TP 2 : Programmation réseau

Objectif du TP

Développer une communication rudimentaire entre deux processus, en adoptant une architecture de type client/serveur. La communication entre le client et le serveur se fera en utilisant l'interface de communication des sockets.

Rappel de l'architecture client/serveur

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

serveurs Applications qui sont en attente d'une communication. Lorsque qu'ils reçoivent une demande de connexion, ils renvoient une réponse qui peut être positive ou négative.

clients Applications qui prennent l'initiative du lancement de la communication. Ils transmettent une requête de connexion au serveur puis ils attendent sa réponse.

La connexion peut se dérouler selon plusieurs modes : mode connecté ou non, orienté flux ou datagramme. En pratique, on dispose du protocole TCP qui offre un mode connecté orienté flux et du protocole UDP qui offre un mode non connecté orienté datagramme.

En offrant un mode orienté flux, le protocole TCP assure le découpage du flux en segments, la réémission des segments perdus, l'élimination des duplicats et l'adaptation de la taille des segments au débit. En revanche, le protocole UDP n'offre qu'un mode orienté datagramme et donc n'assure rien : c'est à l'application de mettre en œuvre les mesures nécessaires si d'autres propriétés sont requises (on peut par exemple vouloir disposer d'un mode non connecté orienté datagramme, mais avec réémission des datagrammes perdus et élimination des duplicats).

Partie 1 Communication orientée datagramme

La communication se fait par un socket attaché à un port prédéfini. La socket est de type datagramme (`SOCK_DGRAM`) et son domaine est TCP/IP (`PF_INET`). Pour traiter les requêtes des clients, le serveur lit l'information arrivée sur ce socket. La lecture sur un socket est 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 caractères. La structure de la requête est une simple chaîne de 20 caractères maximum.

Ce qu'il faut faire (Fig 1)

Pour le serveur : on vous fournit un squelette de code du serveur que vous devez recopier dans un fichier **`serveur_udp.c`**. Puis, vous devez le compléter à partir des éléments du cours et du TP. Le serveur reçoit une chaîne de caractères du client pour l'afficher sur la sortie standard. On lance le serveur avec comme argument le port d'écoute, en l'occurrence 9600.

Pour le client : on vous fournit un squelette de code du client que vous devez recopier dans un fichier, **`client_udp.c`**. Complétez ce fichier à partir des éléments du cours et du TP. Le client lit sur l'entrée standard, puis il envoie la chaîne de caractères tapée au serveur. On lance le client avec

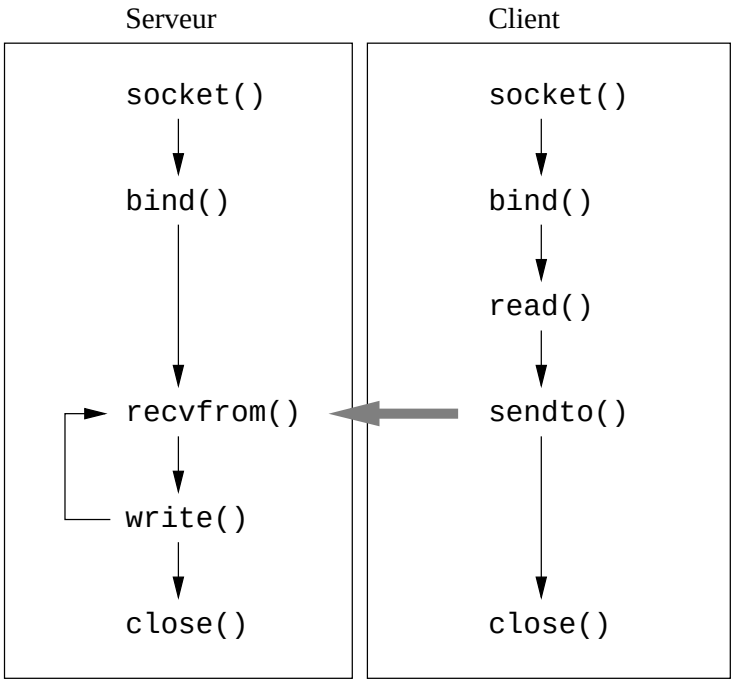

FIG. 1 – La structure du client et du serveur

comme arguments le nom de la machine où se trouve le serveur et le numéro du port sur lequel le serveur opère, en l'occurrence 9600.

Partie 2 Communication orientée flux

La communication se fait par un socket attaché à un port prédéfini. Le socket est de type flux (`SOCK_STREAM`) et son domaine est TCP/IP (`PF_INET`). Le serveur met son socket en écoute. Dès la réception d'une demande de connexion, il initialise un nouveau socket et crée un fils pour traiter la requête. Le fils traite la requête en utilisant le nouveau socket. Le serveur se remet à l'écoute sur le socket initial.

Ce qu'il faut faire (Fig 2)

Suivre le schéma donné et construire le code du serveur ainsi que celui du client.

Annexe : les structures et primitives à utiliser

1. Les familles d'adresses

Il existe plusieurs familles d'adresses, chacune correspond à un protocole particulier. Les familles les plus répandues sont :

`AF_UNIX`	Protocoles internes d'UNIX
`AF_INET`	Protocoles TCP/IP
`AF_NS`	Protocoles Xerox NS
`AF_IMPLINK`	Famille spéciale pour des applications particulières auxquelles nous ne nous intéresserons pas.

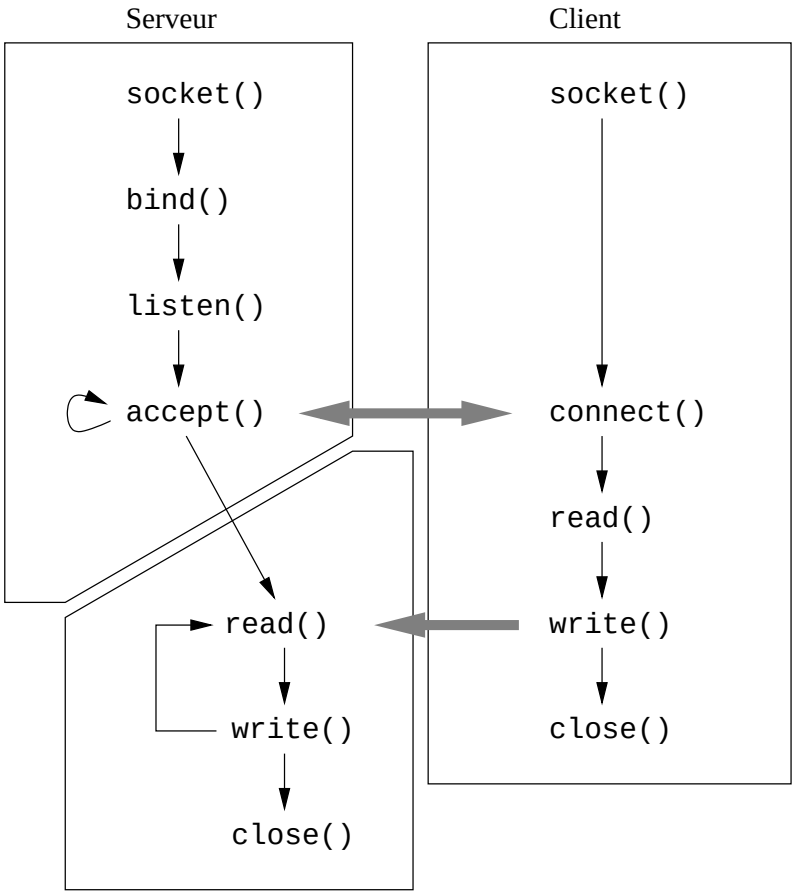

FIG. 2 – La structure du client et du serveur

2. Les structures d'adresses

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'adresse : AF_INET */
  u_short               sin_port;    /* numéro de port */
  struct in_addr        sin_addr;    /* adresse IP */
  char                  sin_zero[8]; /* inutilisé */
};
```

3. L'appel système `socket`

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

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

L'argument `domain` sert à spécifier la famille de procotoles qu'on veut utiliser. On utilise les mêmes constantes que pour les familles d'adresses, mais en utilisant le préfixe `PF_` au lieu de `AF_`, donc pour notre cas c'est `PF_INET`.

Les valeurs pour l'argument `type` qui nous intéressent ici sont `SOCK_STREAM` et `SOCK_DGRAM` qui correspondent respectivement à un socket orienté flux et orienté datagramme.

L'argument `protocol` sert à spécifier quel protocole en particulier on veut utiliser, dans les cas où il subsisterait une ambiguïté. Dans le cas de `PF_INET`, on peut passer `0`, puisque `SOCK_STREAM` et `SOCK_DGRAM` correspondent respectivement à TCP et UDP.

L'appel système renvoit un entier similaire à un descripteur de fichier.

4. L'appel système `bind`

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

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

Le premier argument est l'entier retourné par l'appel système `socket`, le deuxième est un pointeur sur une adresse spécifique au protocole et le troisième est la taille de cette adresse.

L'appel système est utilisé pour spécifier l'adresse locale (adresse IP et port dans le cas de TCP ou UDP) du socket. Si l'appel à `bind` est omis, alors le système attribuera automatiquement une adresse au socket lors d'un appel à `sendto`, `recvfrom`, `listen` ou `connect`. Dans le cas d'un serveur, il est clair qu'on veut avoir une adresse locale bien définie et connue pour que les clients sachent comment communiquer avec lui.

On peut dire qu'en général l'adresse locale d'un client n'a pas d'importance (on peut donc s'abstenir d'utiliser `bind`), mais il existe des services pour lesquels ce n'est pas le cas.

5. L'appel système `connect`

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

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

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

En mode connecté (en l'occurence TCP), `connect` permet donc d'initier une connexion afin de pouvoir échanger des données.

En mode non connecté (comme par exemple UDP), `connect` permet simplement d'associer une adresse distante par défaut et ainsi pouvoir envoyer des datagrammes sans spécifier de destination (avec l'appel `send` au lieu de `sendto`).

Les arguments sont analogues à ceux de `bind`, sauf que pour le deuxième il s'agit de l'adresse distante.

6. L'appel système `listen`

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

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

Cet appel sert à indiquer qu'un socket doit être mis en attente de connexion. Il n'a donc de sens que pour le mode connecté (comme TCP). Le deuxième argument sert à indiquer la taille de la file d'attente des tentatives de connexions (c'est-à-dire combien de connexions peuvent être mises en attente avant qu'on les accepte avec l'appel `accept`).

7. L'appel système `accept`

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

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

Cet appel sert à accepter une connexion entrante. Le premier argument doit être un socket prêt à recevoir des connexions. Les second et troisième arguments servent à récupérer l'adresse distante de la connexion.

Note : Bien que ces arguments soient des pointeurs vers des variables dans lesquelles le système va écrire, il vaut mieux les initialiser correctement avant l'appel par soucis de portabilité (certains systèmes tiennent compte des information que ces variables contiennent avant de les remplacer).

8. L'appel système `gethostbyname`

```
#include <netdb.h>

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

Cet appel sert à récupérer des informations sur l'adresse IP associée à un nom de machine. En gros, cet appel va utiliser les mecanismes de résolution de noms configurés sur la machine courante (**`/etc/hosts`**, NIS, DNS, ...).

L'appel retourne un pointeur vers une structure qui a la forme suivante :

```
struct hostent {
  char  *h_name;       /* nom officiel de la machine */
  char  **h_aliases;   /* liste des autres noms */
  int   h_addrtype;    /* type d'adresse (AF_INET) */
  int   h_length;      /* longueur de l'adresse */
  char  **h_addr_list; /* liste des adresses */
};
#define h_addr  h_addr_list[0] /* pour compatibité */
```

9. Les appels système pour l'émission

En plus des appels d'écriture dans un descripteur de fichier `write` et `writev`, on dispose des appels `send`, `sendto` et `sendmsg`.

Les appels `send`, `write` et `writev` ne permettent pas de spécifier l'adresse de destination et ne sont dont utilisables que dans le cas ou une telle adresse n'est pas nécessaire (avec les sockets en mode connecté ou bien disposant d'une adresse destination par défaut spécifiée avec `connect`).

```
/* pour write */
#include <unistd.h>
/* pour writev */
#include <sys/uio.h>
/* pour send, sendto et sendmsg */
#include <sys/types.h>
#include <sys/socket.h>

int write(int sockfd, char *buf, int len);
int writev(int sockfd, struct iovec *vector, int count);
int send(int sockfd, char *buf, int len, int flags);
int sendto(int sockfd, char *buf, int len, int flags,
           struct sockaddr *to, int tolen);
int sendmsg(int sockfd, struct msghdr *msg, int flags);
```

Les arguments communs `sockfd`, `buf` et `len` sont respectivement le descripteur de socket, un pointeur vers les données à envoyer et la taille des données en octets.

Pour `writev`, l'argument `vector` est un tableau de structures de la forme :

```
struct iovec {
  char          *iov_base; /* adresse de début des données */
  unsigned      iov_len;   /* longueur des données */
};
```

L'argument `count` sert à spécifier la taille du tableau en nombre d'entrées. L'appel `writev` permet donc de spécifier une zone mémoire composée de plusieurs fragments.

L'argument commun `flags` permet de spécifier des options de transmission (voir la page du manuel). Dans notre cas, cet argument doit être mis à `0`.

L'appel `sendmsg` permet d'envoyer des données de façon exotique et ne nous servira pas dans ce TP. Des informations détaillées sont disponibles dans le manuel.

10. Les appels système pour la réception

Les appels pour la réception sont les duaux des appels pour l'émission. On dispose ainsi des classiques `read` et `readv`, ainsi que de `recv`, `recvfrom` et `recvmsg`.

```
/* pour read */
#include <unistd.h>
/* pour readv */
#include <sys/uio.h>
/* pour recv, recvto et recvmsg */
#include <sys/types.h>
#include <sys/socket.h>
```

```
int read(int sockfd, char *buf, int len);
int readv(int sockfd, struct iovec *vector, int count);
int recv(int sockfd, char *buf, int len, int flags);
int recvfrom(int sockfd, char *buf, int len, int flags,
             struct sockaddr *from, int *fromlen);
int recvmsg(int sockfd, struct msghdr *msg, int flags);
```

Les arguments sont analogues aux appels de réception. Dans le cas de `recvfrom` cependant, les pointeurs `from` et `fromlen` servent à recevoir l'adresse source des données recues (à noter que la remarque concernant `accept` s'applique aussi ici).

Annexe : les codes

```
/*
 * Code 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 local du serveur */
#define PORT 9600


int main(int argc, char *argv[])
{
  /*
   * Variables du serveur
   *
   * Déclarer ici les variables suivantes :
   *   - sockfd le descripteur de socket
   *   - structure d'adresse locale du serveur
   *   - structure d'adresse du client
   *   - taille de l'adresse du client
   */

  /*
   * Code du serveur
   *
   * - Ouvrir le socket du serveur
   * - Remplir la structure d'adresse locale du serveur :
   *   - la famille d'adresse
```

```
    *   - l'adresse IP
    *   - le port
    * - Spécifier l'adresse locale du socket du serveur
    */

   /*
    * Boucle générale du serveur (infinie)
    */

    while (1) {

      /*
       * Code de l'éintrieur de la boucle
       */

    }

    return 0;
}

/*
 * Code du client
 */

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

#define SIZE    100
#define PORT    9600

int main (int argc, char *argv[])
{
  /*
   * Variables du client
   *
   * Déclarer ici les variables suivantes :
   *   - sockfd le descripteur de socket
   *   - structure d'adresse du serveur
   *   - pointeur vers la structure descriptive de machine (hostent)
   *   - zone de mémoire destinée à accueillir la chaîne
   *     entrée au clavier
   *   - taille de la chaîne à envoyer
   */
```

```
    /*
     * Code du client
     *
     * - Ouvrir le socket du client
     * - Récupérer l'adresse IP du serveur à partir de son nom
     *   donné en ligne de commande
     * - Remplir la structure d'adresse du serveur
     * - Lire une ligne de l'entrée standard
     * - Envoyer la chaîne lue au serveur
     */

    return 0;
}
```