TP1 Programmation répartie
Sockets TCP en C

Safa YAHI

Exercice 1 : Serveur daytime

Considérons le serveur TCP daytime suivant :

```
#include <arpa/inet.h>
#include <sys/socket.h>
#include <unistd.h>
#include <iostream>
#include <string.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define NUM_PORT 10013
#define BACKLOG 128
#define NB_CLIENTS 10000

using namespace std;

void exitErreur(const char * msg) {
	perror(msg);
	exit( EXIT_FAILURE);

}

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

	int sock_serveur = socket(AF_INET, SOCK_STREAM, 0);

	struct sockaddr_in sockaddr_serveur;

	sockaddr_serveur.sin_family = AF_INET;
	sockaddr_serveur.sin_port = htons(NUM_PORT);
	sockaddr_serveur.sin_addr.s_addr = htonl(INADDR_ANY);

	int yes = 1;
	if (setsockopt(sock_serveur, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int))
			== -1)
		exitErreur("setsockopt");

	if (bind(sock_serveur, (struct sockaddr *) &sockaddr_serveur,
			sizeof(sockaddr_in)) == -1)
		exitErreur("bind");
```

```
        if (listen(sock_serveur, BACKLOG) == -1)
                exitErreur("listen");

        char * msg;
        time_t date;

        cout << "Serveur DayTime lancé  sur le port " << NUM_PORT << endl;

        for (int i = 1; i <= NB_CLIENTS; i++) {

                int sock_client = accept(sock_serveur, NULL, NULL);
                if (sock_client == -1)
                        exitErreur("accept");

                date = time(NULL);
                msg = ctime(&date);

                if (write(sock_client, msg, strlen(msg)) == -1)
                        exitErreur("write");

                close(sock_client);

        }
        close(sock_serveur);
        return 0;
}
```

1) Déterminez le type de la socket créée avec :
   `int sock_serveur = socket(AF_INET, SOCK_STREAM, 0);`
2) Quelle est l'adresse de la socket d'écoute de ce serveur (adresse IP et numéro de port en local) ?

3) Combien de sockets (passives et actives) utilise-t-on côté serveur ?

4) Lancez le serveur sur votre machine (téléchargez le fichier .cxx depuis Ametice et ne faites pas copiez coller depuis cet énoncé, compilez et ensuite exécutez-le). Vérifiez qu'il utilise bien l'adresse de socket que vous avez identifiée dans la question précédente. Pour cela, utilisez la commande "**netstat -ltn**" ("l" pour "listening", "t" pour "TCP" et "n" pour "numeric").

5) L'utilitaire Telnet peut être utilisé comme client TCP. Dans un terminal, il suffit de faire «telnet IP_serveur port_serveur » et d'échanger ensuite les messages qu'il faut.

Depuis votre machine, envoyez une requête à votre serveur via Telnet en utilisant l'adresse 127.0.0.1 (localhost). Refaites la même chose en utilisant l'adresse Ethernet IPv4 de votre machine (cette dernière peut être obtenue avec ifconfig).

6) Envoyez à votre serveur une requête via Telnet depuis la machine de votre voisin(e) (en utilisant bien sûr votre adresse Ethernet et non pas 127.0.0.1).

7) L'utilitaire "netcat" ou "nc" peut être aussi utilisé comme client TCP. Testez-le.

8) Lancez une deuxième instance de votre serveur (sans arrêter la première). Que se passe-t-il ? Repérez l'instruction du serveur qui nous a permis d'afficher ce message d'erreur.

9) Essayez maintenant de le lancer sur le port 13. Quel est le problème ?

Notez l'utilité de vérifier à chaque fois la valeur de retour des fonctions de l'API sockets afin de bien gérer les éventuelles erreurs !

10) =>Arrêtez le serveur et modifiez son code pour qu'il écoute sur l'adresse IPv4 de votre interface Ethernet (et non pas 127.0.0.1). On garde le même port.

Pour ce faire, remplacez :

```
sockaddr_serveur.sin_addr.s_addr = htonl(INADDR_ANY);
```

par :

```
inet_aton(« adresse IP de votre interface Ethernet », & sockaddr_serveur.sin_addr) ;
```

et non pas par :

```
inet_aton(«adresse IP de votre interface Ethernet », & sockaddr_serveur.sin_addr.s_addr) ;
```

=>Lancez-le et vérifier la nouvelle adresse de socket d'écoute par “netstat -ltn”. Vérifiez qu'il répond aux requêtes de type: telnet votre_adresse_Ethernet 10013 ? Est-ce qu'il répond aux requêtes de type « telnet localhost 10013 »?

=> Mettez en commentaire cette dernière modification et réutilisez INADDR_ANY pour que votre serveur écoute sur toutes les interfaces réseaux de sa machine.

11) Un serveur a besoin parfois d'avoir au niveau applicatif l'adresse IP d'un client. Par exemple, ça peut être utile pour mettre à jour un fichier log (où il mémorise les connexions des différents clients) ou de vérifier si un client n'est pas « black listé » s'il maintient une ACL.

=> Modifiez le serveur pour qu'il affiche, à chaque connexion d'un client, l'adresse IP et le port de ce dernier. Utilisez pour ce faire le deuxième paramètre de la fonction accept( ). Pensez à utiliser la fonction ntohs() pour afficher le port du client, et la fonction inet_ntoa() pour afficher son adresse IP.

=> Depuis une même machine, lancez plusieurs clients pour constater que le port client change d'une exécution à une autre (car il est choisi arbitrairement par l'OS).

12) [Facultatif] Refaites la question précédente en utilisant cette fois la fonction getpeername() et non pas la fonction accept(). Référez vous au manuel associé.

Exercice 2 : Client DayTime

=> Implémentez le client TCP daytime tout en gérant les éventuelles erreurs. Aussi, prenez en compte le fait qu'un message TCP peut arriver par segments au niveau du destinataire (même si l’émetteur fait un seul write()). Ainsi, un seul appel à la fonction read(), ne garantit pas de le récupérer dans sa totalité.
=> Testez votre client en local et avec vos voisins via le serveur DaytTime précédent.

Exercice 3 : Client Web

HTTP (Hypertext Transfer Protocol) est un protocole utilisé pour l'échange de données entre clients et serveurs sur le Web depuis 1990. C'est un protocole de la couche application qui opère au dessus du protocole TCP. Les clients HTTP les plus fréquemment utilisés sont les navigateurs Web. On peut

citer aussi les aspirateurs de sites. Pour les serveurs HTTP, il existe plusieurs implémentations telles que : Apache HTTP Server ou IIS. Par défaut, un serveur HTTP utilise le port 80. Il existe plusieurs versions du protocole HTTP : HTTP/0.9, HTTP/1.0 et HTTP/1.1 et HTTP/2. Ces deux derniers sont des standards.

Pour simplifier les choses, nous nous intéressons ici uniquement au protocole HTTP/0.9 pour sa simplicité (même s'il ne constitue pas un standard).

HTTP/0.9 est la première version de ce protocole. Dans cette version, nous avons un seul type de requêtes, à savoir la requête **GET** qui permet de demander une ressource (page HTML, image, etc). Le dialogue entre le client et le serveur se veut très simple :

- Tout d'abord, le client se connecte au serveur
- Le client envoie ensuite au serveur une requête GET dont la syntaxe est comme suit :

GET /chemin_fichier\n

où /chemin/fichier est le chemin du fichier demandé par le client.

- Le serveur répond au client en lui envoyant le contenu du fichier demandé
- Enfin, le serveur ferme la connexion

=> Implémentez un client Web pour le protocole HTTP/0.9 en utilisant les sockets TCP en C/C++ Testez-le avec le serveur Web donné par votre enseignant.e.

Exercice 4 (Facultatif) : Chat à tour de rôle entre client et serveur

Considérons une application **client / serveur TCP** qui s'échangent, à tour de rôle, des messages. Plus précisément :

- le client se connecte au serveur,
- lui envoie un premier message que l'utilisateur aurait saisi.
- le serveur lui répond par un autre message saisi par l'utilisateur
- le client lui envoie un nouveau message et ainsi de suite jusqu'à ce que le client envoie le message “Bye” ou bien il y a déconnexion.

On suppose que le serveur traite les clients séquentiellement et non pas “simultanément”.

Par ailleurs, on sait que pour récupérer le message envoyé par l'autre en TCP, il faut une boucle de read(). Dans cette application, la condition d'arrêt de cette boucle ne peut pas être « read()=0 » car il s'agit d'un dialogue à plusieurs étapes et la déconnexion (et donc read() retourne 0) ne se fait qu'à la fin.

On suppose alors que les messages échangés sont sous forme de lignes se terminant par « \n ».

=> Écrivez d'abord le serveur en C/C++ et testez-le avec un client Telnet. Notez que lorsque l'utilisateur saisit par exemple « Bye » en Telnet, le message envoyé est « Bye\r\n ». Pensez à utiliser getline() au lieu de cin pour lire les messages afin de prendre des « ».

=> Écrivez le client en C/C++

=>Testez cette application en réseau pour discuter avec vos voisin(e)s.