Info F-201 : Projet d'OS
Rapport

Auteurs: Liefferinckx Romain, Rocca Manuel, Radu-Loghin Rares
Matricules: 000591790, 000596086, 000590079
Section: INFO

2024, 17 Décembre

Contents

1 Introduction

1.1 Présentation du projet et contexte

Dans le cadre de notre cours d'OS, nous avons réalisé un projet qui consiste à implémenter unz messagerie en ligne comprenant un serveur en C. Ce chat, permet la communication entre deux clients via des sockets en ligne et sur des ordinateurs différents. La messagerie est composé de deux parties, celle décrite ci-dessus et une autre écrite en bash, faisant office de chat-auto permettant la facilitation de l'utilisation du programme chat. Ce chat-auto est conçu pour simuler un client en automatisant les taches d'envoies de messages à l'interlocuteur.
Le projet se compose donc de deux parties : le programme de messagerie avec (client) et (server) et le script Bash (chat-auto).

1.2 Objectifs du projet

L'objectif de ce projet est de mettre en pratique les concepts vus en cours d'OS, notamment les sockets, les threads, la gestion des signaux, les mutexs,et le bash. Ce rapport décrit les choix d'implémentation, les difficultés rencontrées et les solutions mises en œuvre utilisée dans la construction de ce projet.

2 Choix d'Implémentation

2.1 Choix du langage

Dans le cadre de ce projet, nous avions le choix entre le C et le C++ comme langage de programmation. Nous avons fait le choix d'utiliser du C car c'est en C que nous avons vu la matère durant les séances de travaux pratiques et que la solution du premier projet a été faites en C. De plus, ayant fait le premier projet en C, il aurait été plus compliqué de faire le deuxième en C++ car il aurait fallu faire des recherches supplémentaires pour comprendre les différences entre les deux langages.

2.2 Choix de la méthode de gestion des clients

Dans le cadre de ce projet, nous avons fait le choix de gérer les clients via des threads. Chaques clients est géré par un thread différent. Cela permet de gérer plusieurs clients en même temps et de ne pas bloquer le serveur lorsqu'un client envoie un message. C'est la fonction handle_client qui est exécutée dans un thread différent pour chaque client. Nous avons choisi cette méthode car ceux-ci présentent plusieurs avantages :

- **Concurrence et Réactivité :** L'utilisation des threads permet de traiter les requêtes des clients de manière concurrente. Chaque client est géré indépendamment, ce qui améliore la réactivité du serveur. Si un client envoie seulement des messages de manière sporadique, cela n'affecte pas les autres clients.
- **Simplicité de l'implémentation :** Les threads sont relativement simples à utiliser (pthread).
- **Utilisation efficace des ressources :** Les threads partagent le même espace mémoire, ce qui permet une communication rapide et efficace entre eux. Les processus quant a eux n'auraient pas été une solution légère en mémoire pour gérer les clients.
- **Gestion à grande échelle :** Avec les threads, le serveur peut facilement gérer un grand nombre de clients simultanément. Chaque nouveau client est simplement un nouveau thread, ce qui permet une gestion dynamique des connexions.

Alternatives considérées :

- **Multiprocessing :** Utiliser des processus multiples pour chaque client aurait été une alternative. Cependant, cette méthode consomme plus de ressources car chaque processus a son propre espace mémoire. La communication entre processus est également plus complexe et moins efficace que la communication entre threads.
- **Multiplexing/Polling (select, poll, epoll) :** Une autre méthode aurait été d'utiliser des techniques de polling comme `select`, `poll` ou `epoll`. Bien que ces techniques soient certainement efficaces pour gérer de nombreuses connexions simultanées sans bloquer, elles sont plus complexes à implémenter et n'ayant pas beaucoup de temps pour ce projet ainsi que des connaissance restreinte sur le sujet, nous ne nous sommes pas risqué a les utiliser.

En définitive, l'utilisation des threads nous a permis de créer un serveur fluide, efficace et facile à maintenir, capable de gérer plusieurs clients simultanément sans être trop compliqué a gérer.

2.3 Gestion des accès concurrents

Pour que notre messagerie fonctionne correctement, la gestion des accès concurrents était un défi cruciale à réaliser pour assurer l'intégrité des données partagées entre les différents threads. Nous avons donc utilisé des mutex pour synchroniser l'accès aux ressources partagées, telles que la liste des clients connectés. Voici quelques exemples ou nous les avons utilisé :

- **Protection de la liste des clients :** La liste des clients connectés est une ressource partagée entre les différents threads. Pour éviter les conditions de course et garantir la sécurité des données, nous avons utilisé un mutex nommé `clients_mutex`. Avant d'ajouter ou de supprimer un client de la liste, le thread doit verrouiller ce mutex. Une fois les opérations terminée, le mutex est déverrouillé.
- **Fonctions critiques :** Les fonctions qui modifient la liste des clients, telles que `addClient` et `remove_client`, sont protégées par des mutex. Cela garantit que ces fonctions ne peuvent pas être exécutées simultanément par plusieurs threads, évitant ainsi les conflits et les incohérences.
- **Comptage des clients actifs :** Pour compter le nombre de clients actifs, nous avons également utilisé un mutex pour protéger l'accès à la liste des clients. Cela garantit que le comptage est précis et que la liste n'est pas modifiée pendant l'opération.

2.4 Gestion des signaux

3 Difficultés Rencontrées et Solutions

4 Solutions Originales et Améliorations

4.1 Gestion des variables globales

Lors de la création du projet, étant donné que nous avons choisit le C comme langage de programmation, il était interdit d'utiliser de l'orienté objet et donc aucune variable dans les instanciations des objets. Le premier réflexe est donc de mettre "const <nom de la variable>= "valeur" lorsque celle-ci ne doit pas être modifié et "<nom de la variable>= "valeur"

lorsqu'elle peut l'être. Nous avons alors fait le choix de mettre tout les variables globales non modifiable sous la forme "#define <nom de la variable><valeur>". Cela permet d'avoir une lisibilité accrue, et économiser de l'espace mémoire. Des problèmes de compatibilité entre types pourraient apparaître, certes, mais nous n'utilisons les variables globales que comme types simples comme int ou str, tout en faisant attention à leur contexte d'utilisation, nous permettant ainsi d'éviter ces erreurs.

5 Conclusion

Ce projet nous a permis de mettre en pratique et de se familiariser avec les concepts vus en cours d'OS, tels que les sockets, les signaux, les threads, et les mutex en C et de pratiquer le bash. Celui-ci, nous a appris à utiliser les outils de programmation en C comme sigaction, les fonctions relatives aux sockets telles que listen, bind,... . Ce projet nous a également permis de travailler à nouveau avec le même groupe que pour le premier projet, ce qui nous a permis de nous améliorer avec ce même groupe, impliquant une évolution dans la gestion et répartition du travail.