Serveur et client simple

Dans ce "cours", on va voir un serveur et un client simple. Le serveur communiquera avec un client à la fois et mettra en attente les autres.

Pour l'exemple, le client enverra des message et le serveur mélangera les lettres composants le message.

Classiquement les clients / serveur simple fonctionnent de cette façon :

Serveur (appelé)	**Client (appelant)**
Créer les sockets	
secoute = socket(...)	sclient = socket(...)
Attacher la socket à une adresse	
bind(secoute,...)	
Mettre la socket en mode écoute	
listen(secoute,...)	
Accepter une demande de connexion	**Demander une connexion**
sservice = accept(secoute,...)	connect(sclient,...)
Communiquer	
read(sservice,...)	write(sclient,...)
write(sservice,...)	read(sclient,...)
Fermer la connexion dans un ou deux sens	
shutdown(sservice,...)	shutdown(sclient,...)
Fermer la connexion et la socket associée	
close(sservice)	close(sclient)
Fermer la socket d'écoute	
close(secoute)	

Le serveur

Commençons par définir nos constantes, une pour le port utilisé par le serveur et une pour la taille maximale du BUFFER.

```
#define PORT 5015
#define BUFFER_SIZE 1024
```

Puis voici le main

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

    int nbLus;

    char message[BUFFER_SIZE];

    //1. création des sockets
    int secoute = socket(AF_INET, SOCK_STREAM, 0);
    if (secoute == -1){
        perror("Erreur de création de socket");
        return 1;
    }
    int sservice;

    //2. création des structure d'adresse
    struct sockaddr_in saddr = {0};
    struct sockaddr_in caddr = {0};
    unsigned int caddrlen;

    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(PORT);
    saddr.sin_addr.s_addr = inet_addr(argv[1]);

    //3. attacher la socket à une addresse
    if(bind(secoute, (struct sockaddr *) &saddr, sizeof(saddr))){
        perror("Erreur lord du bind");
        return 1;
    }

    //4. mettre en mode ecoute la socket
    if(listen(secoute, 5) == -1){
        perror("Erreur lors du listen");
    }

    while(1){
        //5. accepter une demande
        caddrlen = sizeof(caddr);
        sservice = accept(secoute, (struct sockaddr *) &caddr, &caddrlen);
        if(sservice == -1){
            perror("Erreur lors de la création de la socket sservice");
        }

        while(1){
            //6. communication
            nbLus = read(sservice, message, BUFFER_SIZE-1);
            if(nbLus == -1){
                perror("Erreur lors du read");
                return 1;
            }
            if(nbLus == 0){
                break;
```

```
            }

            message[nbLus] = '\0';

            shakeMessage(message, nbLus-1);
            write(sservice, message, nbLus);
        }
        //7. fermeture de la socket de service
        shutdown(sservice, SHUT_RDWR);
        close(sservice);
    }
    //8. fermeture de la socket d'écoute
    shutdown(secoute, SHUT_RDWR);
    close(secoute);
    return 0;
}
```

Quelques explications :

On commence par initialiser les variables nécessaires :

- nbLus : va contenir le nombre de caractère lu par read(). nbLus contiendra le nombre de caractère de la chaine + 1 (pour le '\0' de fin de chaine)
- un tableau de caractère message de taille BUFFER_SIZE : va contenir le message reçu par read(). Dans cette chaine, le caractère message[nbLus] devra être le caractère de fin de chaine (on le rajoutera toujours par prévention).

Et on implémente le serveur

1. **On crée les sockets avec la primitive `socket()`.**

   La primitive socket prends 3 paramètres :

   - le domaine :
     - AF_LOCAL / AF_UNIX pour la communication entre les processus sur le même système.
     - AF_INET / AF_INET6 pour la communication via les réseaux internet.
   - le type :
     - SOCK_STREAM : communiquent dans les réseaux ipv4 (AF_INET).
     - SOCK_DGRAM : datagramme.
   - le protocole : on utilisera 0 par défaut

   Elle renvoie un entier correspondant au descripteur de fichier (ou -1 si echec).

   Dans notre cas, on crée une socket d'écoute qui va ecouter les tentative de connexion entrante. On déclare une variable entière `sservice` pour la socket de service que l'on créera pour établir la communication avec le client.

2. **On crée une structure d'addresse.**

   ```
   struct sockaddr_in saddr = {0}; //Initialisation
   ```

```
saddr.sin_family = AF_INET;
saddr.sin_port = htons(PORT);
saddr.sin_addr.s_addr = inet_addr(argv[1]);
```

Il y a 3 éléments à spécifier pour créer une adresse : le domaine (family), le port, et l'adresse.

Quelques outils en plus :

- **htons** pour host to network (small 16bits) permet de convertir le numero de port en données compréhensible par tous les ordinateurs.
- **htonl** pour host to network (large 32bits).
- **ntohs** pour network to host (small 16bits).
- **ntohl** pour network to host (large 32bits).
- **inet_addr** converti une string en addresse ipv4.
- **INADDR_ANY** correspondant à toutes les addresses.
- **PORT = 0** laisse la machine choisir le port.

Dans notre cas, on crée une adresse pour la socket d'écoute et une autre pour le client (avec une variable qui va contenir la taille de l'adresse du client). L'adresse de la socket d'écoute correspond au paramètre réseau qui vont être exposés aux demandes, c'est par là que le serveur sera accessible.

3. **On attache l'adresse à la socket**

   Pour attacher une adresse à une socket, on utilise la primitive `bind()`.

   Cette primitive prends trois paramètres:

   - une socket
   - une référence vers une adresse au format sockaddr (d'où le casting).
   - la taille de l'adresse (**sizeof** revoie la taille d'une variable).

   Renvoie 0 en cas de succès et -1 en cas d'échec.

4. **Puis on ouvre l'écoute**

   Pour ouvrir l'écoute, on utilise la primitive `listen()`. Elle prend deux paramètre :

   - une socket (celle qui va être ouvert à l'écoute)
   - un entier correspondant au nombre de requêtes pouvant être mise en attente.

   Renvoie 0 en cas de succès et -1 en cas d'échec.

Ensuite on rentre dans la boucle du serveur qui va attendre les connexions, c'est une boucle sans fin, sauf si l'utilisateur l'arrête manuellement.

5. **On accepte une demande de connexion.**

   Pour accepter une demande, on utilise la primitive `accept()`. Elle prends en paramètre :

- la socket qui reçoit les demande (secoute).
- une structure d'adresse pour stocker les informations du client
- un pointeur vers une variable contenant la taille de l'adresse. (c'est pour cela que l'on a créer une variable caddrlen).

Elle renvoie le descripteur du fichier en cas de succès, ou -1 en cas d'échec.

`accept()` **mets en attente le processus jusqu'a la réception d'une demande de connexion.**

Dans notre cas, on stock le fichier dans sservice, la socket de service.

Et on rentre dans la boucle de service, c'est une boucle sans fin également jusqu'à que ce que connexion soit terminée (0 caractère lu).

6. **Et on communique**

   On commence par lire le message reçu avec la primitive `read()`. Cette primitive prend 3 paramètre :

   - le fichier sur lequel on lit (ici sservice)
   - un pointeur vers l'endroit ou va être stocké le message reçu. Dans notre cas, le tableau de caractère message.
   - la taille maximal de bits à lire (BUFFER_SIZE-1 pour compter le caractère de fin de chaine).

   `read()` mets en attente le processus jusqu'à la réception d'un message.

   `read()` renvoie :

   - le nombre d'octets lue en cas de succès
   - `0` si la connexion est terminée (si elle est terminée on sort de la boucle infinie).
   - `-1` en cas d'erreur.

   On poursuis en modifiant notre message, avec la fonction shakeMessage() qui va mélanger les caractères de notre message. Puis on le renvoie.

   Pour ecrire un message on utilise la primitive `write()` qui prend 3 paramètre également :

   - le fichier sur lequel on va écrire (ici sservice)
   - un pointeur sur le message à envoyer (ici notre message).
   - la taille du message que l'on envoie.

   `write()` est aussi bloquant, il va attendre que le processus distant acquite. Si le tampon est plein, il attend l'acquittement avant d'envoyer le reste.

7. **On ferme la connexion**

   Une fois que l'on est sortie de la boucle, cela signifie que la transmission est terminée, on peut fermer la connexion avec le processus distant.

   On utilise

```
shutdown(<socket>, SHUT_RDWR);
close(<socket>);
```

8. **On ferme la socket d'écoute à la fin du programme**

Le client

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

    int nbLus;
    char message[BUFFER_SIZE];

    /* 1. Création de la socket */
    int sclient = socket(AF_INET, SOCK_STREAM, 0);
    if(sclient == -1){
        perror("Erreur lors de la création de la socket");
    }

    /* 2. Création de la structure d'adresse DU SERVEUR */
    struct sockaddr_in saddr = {0};
    saddr.sin_family = AF_INET;
    saddr.sin_port = PORT;
    saddr.sin_addr.s_addr = inet_addr(argv[1]);

    /* 3. Tentative de connexion */
    while(connect(sclient, (struct sockaddr *) &saddr, sizeof(saddr)))
    printf("Connecté !\n");

    /* 4. Communication */
    while(1){
        write(1, "Message > ", strlen("Message > "));
        nbLus = read(0, message, BUFFER_SIZE-1);

        if(nbLus == -1){
            perror("Erreur de read sur l'entrée std");
        }
        if(nbLus == 0){
            break;
        }

        write(sclient, message, nbLus);

        nbLus = read(sclient, message, BUFFER_SIZE-1);
        if(nbLus == -1){
            perror("Erreur de read sur l'entrée std");
        }
        if(nbLus == 0){
            break;
        }

        write(1, "Message mélangé : ", strlen("Message mélangé : "));
        write(1, message, nbLus);
    }
```

```
    /* 5. Fermeture de la socket */
    shutdown(sclient, SHUT_RDWR);
    close(sclient);
}
```

Pour le client, la structure est plus simple.

1. **On crée la socket de communication**

   On crée la socket qui servira à communiquer, c'est sur cette socket que l'on recevre et que l'on enverra les messages

2. **On crée la structure d'adresse du serveur**

   La structure d'adresse que l'on crée correspond aux information de connexion vers le serveur. Le port auquel il est exposé, son adresse ip... *La création fonctionne comme au dessus*.

3. **On se connecte au serveur**

   On crée une boucle infinie qui essayera de se connecter tant que la connexion ne sera pas établie. Pour cela on utilise la primitive `connect()` qui prends 3 paramètre :

   - La socket de transmission.
   - Une structure d'adresse correspondante au serveur sur lequel on cherche à se connecter.
   - La taille de l'adresse.

   L'appel à cette primitive est bloquant, le processus sera bloqué jusqu'à ce que la connexion s'établisse (elle renvoie à alors 0) ou échoue (elle renvoie alors -1).

4. **On communique**

   De la même façon que tout à l'heure, on utilise les primitives `read()` et `write()` pour communiquer avec le serveur, via la socket `sclient`.

   Dans ce programe, on commence par lire le message que l'utilisateur écrit sur l'entrée standard (0). On l'envoie au serveur `write(sclient, message, nbLus);`. Puis on lit la réponse renvoyée par le serveur `nbLus = read(sclient, message, BUFFER_SIZE-1);`. Et on affiche la réponse sur la sortie standard (1).

   strlen() renvoie la taille d'une chaine de caractère

5. **On ferme la connexion à l'arrêt de la transmission.**

Et voilà !

En bonus la fonction shakeMessage()

```
#include <time.h> /* Pour la graine*/

void shakeMessage(char* message, int nbCaractères){
    srand(time(NULL)); //Initialisation de la graine
```

```
    for (int i = 0; i < nbCaractères; i++){
        int pos1 = rand() % nbCaractères;
        int pos2 = rand() % nbCaractères;

        char temp = message[pos1];
        message[pos1] = message[pos2];
        message[pos2] = temp;

    }
}
```

On prends en paramètres un pointeur vers le tableau de caractère et la taille de la chaine qu'il contient (le nb de caractère avant '\0'). Puis on échange nbCaractère fois.

Et les imports de fichiers

```
#include <stdio.h>
// Fournit des fonctions d'entrée/sortie comme printf().

#include <stdlib.h>
// Contient des fonctions de gestion de la mémoire comme malloc().

#include <string.h>
// Gère les chaînes de caractères avec des fonctions comme strcpy().

#include <unistd.h>
// Donne accès aux appels système comme fork().

#include <sys/types.h>
// Définit des types utilisés dans les appels système, comme pid_t.

#include <errno.h>
// Utilise la variable globale errno pour gérer les erreurs.

#include <sys/socket.h>
// Gère les sockets réseau avec des fonctions comme socket().

#include <netinet/in.h>
// Définit des structures de réseau, notamment sockaddr_in pour les
adresses IP.

#include <arpa/inet.h>
// Fournit des fonctions de conversion IP, comme inet_ntoa().

#include <netdb.h>
// Résout les noms d'hôtes avec gethostbyname().
```