TP 1 - Programmation Réseau et Système

4TC PRS - INSA Lyon

Mise en place d'un serveur TCP

Dans ce TP, vous allez développer un serveur et un client TCP en utilisant l'API Sockets en C. L'utilisation du serveur et du client sera très simple : le texte qui va être écrit du coté client sera transmis au serveur, affiché à l'écran par le serveur, retransmis au client et affiché à l'écran par le client.

Pour pouvoir bien résoudre ce sujet, il y a deux pré-requis importants : savoir écrire un programme simple en C et savoir compiler des programmes C.

Vous devez créer deux fichiers, *serveur.c* et *client.c*, dans lesquelles vous allez programmer un serveur TCP et un client TCP respectivement.

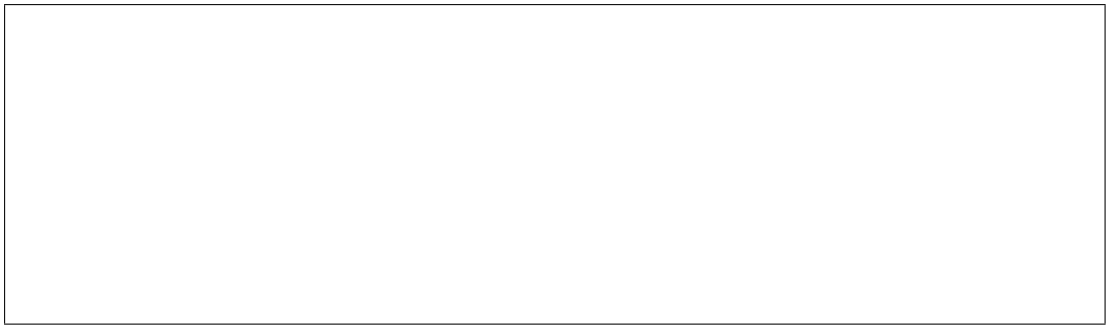
Nous allons commencer par la création d'une socket, qui se fait tout simplement en appelant la fonction *socket()*. Cette fonction prend trois paramètres :

- *domaine* - donne la famille d'adresses dans laquelle la socket sera créée. Pour le domaine IPv4, il faut utiliser la valeur AF_INET. Pour le domaine IPv6, il faut utiliser la valeur AF_INET6.
- *type* - le type de couche transport désirée, en fonction de critères applicatifs. Pour TCP, il faut utiliser la valeur SOCK_STREAM. Pour UDP, il faut utiliser la valeur SOCK_DGRAM. Si vous ne voulez pas utiliser de couche transport, vous pouvez utiliser la valeur SOCK_RAW.
- *protocole* - définit un protocole spécifique, au cas où le domaine choisi propose plusieurs protocoles du type désiré. Dans le cas de la pile TCP/IP, cette situation n'existe pas, et la valeur de ce paramètre est toujours 0.

La fonction *socket()* retourne une valeur **int**. En effet, une socket est traitée comme un fichier, dans lequel on peut écrire et lire, et la valeur retournée n'est rien d'autre que le descripteur de ce fichier.

Dans les fichiers *serveur.c* et *client.c*, créez une socket TCP et affichez la valeur du descripteur de fichier obtenu. Attention, en plus des en-têtes C classiques, vous devez inclure dans votre fichier les en-têtes *<sys/types.h>* et *<sys/socket.h>*, qui contiennent, entre autres, la définition de toutes les constantes nécessaires (AF_INET, SOCK_STREAM, etc.).

Question 1 *Quelle est la valeur du descripteur ? Quelle est l'explication de ce résultat ?*



Dans le cas d'une valeur négative du descripteur de fichier, la socket n'a pas été créée correctement. Traitez cette possibilité d'erreur dans votre code en arrêtant l'exécution du programme. De manière générale, toutes les fonctions de l'API Sockets retournent des valeurs **int**, inférieures à 0 en cas d'erreur. Pensez à traiter à chaque appel ces erreurs possibles ; ça peut être très important pour le débuggage.

Note : Certains systèmes d'exploitation gardent la socket bloquée même après l'arrêt du processus qui l'utilisait, au cas où il reste des packets en transit. Cependant, ce mécanisme peut poser des problèmes, surtout dans une phase de développement quand on veut pouvoir tuer des processus et les relancer sur la même socket. La fonction *setsockopt()* permet de modifier certains paramètres de la socket, dont le paramètre SO_REUSEADDR, une variable booléenne qui permet (ou pas) la réutilisation immédiate de la socket. Par défaut à 0 (faux), ce paramètre peut être passé à 1 (vrai) en utilisant le code suivant :

```
int reuse= 1 ;
setsockopt(descripteur, SOL_SOCKET, SO_REUSEADDR,
&reuse, sizeof(reuse)) ;
```

La socket que vous venez de créer n'est pour l'instant pas associée à la couche réseau et elle peut être utilisée soit du côté client, soit du côté serveur. Pour le serveur, il faut faire le lien entre cette socket et un couple (adresse IP, port). Pour ce faire, vous devez utilisez la structure *sockaddr*, qui est un container générique pour tout type d'adresse réseau. *sockaddr* utilise une représentation opaque, dans un format chaine de caractères suffisamment grand pour toute famille d'adresses. Pour simplifier la manipulation des adresses, dans le cas d'une couche réseau IP, on utilise *struct sockaddr_in*, définie dans l'en-tête<*netinet/in.h*>, qui a la même taille qu'une structure *sockaddr*, mais une représentation interne différente. Cela facilite un certain nombre d'opérations, mais il ne faut pas oublier que *struct sockaddr_in* n'est a priori pas connue par les fonctions de l'API Sockets, qui sont définies avec des paramètres *struct sockaddr*. Il ne faut donc pas oublier de faire un cast chaque fois quand vous passez une variable *sockaddr_in* comme paramètre d'une telle fonction.

Note : Il est conseillé de remettre à zéro une structure *struct sockaddr_ in* avant son initialisation, pour éviter tout problème d'accès à des zones mémoire libérées mais non effacées. Une possibilité pour cette remise à zéro utilise la fonction *memset()*, définie dans *<string.h>* :

```
struct sockaddr_in my_addr;
memset((char*)&my_addr, 0, sizeof(my_addr));
```

Pour pouvoir utiliser une structure *sockaddr_ in*, il faut initialiser trois de ses champs :

- *sin_ family* - qui doit correspondre à la famille (au domaine) de la socket à laquelle on associera l'adresse.
- *sin_ port* - le numéro du port sur lequel la socket sera ouverte.
- *sin_ addr* - l'adresse réseau associée à la socket (dans notre cas une adresse IP). Ce champ est, à son tour, une structure *struct in_ addr*. Pour l'initialiser, il faut pratiquement initialiser le champ *s_ addr* de la structure *sin_ addr*. Par exemple, cela peut se faire avec le code suivant :
  `my_addr.sin_addr.s_addr= INADDR_ANY;`

Du côté serveur, les deux adresses qu'on utilise le plus souvent pour connecter la socket sont INADDR_ANY et INADDR_LOOPBACK. La première option permet de connecter la socket à toutes les interfaces disponibles, tandis que la deuxième permet la connexion à l'hôte local (localhost).

Question 2 *Quelle est la valeur (en base 10) de INADDR_ ANY et INADDR_ LOOPBACK ? Qu'est-ce que cela donne en format IP ?*

Question 3 *Comment expliquez-vous l'utilisation d'une adresse de type INADDR_ ANY ? Pourquoi on ne récupère pas explicitement l'adresse IP d'une interface ?*

Différents architectures matérielles utilisent différentes ordres d'organisation en mémoire des octets qui forment une structure de données (la fameuse dispute *big endian*

vs. *little endian*). L'API Sockets a comme objectif de permettre l'interconnexion de machines hétérogènes et, pour cela, le module *<arpa/inet.h>* propose une série de fonctions permettant de traduire les structures dans un *format réseau* (qui est en réalité le format *big endian*). Ces fonctions sont :

- **int** *htonl*(**int** host_long) - passe un entier long du format machine en format réseau ;
- **int** *htons*(**int** host_short) - passe un entier court du format machine en format réseau ;
- **int** *ntohl*(**int** host_long) - passe un entier long du format réseau en format machine ;
- **int** *ntohs*(**int** host_short) - passe un entier court du format réseau en format machine.

Vous devez utiliser ces fonctions à l'initialisation de la structure *sockaddr_in* (pour le port et l'adresse IP) et chaque fois quand vous voulez transmettre une structure multi-octets sur le réseau.

Question 4 *Est-il nécessaire d'utiliser les fonctions* htonl() *et* ntohl() *lorsque l'adresse IP utilisée est INADDR_ANY ? Pourquoi ?*

Du côté client, vous devez aussi définir une structure *sockaddr_in*. Par contre, différemment du serveur, cette structure n'enregistrera pas le numéro de port et l'adresse locale (le client), mais ceux de la machine distante (le serveur auquel on veut se connecter). Cela veut dire que le client doit connaitre l'adresse IP et le port sur lesquels fonctionnent le serveur. Deux fonctions qui peuvent beaucoup faciliter l'initialisation du champ *s_addr* avec une adresse IP connue est *inet_aton()* et *inet_ntoa()*. L'avantage de cette fonction est qu'on peut lui donner comme paramètre une chaine de caractères contenant l'adresse IP, qu'elle convertira au bon format.

Question 5 *Afficher la valeur retournée par la fonction* inet_aton(). *Est-ce qu'il faut combiner cette fonction avec les fonctions* htonl() *et* ntohl() *?*

Pour le serveur, une fois la structure *sockaddr_in* initialisée, vous devez réaliser le lien entre cette structure et la socket, en utilisant la fonction **int** *bind()*. Cette fonction demande trois paramètres :

— *descripteur* - le descripteur de la socket serveur.
— *adresse* - un pointeur vers une structure *sockaddr* à associer à la socket. Vous pouvez utiliser un pointeur vers votre structure *sockaddr_in*, en n'oubliant pas le cast.
— *taille* - spécifie la taille, en octets, de la structure pointée par *adresse*.

Note : La fonction *sizeof()* vous permet d'obtenir automatiquement la taille d'une variable.

La socket est maintenant créée et initialisée. Il ne faut pas oublier que vous êtes du coté serveur : le prochain pas est donc de déclarer cette socket comme passive, en attente d'une demande de connexion. Cela se fait en utilisant la fonction *listen()*, avec deux paramètres : le descripteur de la socket en question et un paramètre **int** qui donne le nombre maximal de connexions en attente sur la socket.

Pour le client, vous n'avez pas besoin d'utiliser la fonction *bind()* (mais vous pouvez le faire). Le serveur n'a pas besoin de connaitre en avance l'adresse du client, car la connexion est toujours initialisée par ce dernier, et ses identifiants seront envoyés dans la demande de connexion. Si la socket client n'est pas associée à une structure *sockaddr_in*, le système d'exploitation va automatiquement lui associer une lorsqu'une demande de connexion est faite. Le client fait une demande de connexion en utilisant la fonction *connect()*, avec ces trois paramètres :
— *descripteur* - le descripteur de la socket client.
— *adresse* - un pointeur vers une structure *sockaddr* qui donne l'adresse du serveur.
— *taille* - comme dans le cas de *bind*, ce paramètre spécifie la taille, en octets, de la structure pointée par *adresse*.

Le serveur et le client doivent prendre la forme d'un *daemon*, c'est à dire d'un processus qui s'exécute en permanence, sans l'intervention de l'utilisateur. Pour cela, votre serveur et votre client doivent inclure, après l'appel à la fonction *listen()*, une boucle infinie.

Question 6 *Donnez deux syntaxes différentes pour réaliser une boucle infinie.*

A l'intérieur de cette boucle infinie, du côté serveur, vous devez rester en attente d'une demande de connexion et, éventuellement, la traiter. Le traitement d'une demande de connexion se fait à l'aide de la fonction **int** *accept()*. Cette fonction est bloquante, ce qui implique que votre programme n'avancera pas au delà de cet appel tant qu'une demande de connexion n'est reçue. La fonction *accept()* prend trois paramètres :

— *descripteur* - le descripteur de la socket ouverte par le serveur ;
— *adresse* - est un paramètre résultat. La valeur initiale doit être un pointeur vers une structure *sockaddr* non-initialisée. Après l'appel, la structure contiendra les identifiants du client ;
— *taille* - un pointeur vers un entier de type *socklen_t* initialisé avec la taille de la structure *sockaddr* vide, qui pointera après l'appel vers la taille réelle des données retournées.

Question 7 *Quelle est la valeur retournée par la fonction* accept() *? Que représente cette valeur ?*

Vous avez établi une connexion entre le serveur et le client. Tout ce qu'il vous reste à faire maintenant est de transmettre et de recevoir des messages. Pour ce faire, vous pouvez simplement écrire et lire sur les sockets correspondants, en utilisant les fonctions *read()* et *write()* définies dans l'en-tête<*unistd.h*>. Les deux fonctions ont la même signature :
— *descripteur* - le descripteur de la socket utilisée pour lire ou écrire ;
— *buffer* - un tableau de caractères, qui représente le buffer dans lequel vous allez lire ou écrire ;
— *taille* - un entier qui donne le nombre d'octets écrits dans le buffer (dans le cas de *write()*) ou la taille maximale du buffer de lecture (pour *read()*) .

Question 8 *Donnez le code utilisé pour transmettre et recevoir des messages coté serveur.*

Question 9 *Donnez le code utilisé pour transmettre et recevoir des messages coté client.*

Faites attention, les sockets TCP transportent un stream d'octets, pas un stream de packets, mais derrière ces sockets, il y a bien une fragmentation en segments TCP et packets IP. Cela veut dire que, si vous écrivez 20 octets sur une socket, vous n'êtes pas sûrs de lire 20 octets au récepteur. En effet, vous pouvez recevoir moins si vos données ont été fragmentées. Si la quantité de données à lire est importante pour vous, c'est à vous de développer un protocole qui vous permet de transmettre cette information avec les données utiles (plus de détails sur cela dans les prochains TPs).

Question 10 *Du coté serveur, récupérez le port et l'adresse IP du client. Quelles sont les valeurs de ces paramètres ?*

Côté client, si vous souhaitez découvrir le port de la connexion (du client), vous pouvez utiliser la fonction : *getsockname (int sockfd, struct sockaddr *addr, socklen_t *addrlen).*

Finalement, n'oubliez pas de fermer vos sockets, en utilisant la fonction *close().*