *lire_client(int sock)
{
        /*Allocation memoire: taille maximum de 512 -> MAXBUFFER*/
        char *msg = (char *)malloc(MAX_BUFFER * sizeof(char));
        int nb_lu;

        /*Lecture du socket*/
        nb_lu = read(sock, msg, MAX_BUFFER);
        /*Test de message non vide*/
        if( nb_lu > 0 )
        {
        /*Affichage sur la console du message lu*/
                printf("Message: %s\n", msg);
                fflush(stdout);
        /*On retourne le message*/
                return msg;
        }else{
        /*Si aucun message on retourne NULL*/
                return NULL;
        }
return NULL;
}
```