Travail pratique #2 - IFT-2245

Christophe Apollon-Roy (920403)
et
Amélie Lacombe Robillard (20016735)

1er avril 2018

Introduction

Pour ce travail pratique, nous avons dû simuler la communication TCP/IP entre un serveur et des clients dans le but de gérer l'allocation de ressources fictives aux clients par le serveur, en utilisant l'algorithme du banquier afin d'éviter les interblocages. Pour réaliser cela, nous avons, en se basant sur le code fourni, écrit les programmes client et serveur s'échangeant de l'information sous la forme de courts messages par le biais de sockets. Les premières étapes de ce projet ont été de nous familiariser avec les librairies C en charge des threads et des sockets, puis d'analyser le code existant afin de décider où et comment y intégrer les fonctionnalités manquantes. Notre plus grande difficulté s'est révélée être la mise en place de la communication client-serveur, s'agissant de notre première expérience avec les sockets.

Le rapport suivant décrira les problèmes rencontrés et les solutions que nous avons choisis d'implémenter pour les résoudre. La première partie traitera de l'implémentation du client, suivie de celle du serveur et finalement des limitations et améliorations possibles de notre programme.

Implémentation du client

Selon la donnée, le programme client doit exécuter, par l'envoi de courts messages, les tâches suivantes : initialiser le serveur, informer le serveur des usages de ressources requis pour terminer ses tâches, avertir le serveur que lesdites tâches sont terminées puis terminer l'exécution du serveur une fois tous les clients déconnectés. La base de l'implémentation de ces fonctions est assez simple ; le client construit ses messages en générant des *strings* composées de mnémoniques de 3 lettres suivies de chiffres et garde en mémoire sur la pile des tableaux de *int* gardant la trace de son usage des ressources allouées par le serveur. Le client utilise ces tableaux afin d'assurer que les requêtes générées sont toujours valides pour le serveur, et d'assurer la désallocation de toutes les ressources lors de la dernière requête d'un client.

Par contre, l'exécution devient beaucoup plus difficile lorsque la communication par TCP/IP entre en jeu. L'usage des sockets en C étant déjà complexe à la base - une majeure partie du temps fut passée à lire les pages du manuel Linux associées à chacune des fonctions de communication TCP -, simplement avoir une connexion entre le client fût une tâche ardue. En effet, les premières itérations du programme étaient victimes de l'occurrence fréquente d'erreurs *SIGPIPE* signifiant un bris de la communication client-serveur. Comme première solution à ce problème nous avons essayé la fermeture puis la réouverture et reconnexion du socket client à chaque opération *send* et *recv* mais, malgré le fait que cela réglait les erreurs*SIGPIPE*, la quantité massive de *file descriptors* générés devenait rapidement un fardeau immense sur les processeurs plus faibles, causant le gel de l'exécution du programme. Finalement, nous avons décidé de restreindre l'usage d'opérations de reconnexion à deux endroits du programme client : le début de la

section configurant le serveur et le début de la section envoyant au serveur le message d'initialisation du client. Ceci était nécessaire étant donné la fermeture de la connexion du côté serveur après son initialisation.

Étant donné la nature du programme demandé par la donnée du travail, il a fallu gérer les conditions de course générées par le multithreading. En effet, pour éviter la corruption des données utilisées par les threads du programme client, nous avons opté pour l'usage d'un *mutex* par statistique pour éviter que la mise à jour d'une statistique par un thread client empêche un autre client d'ajuster une statistique différente. L'usage de sémaphores aurait été un autre choix viable, mais, dans ce cas-ci, il ne ferait qu'ajouter plus de complexité à l'implémentation sans être plus efficace que les *mutex*.

Nous avons également utilisé un *mutex* pour résoudre le problème de l'envoi des commandes d'initialisation au serveur par un thread client car, malgré l'existence de plusieurs threads clients, ces commandes ne doivent être envoyées qu'une seule fois. L'usage du *mutex* assure donc qu'un seul thread entrera dans cette section du code.

Finalement, nous avons résolu le problème de synchronisation de la fin d'exécution des threads en gardant compte du nombre de threads encore en exécution et du statut du serveur. En effet, bien que chacun des threads aient le même nombre de requêtes à envoyer, à cause des délais de calculs aléatoires et des requêtes mises en attente, certains threads terminent leur exécution plus rapidement que d'autres. Garder compte du nombre de threads en cours d'exécution permet d'assurer l'envoi de la commande de fin d'exécution au serveur une seule fois, et d'attendre la réponse à cette commande pour terminer l'exécution du client.

Implémentation du serveur

Le serveur a pour tâche de recevoir les requêtes clients, déterminer si la requête peut être satisfaite et envoyer la réponse adéquate au client. Afin d'éviter l'interblocage entre les threads clients suite à l'allocation de ressources, le serveur implémente l'algorithme du banquier, lequel n'alloue des ressources à un client que si l'état résultant du système est sécuritaire, signifiant qu'il ne peut mener à un interblocage. Coder en C la logique de cet algorithme a été la partie la plus simple du travail, suivant le pseudo-code vu dans les notes de cours et la documentation disponible sur Wikipédia (source : https ://en.wikipedia.org/wiki/Banker %27s_algorithm).

Pour déterminer la sécurité d'un état, l'algorithme utilise des structures pour garder trace de l'état du système, des ressources allouées à chaque client et de la quantité maximale de ressources qu'un client peut demander. Puisque le nombre de ressources est envoyé au serveur lors de son initialisation, il est possible d'initialiser certaines de ces structures à la taille adéquate. Cependant, le nombre de clients pouvant se connecter au serveur n'étant pas prédéterminé et pouvant varier au cours de l'exécution, nous avons dû avoir recours à des tableaux dynamiques afin d'augmenter la taille du tableau au fil de l'ajout de nouveaux clients. L'implémentation de ces tableaux dynamiques est basée sur le code fourni lors des séances de démonstration (source : http ://www-ens.iro.umontreal.ca/ hamelfre/demos/ift2245/).

Un second problème en lien avec les structures de données du banquier provient de l'assignation d'un index au client. Les données étant stockées dans des structures de type tableaux et matrices, il est nécessaire au serveur de savoir à quel index les données d'un certain client sont stockées afin d'y accéder. La solution la plus simple serait d'utiliser le numéro d'identifiant du client comme index, mais cette solution est peu robuste et peut résulter en des structures inutilement grandes. Nous avons plutôt choisi d'attribuer un index aux clients suivant leur ordre d'initialisation et de garder un tableau des index des clients selon

leur identifiant. Ce tableau permet également de garder la trace des clients ayant terminé leur exécution en retirant leur identifiant lors de la fin de leur connexion.

Si le serveur n'avait été composé que d'un seul thread, l'implémentation aurait alors été complète. Or, le serveur pouvant compter plusieurs threads, chacun servant un client différent, et traitant ainsi simultanément plusieurs requêtes, il est nécessaire de partager les données de l'algorithme du banquier entre tous les threads. Ce partage de données donnant lieu à des conditions de courses, nous avons dû d'abord identifier les sections critiques du code de l'algorithme, tout particulièrement les accès aux données partagées, et ajouter des *mutex* afin d'en rendre l'accès exclusif à un thread à la fois. Un second *mutex* a été utilisé pour protéger l'accès et la mise à jour des données du journal.

Finalement, l'acquisition de *mutex* pouvant devenir une source potentielle d'interblocage entre les threads serveur, nous nous sommes assurés que le code respecte la condition qu'un thread ne peut acquérir qu'un seul *mutex* à la fois, exécute une série d'opérations ne pouvant être mises en attente, puis libère son *mutex* immédiatement après. Il est ainsi impossible que l'acquisition de *mutex* résulte en un interblocage entre deux threads du serveur.

Problème de famine

Si la version finale de notre programme est fonctionnelle et conforme à la donnée du travail, elle comporte cependant certaines limitations. Bien que les conditions de courses et les interblocages soient évités grâce aux *mutex* et à l'algorithme du banquier respectivement, les threads clients peuvent être sujets à la famine. Ce problème survient lorsque, pour une raison quelconque, un processus est négligé par le système pour une durée de temps non-bornée. Ainsi, le processus peut attendre potentiellement indéfiniment sans recevoir les ressources dont il a besoin pour son exécution.

Dans le cas de notre programme, un thread serveur établit une connexion avec un thread client et gère ses requêtes jusqu'à la fin de l'exécution de celui-ci, suivant l'ordre du premier arrivé premier servi (FIFO). Or, si le nombre de threads client est supérieur au nombre de threads serveur, les threads client en surplus doivent attendre qu'un client se termine pour envoyer leurs requêtes. Pour éliminer le risque de famine dans un tel scénario, il faudrait permettre à un thread serveur de desservir simultanément plus d'un thread client. Une solution possible serait d'implémenter un système d'ordonnancement des requêtes.

Conclusion

En conclusion, ce travail a été très formateur et nous a permis d'appliquer à un problème concret les concepts de multithreading, de gestion des conditions de course et d'interblocage tel qu'étudiés en classe. De plus, nous avons appris à gérer un protocole de communication client-serveur via des sockets.

Si ce travail était à refaire, établir plus rigoureusement la logique du protocole de communication entre le client et le serveur et l'ordre d'envoi des requêtes plus tôt dans le projet nous aurait évité de traîner des problèmes de sockets jusqu'à tard dans la progression du projet. Finalement, avoir accès préalablement à davantage de documentation sur les sockets, tout particulièrement sur le fonctionnement des non-blocking sockets, nous aurait sauvé beaucoup de temps passé à comprendre et résoudre les problèmes liés à leur bon fonctionnement.