T.P. Sockets

Programmation Sockets UDP/TCP

S. BOUDJIT

1 Objectif du TP

Développer une communication rudimentaire entre deux processus, en adoptant une architecture de type client/serveur. La communication entre client et serveur étant basé sur l'interface de communication socket.

2 Rappel de l'architecture Client/Serveur

Dans l'architecture de type client/serveur les applications peuvent être classées en deux catégories :

- Les **Serveurs** : applications qui attendent la communication, c'est à dire l'attente d'une demande d'ouverture de connexion, la réception d'une requête et l'envoie d'une réponse.
- Les **Clients** : applications qui prennent l'initiative du lancement de la communication, c'est à dire demande de connexion, l'envoie d'une requête, l'attente de la réponse, reprise de l'exécution du programme.

Une autre caractéristique des applications est leurs mode de connexion. Deux modes de connections peuvent être utilisés (Mode connecté ou STREAM , Mode Non-connecté ou DATAGRAM) .

Le protocole TCP/IP est un protocole orientée connexion et le protocole UDP/IP est un protocole orientée sans connexion. L'avantage de l'utilisation d'un protocole comme TCP/IP est la fiabilité : la couche transport (TCP) effectue elle même le cheksum , la réémission de morceau de message perdus, l'élimination des morceau dupliqués, l'adaptation du débit. Un protocole comme UDP/IP n'effectue pas ces vérifications qui doivent alors être faites par le protocole de communication de niveau applicatif. Pour cette raison, la programmation de clients ou serveurs en mode non connecté est plus complexe qu'en mode connecté. Cependant, en réseau local où le transport est fiable, il peut être avantageux d'utiliser UDP car ce mode de communication demande moins d'opérations que TCP.

3 Travail à faire :

3.1 Partie 1 : communication type DATAGRAM dans le domaine INTERNET

La communication se fera par une socket attachée à un port prédéfini. La socket sera de type DATAGRAM et son domaine sera INTERNET (AF_INET). Le serveur de base recevra une requête en lisant sur la socket et la traitera. La lecture sur une socket étant par défaut bloquante, cette propriété est utilisée pour mettre le serveur en attente de réception de requêtes. Le Client se connecte au serveur et transmet une chaîne de caractère. La structure du client et du serveur est donnée dans la figure suivante :

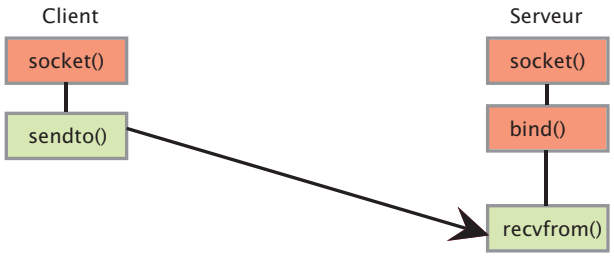

Fig. 1 – Mode non connecté

1. *Pour le serveur* : On vous fournit un squelette de code du serveur que vous devez recopiez dans un fichier, puis renommer en serveur_udp.c, et compléter le à partir des éléments du cours. Puisqu'il s'agit d'un serveur basic, il recevra une chaîne du client et il l'affichera sur la sortie standard. On lancera le serveur en background avec comme paramètre le port d'écoute , en l'occurrence 9600 .
2. *Pour le client* : On vous fournit un squelette de code du client que vous devez recopiez dans un fichier, puis renommer en client_udp.c, et enfin compléter à partir des éléments du cours. Puisqu'il s'agit d'un client basic, on lui demandera de lire tout simplement sur l'entrée standard ¡stdin¿ du clavier et d'envoyer au serveur ce qui a été tapé.

On lancera le client en foreground, afin d'avoir le contrôle du clavier pour envoyer les messages. Les 2 arguments sur la ligne de commande du client seront d'une part le nom du host où se trouve le serveur et d'autre part le numéro du port sur lequel ce dernier écoute, en l'occurrence 9600 .

Squelette du Client

```
/*
*  Ce client attend des entrees sur stdin et les
*  envoie au serveur auquel il est connecte.
*/

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

#define TAILLE 100
#define PORT 9600

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

/**************** variables du client ************/

/*
* Definir ici la socket du client
*/

/*
* Definir ici la structure de la socket du client
```

```
*/

/*
* Definir ici la structure d'une socket
* d'information sur le serveur
*/

/*
* Definir ici le pointeur de structure hostent pour
* resoudre le nom du serveur en adresse IP
*/

/*
* Definir ici le buffer contenant la chaîne de
* carectere a faire parvenir au serveur
*/

/*
* Definir ici la chaine contenant le nom de
* la machine du serveur
*/

/***************** code du client ****************/

/*
 * * Entrer les messages a transmettre via le
 * clavier (stdin)
 * */

/*
 * * Entrer le nom de la machine du serveur
 * */

/*
* Ouvrir ici le point de communication du client
* qui rendra le file-destripteur de la socket client
*/

/*
* Garnir ici la struture de la socket client
* qui comprend les champs:
*  - famille INET
*  - adresse IP de la machine ou se trouve
*    le serveur (recuperee par gethostbyname(),
*    nom de la machine ou se trouve le serveur)
*  - numero de port du serveur
*    (definie au debut du code)
*/

/*
transmettre la chane qu serveur
*/

/*
* Fermer ici la communication en sortie
*/

/*
* Terminer ici la vie du client
*/
 exit(0);
}
```

Squelette du Serveur

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

/* port d'attache du serveur */
#define PORT 9600

int main(int argc, char *argv[])
{
/********** variables du serveur **********/

/*
* Definir ici la variable socket du serveur
*/

/*
* Definir ici la structure d'une socket
* d'information sur le serveur
*/

/*
* Definir ici la structure d'une socket
* d'information sur le client
*/

/*
* Definir ici la taille de la socket du client
*/

/********** code du serveur **********/
/*
* Ouvrir ici le point de communication de
* la socket du serveur
*/

/*
* tester si la socket a ete bien creer
* */

/*
* Garnir ici la structure de la socket
* du serveur avec:
*      - sa famille
*      - sur (la/les)quelle(s) de ses
*        adresses elle ecoute
*      - sur quel port elle ecoute
*/

/*
* Attacher ici le file-descripteur de la
* socket a sa structure
*/
```

```
/*
* Boucle generale du serveur (infinie)
* Le serveur ne doit jamais s'arreter en
* principe (pas d'exit()). Le serveur
* doit lire a partir de la socket et
* afficher sur Ècran le message reçu
* de la part du client.
*/

do {
   }while(1);


exit(0);
}
```

3.2 Partie 2 : communication type STREAM dans le domaine INTERNET

La communication se fera par une socket attachée a un port prédéfini. La socket sera de type STREAM et son domaine sera INTERNET (AF_INET). Le serveur se met à l'écoute sur sa socket, dés la réception d'une demande de connexion il initialise une nouvelle socket et crée un fils pour traiter la requête. Le fils va traiter la requête en utilisant la nouvelle socket. Le serveur se remet à l'écoute sur sa socket initiale. La structure du client et du serveur est donnée dans la figure suivante :

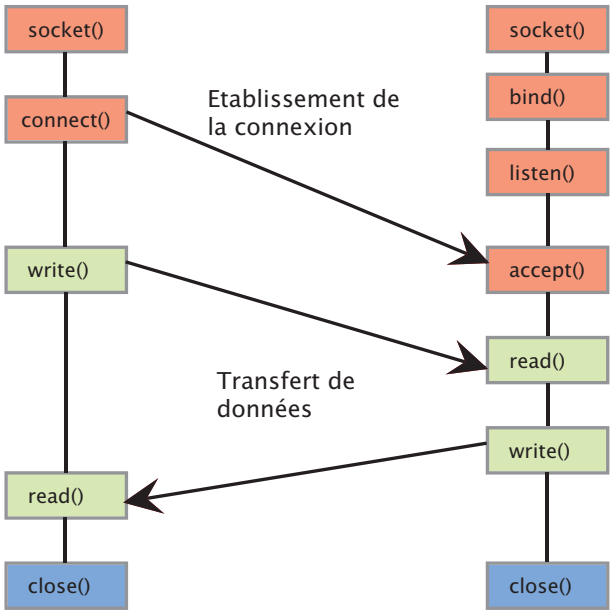

FIG. 2 – Mode connecté

On vous demande de suivre le schéma donné dans la figure précédente et faire le code du serveur ainsi que le code du client.

4 Annexe

4.1 les familles d'adresse

Il existe plusieurs familles d'adresses, chacune correspondant à un protocole particulier. Les familles les plus répandues sont :
- `AF_UNIX` Protocoles internes de UNIX
- `AF_INET` Protocoles Internet
- `AF_NS` Protocols de Xerox NS
- `AF_IMPLINK` Famille spéciale pour des applications particulières auxquelles nous ne nous intéresserons pas.

4.2 les structures d'adresse

La structure d'une adresse de socket sous TCP/IP est la suivante :

```
struct in_addr {
u_long s_addr ;
} ;

struct sockaddr_in {
u_short sin_family;/* famille d'adresses */
u_short sin_port; /* numéro de port */
struct in_addr sin_addr; /* adresse IP */
char sin_zero[8]; /* inutilisé */
};
```

- **sin_family** : AF_INET ;
- **sin_port** : 16 bits de numéro de port (ordonnancement réseau) ;
- **sin_addr** : 32 bits constituants l'identificateur du réseau et de la machine hôte ordonnes selon l'ordre réseau ;
- **sin_zero[8]** : inutilisés ;

4.3 L'appel système **socket()**

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


int socket (int family, int type, int protocole) ;
```

La variable **family** peut prendre 4 valeurs dont les préfixes commencent par AF comme Famille d'Adresse :
- `AF_UNIX` Protocoles internes de UNIX
- `AF_INET` Protocoles Internet
- `AF_NS` Protocols de Xerox NS
- `AF_IMPLINK` Famille spéciale pour des applications particulières auxquelles nous ne nous intéresserons pas.

La variable **type** peut prendre 5 valeurs :

- `SOCK_STREAM` utilisé en mode connecté au dessus de TCP.
- `SOCK_DGRAM` utilisé en mode déconnecté avec des datagrammes au dessus de UDP.

L'argument **protocole** dans l'appel système socket est généralement mis à 0. Il est utilisé dans certaine applications specifiques. L'appel système socket retourne un entier dont la fonction est similaire à celle d'un descripteur de fichier. Nous appellerons cette entier descripteur de sockets (sockfd).

4.4 L'appel système **bind()**

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

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

Le premier argument est le descripteur de socket retourné par l'appel **socket**. Le second argument est un pointeur sur une adresse de protocole spécifique et le troisième est la taille de cette structure d'adresse. Il y a 3 utilisations possibles de **bind** :

- Le serveur enregistre sa propre adresse auprès du système. Il indique au système que tout message reçu pour cette adresse doit lui être fourni. Que la liaison soit avec ou sans connexion l'appel de bind est nécessaire avant l'acceptation d'une requête d'un client.
- Un client peut enregistrer une adresse spécifique pour lui même.
- Un client sans connexion doit s'assurer que le système lui a affecté une unique adresse que ses correspondants utiliseront afin de lui envoyer des messages.

L'appel de **bind** rempli l'adresse locale et celle du process associés au socket.

4.5 L'appel système **connect()**

Un socket est initialement créé dans l'état non connecté, ce qui signifie qu'il n'est associé à aucune destination éloignée. L'appel système connect associe de façon permanente un socket à une destination éloignée et le place dans l'état connecté.

Un programme d'application doit invoquer connect pour établir une connexion avant de pouvoir transférer les données via un socket de transfert fiable en mode connecté.

Les sockets utilisées avec les services de transfert en mode datagramme n'ont pas besoin d'établir une connexion avant d'être utilisés, mais procéder de la sorte interdit de transférer des données sans mentionner à chaque fois, l'adresse de destination.

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

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

- **sockfd** est le descripteur de socket retourné par l'appel socket.
- **servaddr** est un pointeur sur une structure d'adresse de socket qui indique l'adresse de destination avec laquelle le socket doit se connecter.
- **addrlen** taille de la structure d'adresse.

4.6 L'appel système **listen()**

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

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

Cet appel est généralement utilisé après les appels socket et bind et juste avant les appels l'appel accept. L'argument backlog spécifie le nombre de connections à établir dans une file d'attente par le système lorsque le serveur exécute l'appel accept. Cet argument est généralement mis à 5 qui est la valeur maximale utilisée.

4.7 L'appel système **accept()**

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

int accept (int sockfd,
            struct sockaddr *peer,
            int *addrlen) ;
```

L'argument **sockfd** désigne le descripteur de socket sur lequel seront attendues les connexions. Peer est un pointeur vers une structure d'adresse de socket.

Lorsqu'une requête arrive, le système enregistre l'adresse du client dans la **stucture *peer** et la longueur de l'adresse dans ***addrlen**. Il crée alors un nouveau socket connecté avec la destination spécifiée par le client, et renvoit à l'appelant un descripteur de socket. Les échanges futurs avec ce client se feront donc par l'intermédiaire de ce socket.

Le socket initiale **sockfd** n'a donc pas de destination et reste ouvert pour accepter de futures demandes.

Tant qu'il n'y a pas de connexions le serveur se bloque sur cette appel. Lorsqu'une demande de connexion arrive, l'appel système **accept** se termine. Le serveur peut gérer les demandes itérativement ou simultanément :

1. Dans l'approche itérative, le serveur traite lui même la requête, ferme le nouveau socket puis invoque de nouveau **accept** pour obtenir la demande suivante.
2. Dans l'approche parallèle, lorsque l'appel système **accept** se termine le serveur crée un serveur fils chargé de traiter la demande (appel de **fork** et **exec**). Lorsque le fils a terminer il ferme le socket et meurt. Le serveur maître ferme quand à lui la copie du nouveau socket après avoir exécuté le **fork**. Il appel ensuite de nouveau **accept** pour obtenir la demande suivante.

4.8 Obtenir des informations sur une machine par son nom

La fonction gethostbyname (rechercher une machine par son nom) accept un nom de domaine et renvoie un pointeur vers une structure qui contient l'information relative à cette machine. L'appel est le suivant :

```
struct hostent *  gethostbyname (char *chaîne_nom) ;
```

la structure où gethostbyname met les information dans la structure suivante :

```
struct hostent {
char *h_name ;
     /* nom officiel de la machine*/
char *h_aliases ;
     /* liste d'alias */
int h_addrtype ;
    /* type d'adresse (exemple adresse IP) */
```

```
int h_length ;
    /* longueur de l'adresse */
char **h_addr_list ;
    /* liste des adresses */
}
#define h_addr h_addr_list[0]
/* pour compatibilité */
```

4.9 Emission d'information

Une fois que le programme d'application dispose d'un socket, il peut l'utiliser afin de transférer des données. Cinq appels système sont utilisables : **send**, **sendto**, **sendmsg**, **write** et **writev**. **send**, **write** et **writev** ne sont utilisables qu'avec des sockets en mode connecté car ils ne permettent pas d'indiquer d'adresse de destination. Les différences entre ces trois appels sont mineures :

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

write (int sockfd, char *buff,
       int nbytes ) ;
writev (int sockfd, iovec *vect_E/S,
        int lgr_vect_E/S ) ;
int send (int sockfd, char *buff,
          int nbytes, int flags ) ;
```

- **socket** contient le descripteur de socket.
- **buff** est un pointeur sur un tampon où sont stockées les données à envoyer.
- **nbytes** est le nombre d'octets ou de caractères que l'on désire envoyer.
- **vect_E/S** est un pointeur vers un tableau de pointeurs sur des blocs qui constituent le message à envoyer.
- **flags** : drapeau de contrôle de la transmission, par défaut mettre à 0.

Pour le mode non connecté on a deux appels **sendto** et **sendmsg** qui imposent d'indiquer l'adresse de destination :

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

int sendto (int sockfd, char *buff,
            int nbytes, int flags,
            struct sockaddr *to, int addrlen) ;
```

- **to** est la structure de l'adresse de la socket destinataire.
- **addrlen** est la taille de la structure d'adresse **to**.

Les quatres premiers arguments sont les mêmes que pour **send**, les deux derniers sont l'adresse de destination et la taille de cette adresse. Pour les cas où on utiliserait fréquemment l'appel **sendto** qui nécessite beaucoup d'arguments et qui serait donc d'une utilisation trop lourde on définit la structure suivante :

```
struct struct_mesg {
int *sockaddr ;
int sockaddr_len ;
iovec *vecteur_E/S ;
int vecteur_E/S_len ;
int *droit_d'acces ;
int droit_d'acces_len ;
}
```

Cette structure sera utilisée par l'appel sendmsg :

```
int sendmsg (int sockfd, struct struct_mesg,
             int flags ) ;
```

4.10 reception d'information

On distingue 5 appels système de réception d'information qui sont symétriques au appels d'envoi. Pour le mode connecté on a les appelles **read**, **readv** et **recv** et pour le mode sans connexion on a les appelles **recvfrom** et **recvmsg**.

```
int read (int sockfd, char *buff, int nbytes ) ;
int readv (int sockfd, iovec *vect_E/S,
           int lgr_vect_E/S ) ;
int recv (int sockfd, char *buff,
          int nbytes, int flags ) ;
```

- **sockfd** est le descripteur sur lequel les données seront lues.
- **buff** est un pointeur sur un buffer où seront stockées les données lues.
- **nbytes** est le nombre maximal d'octets ou de caractères qui seront lus.
- **readv** permet de mettre les données lues dans des cases mémoire non contigues. Ces cases mémoires sont pointées par un tableau de pointeurs qui lui même est pointé par **vect_E/S**.
- **lgr_vect_E/S** est la longueur de ce tableau.

Pour le mode sans connexion, il faut préciser les adresses des correspondants desquels on attend des données.

```
int recvfrom (int sockfd, char *buff, int nbytes,
              int flags, struct sockaddr *from,
              int *addrlen) ;
```

- **from** est la structure de l'adresse de la socket émettrice (**recvfrom** : permet de récupéré cette structure). - **addrlen** est la taille de la structure d'adresse **from** (**recvfrom** : permet de récupéré la taille de la structure).

Pour les mêmes raisons que pour sendto et sendmsg, et pour des raison de symétrie on a définie l'appel recvmsg qui utilise la même structure que sendmsg.

```
int recvmsg (int sockfd, struct struct_mesg,
             int flags ) ;
```