Rapport Projet OS 2

BENOUDA Haytam 572722 BADI BUDU Chris 569082

ANNAIM Marouane 572692

Année 2023-2024

Contents

1 Introduction

Dans le cadre du cours de Systèmes d'exploitation INFO-F201, nous avons réalisé un projet d'une application de comparaison d'image. Nous avons reçu comme tâche d'améliorer cette application dans notre second projet du cours en modifiant notre implémentation. Il a fallu pour cela que nous utilisons des threads et des sockets, découpant ainsi le projet en une partie serveur et une partie client. Ce rapport a donc pour but d'expliquer nos choix d'implémentation, les problèmes rencontrés et comment nous y avons fait face.

2 Client

Pour commencer, dans le code du client, nous avons implémenter différentes fonctions à commencer par LectureImageBMP qui nous permet d'ouvrir l'image que le client entre sur le stdin en donnant son chemin d'accès. Cette fonction va par la suite lire ce que contient l'image et stocker son contenu dans une variable de type char. Nous avons également ajouté une vérification de la taille de l'image pour qu'elle n'excède pas les 20ko.

Ensuite, nous avons la fonction clientListener qui s'occupe de recevoir le chemin de l'image entrée par l'utilisateur via le fgets qui va appeler la fonction LectureImageBMP pour avoir le contenu de l'image du client stockée. Nous vérifions aussi que le chemin donné par le client n'excède pas une taille de 1000 caractères. Après cela, nous pouvons enfin envoyer au serveur le contenu de l'image à l'aide du socket après avoir fait une demande au serveur de communiquer via celui-ci et d'avoir été accepté. La connexion au socket se fait dans le main du code du client. Dans la deuxième partie de la fonction clientListener, il s'agit du code permettant la réception du résultat final de la comparaison de l'image par le serveur au client. Nous affichons les messages indiquant les résultats obtenus après les avoir réceptionnés via le socket. Le résultat est stocké dans une struct client_data qui est la variable centrale de la fonction. Elle stocke dans la variable chemins_longueur la somme du nombre de caractères des différents chemins d'accès transmis par le client pour qu'on puisse vérifier que ce nombre n'excède 999, la meilleure image reçue du serveur à l'issu des comparaisons et l'image reçue du client ainsi que ses information dans une struct Client. Cette struct contient le nom du chemin d'accès de l'image provenant de la machine du client, un tableau de char où est stocké le contenu de l'image, sa taille qui sera utile lors de l'appel à la fonction pHashRaw pour initialiser le code de hachage perceptif de son image qui sera stockée dans la variable uint64_t de la struct.

3 Serveur

Par la suite, dans le code du serveur, nous avons une première fonction compare_image qui s'occupera de lancer la comparaison entre l'image du client et

ceux de la banque d'image. Le meilleure résultat sera stocké dans la variable globale meilleure_image qui est une struct image. La struct image contient le code de hachage perceptif de l'image, son chemin d'accès ainsi que la distance qui la sépare de l'image avec laquelle elle a été ou sera comparée. Pour revenir sur la fonction compare_image, elle sera exécutée par trois threads de façon concurrente pour avoir une comparaison rapide et efficace. Nous utilisons une mutex pour gérer la concurrence de cette fonction par ces trois threads pour qu'il n'y ait pas de problème de concurrence au sein de la zone critique d'exécution.

La fonction suivante, getPictures, nous permet d'ouvrir le dossier de la banque d'image et d'y ajouter chaque image dans un tableau qui nous sera utile plus tard. Nous utilisons pour cette fonction le fichier list-file réalisé lors du projet 1 nous permettant de lister tous les fichiers contenus dans un dossier donné comme argument. Après cela, nous allons utiliser la fonction PHash pour calculer le code du hashage perceptif de l'image à l'aide de son chemin qu'on a stocké dans le tableau comme expliqué plus haut et on ajoute le hash de l'image dans le tableau également.

Également, nous avons implémenté une fonction create_socket qui se charge de créer le socket principal et l'initialiser au bon port et à la bonne adresse. Il permet également de mettre 10 client en attente d'être accepté à communiquer via le socket, cela se fait à l'aide de la fonction listen.

De plus nous y avons créé un socket qui est stocké dans une struct socket_for _client qui contient la valeur du nouveau socket (s'il est bien initialisé) et la struct to_compare_image qui contient une valeur amount_images qui contient le nombre d'images que contient le tableau de struct image, un tableau de struct image où seront stockées les images de la librairie et une struct client. Ces deux derniers éléments vont permettre aux threads de pouvoir comparer les images avec celles du client dans la fonction compare_image sans manquer d'informations sur les données du clients et celles des images de la librairie dans le tableau.

En ce qui concerne la gestion des 1000 clients connectés simultanément au serveur, tout cela est géré dans la fonction connectToClient qui gère la connexion des clients au serveur. Nous avons utilisé un sémaphore qui contrôle le nombre de threads exécutés. Chaque thread exécuté correspond à un client et avec l'appel à sem_wait, img-search ne boucle pas à l'infini mais attend la prochaine connexion d'un client pour ne pas avoir des clients à l'infini.

Comme expliqué ci-dessus, chaque thread correspond à un client. Le thread va appeler serveClient pour pouvoir recevoir les images du client qui sont stockées dans des structs Client. Par la suite, cette image sera comparée à celle de la librairie du serveur qui sont stockées dans les struct to_compare_image.

Pour qu'il n'y ait pas de problème lors de l'exécution des threads pour chaque client, on utilise une mutex pour ne pas avoir de problèmes de concurrence

d'exécution des threads lors de l'accès à la section critique.

4 Signaux

Nous gérons les signaux SIGINT et SIGPIPE dans la partie serveur et la partie client séparemment. Pour la partie du serveur nous avons la fonction Signal-Handler qui va s'occuper de tuer les trois threads qui lancent la comparaison d'image, permettre à un client en attente de se connecter au socket en libérant la place. S'il s'agit d'un signal SIGINT, nous terminons le serveur mais s'il s'agit d'un signal SIGPIPE, nous envoyons au client une demande d'intérruption via le socket. Le client s'occupera alors de s'arrêter. De même pour le SIGPIPE du client qui fait de même que SIGINT.

5 Problèmes

Durant la réalisation de ce projet, nous avons dû faire face à plusieurs soucis. Pour gérer plusieurs clients simultanément, nous avons voulu créer un thread par client connecté. Nous avons commencé par mettre cela dans le code du client avant de gérer cela du côté du serveur. Nous nous sommes ensuite rendu compte qu'on n'établit pas de limite de client dans notre boucle, elle générait infiniment des threads donc. Après avoir géré la connexion de plusieurs clients, nous avons pu réussir tous les tests. Néanmoins, le test 6 et 19 échouaient une fois sur dix, cela nous a poussé à chercher d'où pouvait venir le problème avant de réaliser qu'il s'agissait d'une erreur dans le test comme expliqué dans le mail reçu par le gérant de ce projet.

6 Conclusion

En conclusion, nous avons réalisé ce projet 2 en implémentant des threads et des sockets. Nous avons dû manipuler des fichiers, gérer plusieurs clients se connectant au socket, accepter la demande de connexion du client, créer des structures permettant de stocker nos variables et pouvoir les utiliser plus facilement et de façon lisible. Nous avons rencontré certains problèmes que nous avons pu régler rapidement sans s'y éterniser. Ce projet nous a permis d'acquérir la matière vue au cours et comprendre ces nouveaux concepts que sont les threads, mutex, sémaphore, socket, etc. Cela nous a poussé à faire plus de recherche sur le sujet pour les comprendre et savoir les manipuler convenablement.