INFO-F201 - Rapport projet Chatroom

Ismael Secundar 504107 & Dave Pikop Pokam 506979

Décembre 2021

Table des matières

1 Introduction

Le présent rapport décrit la réalisation d'un salon de discussion client/serveur en utilisant les sockets afin d'asseoir nos connaissances sur la programmation réseau. En clair, il s'agit de réaliser la communication entre plusieurs clients connectés à un serveur. Il est donc question pour nous d'implémenter deux fichiers sources : le client et le serveur. Ayant fait la majorité des travaux pratiques en C, nous avons fait le choix d'utiliser ce langage de programmation afin de simplifier nos recherches sur internet sur l'usage des sockets dans d'autres langages et en nous limitant sur ce que nous avons vu au cours théorique et aux TP jusqu'ici. Dans la suite, nous seront amenés à décrire le projet en spécifiant nos choix d'implémentations, ensuite, donner les limitations du projet et enfin montrer les difficultés rencontrées.

2 Serveur

2.1 Description

- Il attend une connexion entrante sur un port réseau ;
- À la connexion d'un client sur le port en écoute, il ouvre un socket local au système d'exploitation.
- Suite à la connexion, le processus serveur communique avec le client suivant le protocole prévu par la couche application OSI.
- Tant que l'utilisateur ne ferme pas le serveur il restera à l'écoute d'un potentiel client qui voudrait se connecter au serveur.

2.2 Implémention

Le code source du serveur a été implémenté avec l'appel système select pour gérer la réception simultanée des messages côté client. Il est divisé en plusieurs fonctions et comporte aussi une structure définissant un type de données. Ainsi, nous retrouvons :

- user : Nous avons défini une structure qui va stocker le numéro de socket du client ainsi que son pseudo.
- sigintHandler : Cette fonction vérifie le type de signal envoyé par le serveur. S'il s'agit d'un SIGINT (signal engendré par la touche CTRL+C), alors cette fonction provoque la fermeture du serveur et par la suite, la fermeture de tous les clients qui lui sont connectés.
- parse_args : Cette fonction vérifie si l'utilisateur a bien mis tous les arguments nécessaires à l'exécution, et si les types des arguments sont correctes. Dans notre cas, cette fonction vérifie si le port a bien été passé en argument et s'il est de type entier.
- main : La fonction main, représentant le corps du serveur est structuré en 4 étapes principales :
  - Création du socket serveur et configuration de ses paramètres à l'aide du contexte d'adressage SOCKADDR_IN permettant la configuration de la connexion. En parallèle, nous avons ajouté la fonction setsocket() qui permet de réutiliser le même port à chaque connexion du serveur avec le paramètre SO_REUSEADDR.
  - Connexion du serveur au client : Cette étape nécessite l'appel aux fonctions bind() et listen() ; bind() sert à lier le socket serveur à un port ; et listen() met ce socket en état d'écoute et boucle jusqu'à ce qu'un éventuel client se connecte.
  - Sélection des sockets : Nous avons une table que nous définissons pour stocker les sockets connectés. Ainsi, nous utilisons l'appel système select() pour gérer la communication simultanée entre les sockets. Elle est assez spéciale car tout le gestionnaire de communication se passe dans un seul processus et dans un seul thread.
  - Transmission et réception des messages : elles s'effectuent grâce aux fonctions send() et recv() ou read() et write(). Tout d'abord, à chaque connexion du client, nous récupérons toutes ses informations (port et socket), que nous stockons dans une table de type structure pour avoir un

identifiant unique de chaque client lors d'une communication.
— Fermeture du socket : Pour gérer la déconnexion du client ou du serveur.

Quelques détails d'implémentation : La fonction signal a été utilisée pour capturer le signal "CTRL+C". Celle-ci fait appel à une autre fonction, sigintHandler, qui a été mentionné plus haut.

2.3 Exécution

Pour lancer le serveur, après avoir lancé le Makefile, il suffit d'exécuter la commande suivante sur le terminal :

```
$./serveur <port>
```

3 Client

3.1 Description

— Il se connecte au serveur via le port réseau et à l'adresse ip du serveur.
— Lorsque la connexion est acceptée par le serveur, il est prêt pour communiquer.
— Une fois qu'il envoie un message, celui-ci est transmis à tous les autres clients connectés au serveur.
— Il peut se déconnecter en appuyant sur "CTRL+D"
—

3.2 Implémentation

— buffer : Représente la structure qui stocke toutes les données nécessaires pour construire le format du message à envoyer, à savoir : la longueur du message, le timestamp et le message. Cette structure permet de respecter le type de chaque variable, comme il nous a été demandé. Pour le timestamp, nous avons opté pour un format d'affichage en HH :MM :SS (par exemple 23 :18 :19). Cela nous semblait plus logique pour un chat d'afficher l'heure.
— parse_args : Comme dans le code Serveur, elle vérifie les arguments passés en paramètre et leurs types.
— start_of_msg_char : Permet de vider le tampon associé au flux de sortie spécifié, pour donner l'impression que nous sommes dans un Chat, et donc à chaque début de ligne on aura un ">" qui est affiché.
— clean_up_msg : Elle sert à nettoyer le message, c'est-à-dire à remplacer le caractère de retour à la ligne par le caractère de fin de ligne.
— recv_msg_handler : Représente le thread qui s'occupe de la réception du message. Si la réception du message se passe mal alors le client a sans doute reçu un signal du serveur et doit donc se déconnecter.
— main : Comme dans le serveur, cette fonction principale représente le corps du programme client où nous configurons le socket client avec le contexte d'adressage, nous faisons appel à la fonction connect() pour établir la connexion au serveur, puis nous créons un thread pour la gestion de la réception des messages et le thread principal s'occupe de l'envoie des messages.

Détails d'implémentation : Pour l'envoie des messages, nous utilisons une structure qui contient des données pour construire le message à envoyer.

Nous n'avons pas utilisé les mutex car dans le code client, nous avons juste deux threads qui n'ont pas un accès concurrentiel aux mêmes ressources. D'une part nous avons le thread principal qui gère l'envoie du message et le thread noyau que nous avons créé pour la réception des messages. Ces deux threads n'ont pas de données communes à partager.

3.3 Exécution

Pour lancer le serveur, il suffit d'exécuter la commande suivant sur le terminal, après avoir lancé le Makefile et le serveur :

```
$ ./client <pseudo> <ip_serveur> <port>
```

4 Makefile

4.1 Implémentation

Pour le MakeFile, nous avons utilisé le flag -lpthread vue que nous utilisons un thread dans le fichier client. De plus, comme il nous a été demandé de vérifier les erreurs, nous avons aussi ajouté -Wall.

5 Common.h

Le header common.h permet de vérifier si chacune des fonctions se lance sans erreur, avec la fonction check.

6 Tests réalisés

Afin de vérifier que tout marche bien, nous avons géré plusieurs cas :

1. Dans le cas où un serveur est exécuté avec un port, nous vérifions s'il sera possible à l'utilisateur de relancer le serveur avec le même port.
2. Si l'utilisateur veut terminer le serveur, il doit appuyer sur "CTRL+C". Une fois appuyé, le serveur va déconnecter les clients et ensuite se fermer.
3. Nous avons vérifié si le code se lançait sur deux machines virtuelles :
   (a) Linux dave-VirtualBox 5.11.0-41-generic #45 20.04.1-Ubuntu SMP Wed Nov 10 10 :20 :10 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
   (b) Linux ubuntu-linux-20-04-desktop 5.4.0-90-generic #101-Ubuntu SMP Fri Oct 15 20 :02 :26 UTC 2021 aarch64 aarch64 aarch64 GNU/Linux
4. Nous avons testé notre code avec celui d'un autre groupe et sur leurs machines.
5. Même si tous les clients se déconnectent, le serveur reste actif au cas où d'autres clients voudraient se connecter.
6. Un client se déconnecte avec "CTRL+D", le serveur affiche le pseudo du client déconnecté et son socket.

7 Difficultés rencontrées et solutions apportées

7.1 L'utilisation du langage C

L'usage de ce langage a été un gros frein dans l'implémentation du serveur et du client, car il n'est pas orienté objet du coup il est presque impossible voire difficile de définir des classes pour organiser les données à moins d'importer des bibliothèques.

7.1.1 malloc & free

Tout d'abord, nous avons eu du mal à gérer les char pointers. Il fallait bien faire un malloc et un free à chaque fois pour éviter des erreurs comme "segmentation fault (core dumped)". Parfois, cela prenait beaucoup de temps pour trouver exactement d'où l'erreur venait. Il fallait passer au crible tout le code parce qu'on avait pas un endroit précis où le code a mal fonctionné.

7.1.2 recv & read

Ensuite, nous avons eu du mal à déconnecter un client tout en gardant le serveur et les autres clients actifs. Nous nous sommes rendu compte que cela provenait du recv. Celui-ci a été remplacé par un read dans le code du serveur. Après ce changement, lorsqu'un client appuyait sur CTRL + D, il se déconnectait et le serveur ainsi que les autres clients restaient actifs.

7.1.3 Nettoyage du buffer

De plus, notre message s'affichait mais quand un message plus court que le message d'avant était envoyé, le message s'écrivait sur l'ancien message. Pour éviter que cela se produise, nous avons dû vider le buffer à chaque fois. Pour cela, nous avons utilisé la fonction memset.

7.1.4 Parsing des arguments

Au tout début, lorsque nous avons commencé à coder, nous avons eu du mal à récupérer les arguments. Nous avons assigné les variables du struct aux variables qui ont été données par l'utilisateur. Mais nous ne pouvions pas faire un simple "=". Nous avons dû faire un strcpy qui nous permettait de copier le valeur pointé par un pointeur vers la destination souhaitée.

7.1.5 Préciser -lpthread dans le makefile

Nous avons remarqué qu'il fallait forcément ajouter à l'éditeur de lien côté client la librairie -lpthread lors de la compilation sinon une erreur s'affichait.

8 Conclusion

En somme, il était question durant ce projet de créer un salon de discussion pour que chaque client qui se connecte puisse envoyer et recevoir des messages aux autres via le serveur. Ce mécanisme a été implémenté en C, d'un côté nous avons le code du client qui utilise le multithreading et de l'autre côté le serveur qui utilise l'appel système select et nous nous sommes rendu compte des avantages mais aussi des inconvénients de ces différents mécanismes implémentés.

Aussi, ce projet nous a permis de pouvoir manipuler d'importantes notions de la programmation réseau telles que les sokects, nous avons eu l'opportunité de mettre en évidence les connaissances acquises au cours théoriques et aux TP et nous avons pu ainsi nous rendre compte de certains problèmes liés au langage C qui est nouveau pour nous.