Travail pratique #2 - IFT-2245

Jérémi Grenier-Berthiaumet et Olivier Lepage-Applin

28 mars 2019

1 Général

La partie la plus longue et ennuyante du TP a été de s'assurer que le serveur soit robuste. Nous pensons que pour les prochaines itérations de ce TP, il serait préférable de ne pas faire en sorte que les étudiants aient à s'occuper de cela. S'assurer que le serveur puisse survivre à des requêtes pleines d'erreurs ou qui ne suivent pas le protocole présenté nous a semblé relativement inutile dans le cadre du cours : il y avait amplement de travail avec le reste. Établir un code de base qui fait fonctionner le protocole tel que présenté dans l'énoncé devrait être suffisant.

On pourrait aussi mentionner que plusieurs heures ont été passées à simplement observer le code afin de bien comprendre sa structure avant de commencer à jouer dans ses entrailles. De plus, l'énoncé semblait ambiguë et le protocole peut-être un peu mal défini, ce qui nous a fait perdre du temps pour réussir à savoir ce qui était attendu de nous en général. La structure du code donnait l'impression qu'on ne devait pas écrire du code à l'extérieur des endroits désignés par les `"TODO: BEGIN"` et `"TODO: END"` : encore une fois du temps a été perdu afin de tenter de respecter ces bornes, avant de se faire confirmer qu'on pouvait en fait déborder. Par exemple, l'initialisation du serveur, du côté client, si on se restraignait aux bornes des `TODO`, nécessitait un *mutex* ainsi qu'un booléen pour s'assurer qu'un seul des *threads* client ne l'effectue. Ainsi, la confirmation qu'on pouvait modifier les `main` a changé notre approche en général (en plus de permettre de correctement `free` certaines structures qui y étaient déclarées, sans avoir à modifier la signature de certaines fonctions pour les passer en paramètre).

Toutefois, comparément au TP1, ce TP a tout de même eu l'avantage de ne pas requiérer autant de lecture de documentation.

Finalement, nous avons désormais appris qu'il est possible d'éviter des `malloc` inutiles en passant en paramètre des pointeurs de variables déclarées localement à des fonctions et ce truc (qui nous aurait été fort utile dans le TP1) a été utilisé extensivement et a fortement facilité le passage du "test `valgrind`".

2 Le Travail Pratique

Certaines macros ont été utilisées pour augmenter la lisibilité et rapidité de changement durant le développement. Pensons par exemple à `PRINT_EXTRACTED` qui s'occupe d'imprimer le contenu d'un *array*.

Aussi, mentionnons qu'il a été apprécié de faire en sorte que l'utilisation correcte du `RNG` soit devenu un bonus puisque nous l'avions alors déjà implémenté.

2.1 Sockets

Il a été fortement apprécié d'avoir à notre disposition une fonction '`read_socket`' mise à notre disposition. Cependant, nous avons tout de même eu à écrire nos propres fonctions pour s'occuper de l'écriture dans les sockets (l'envoie d'information). Ainsi, dans `common.c` nous avons ajouté `send_header` et `send_args`.

De plus, puisque le serveur pouvait envoyé des messages d'erreur (i.e. autre chose que des entiers), nous avons aussi ajouté dans `server_thread.c` les fonctions `send_msg` et `send_err`. De façon générale, on

utilise plutôt `send_err` qui nous facilite la vie puisqu'on a alors qu'à passer une string (donc une chaîne de caractères terminée par un `NUL`).

2.2 Mutex

Nous avons implémenté un *mutex* par "variable de statistique" (les compteurs affichés à la fin de l'exécution) afin de permettre une exécution plus indépendante des différents *threads*. Si un unique *mutex* avait été utilisé, le tout aurait tout de même fonctionné, mais le parallélisme aurait grandement été affecté négativement.

2.3 Protocole

On a utilisé `client_thread->id` plutôt que le *pt_ tid* pour les communications (le champ *tid* des requêtes `INIT`, `REQ` et `CLO` des clients). Le fait que dans le cadre du TP cet *id* soit monotone croissant nous a semblé suffisant pour justifier une telle approche et cela rendait le débuggage plus simple à effectuer.

Aussi, on a conservé la fonction qui ne reçoit que le header en premier puisque cela nous était fort utile pour déterminer la longueur des arguments qui étaient à venir (ce qui est, en fait, essentiel puisque cela nous permet de connaître la grosseur du buffer à remplir par la fonction qui lit le contenu du socket).

2.4 Algorithme du banquier

Puisque l'algorithme est sensible aux changements des données associées aux clients, on ne pouvait pas se permettre de changer certaines variables durant son exécution. Pour cette raison, on a utilisé un `mutex_lock` sur la variable `nb_registered_clients` avant de commencer l'algorithme pour ensuite le déverrouiller après l'exécution de l'algorithme. Cela ne fonctionnerait comme prévu que si nous utilisions aussi ce *mutex* lors de l'enregistrement (`INIT`) et le désenregistrement (*CLO*) des clients, ce qui a donc bien sûr été fait (on protège ainsi notre liste de clients, i.e. `clients_list`, de façon générale).

La fonction `bankAlgo` est donc celle qui s'occupe d'exécuter l'algorithme. Les étapes décrites dans les notes de cours sont mises en évidence de façon approximative avec des commentaires du type `/* (Step 2) */`.

De plus, dans le terminal qui s'occupe de rouler le serveur, nous nous sommes arranger pour que l'envoie d'une commande `WAIT` ressorte visuellement parmi toutes les requêtes qui se déroulent rapidement devant vos yeux. En effet, vous allez pouvoir observer que la colonne de texte est généralement de la même largeur, sauf lorsque l'algorithme renvoie un `WAIT` : ce sont ces *printf* de débuggage qui ressortent de temps en temps.

Finalement, il peut sembler que le tout s'exécute de façon séquentielle, mais en observant le début de l'exécution, on peut voir que ce n'est pas le cas : le parallélisme est bel et bien présent. C'est simplement qu'il y a de bonnes probabilités qu'un client soit "bloqué" relativement rapidement par une requête de sa part qui aurait été un peu trop "gourmande".

2.5 Côté client

Premièrement, tel que mentionné dans l'introduction, nous avons modifié le `main` afin de permettre au *thread* principal de se comporter comme un client unique qui s'occupe de gérer l'initialisation du serveur avec les requêtes `BEGIN` et `CONF` dans la méthode "`ct_init_server`". La booléen retourné correspond au succès ou non de cette initialisation et est utilisé pour gérer correctement les erreurs. En effet, si l'initialisation ne s'est pas bien passée, alors les différents `client_thread` ne seront même pas initialisés et la procédure se termine.

Deuxièmement, un sémaphore a été utilisé afin de s'assurer d'éviter du *busy wait* sur la variable `count` lorsque le *main thread* arrive dans la méthode '`ct_wait_server`' après avoir instancié tous les `client_thread` puisque cette instanciation utilise le *flag* `PTHREAD_CREATE_DETACHED` et on ne pouvait donc pas tout simplement utiliser un *join*. Ainsi, dans '`ct_init_server`' on s'occupe d'instancier le sémaphore

et de l'initialiser à 0. Nous aurions préféré l'initialiser à une valeur négative pour nous faciliter la tâche mais il semblerait que UNIX ne permette pas de faire cela. De plus, à chaque fois qu'un `client_thread` envoie sa requête `CLO` au serveur, il entre dans la méthode '`terminate_client`' qui s'occupe de faire un `sem_post`. De cette manière, on utilise ce signal dans une boucle `while` pour vérifier si le dernier client s'était bel et bien déconnecté.

Troisièmement, la signature de la fonction `send_request` a été modifiée pour avoir accès au *struct* 'client_thread'. Cela n'était pas nécessaire, mais rendait le tout plus "*clean*".

2.6 Côté serveur

Une fois de plus, nous avons modifié le `main` afin de pouvoir gérer les cas d'erreur lors de l'établissement de la connection.

Aussi, tel que recommandé par un des démonstrateurs, nous avons utilisé un *array* de fonction pour gérer de façon un peu plus élégante les différents comportements du serveur. Cependant, nous n'avons fait cela que pour une des phases du serveur, i.e. une fois qu'il est initialisé (donc que lorsqu'il s'attends à n'obtenir que des requêtes du type `REQ`/`CLO`/`END`).

La variable `nb_registered_clients` est utilisée pour attendre que tous les clients fassent leur `CLO` lors d'un appel à `END`. Nous utilisons le *busy waiting* puisque contrairement au côté client un *thread* n'atteint cet état que lorsqu'un `END` est reçu (au lieu de dès l'initialisation) : la répercussion est donc, en moyenne, moins grande. Cette boucle se justifie en terme de robustesse pour si jamais un groupe de clients se connectent, mais un signifie qu'il veut fermer le serveur avant que les autres n'aient réellement terminés d'envoyer toutes leurs requêtes avant de se déconnecter (ce qui n'arriverait en fait pas avec *notre* client).

Finalement, les variables de statistique en lien avec `REQ` sont légèrement ouvertes à l'interprétation. Pour notre part, par exemple, s'il y avait un nombre négatif qui indiquait le nombre d'arguments d'une requête `REQ`, celle-ci n'est pas réellement comptée. Cependant, si c'est un nombre d'argument positif mais qui ne correspond pas à `num_resources + 1`, alors celle-ci est comptée comme une erreur.

3 Varia

Cette partie du rapport concerne les points que les démonstrateurs nous ont dit de spécifiquement mentionner dans le rapport ou sinon un simple rappel de ce qu'ils nous ont confirmés qui serait *okay*.

Par rapport à la variable `request_id` utilisée par les clients pour envoyer un certain nombre de requêtes fixé, nous avons décidé que le renvoie d'une requête causé par la réception d'un `WAIT` n'incrémenterait pas cette valeur.

Aussi, les démonstrateurs nous ont dit qu'il était correcte de simplement `exit` et de ne pas tenter de `free` lors d'une erreur sur `malloc`.

Un démonstrateur a aussi mentionné qu'il ne serait pas nécessaire de faire la vérification sur la variable de retour du `close(socket_fd)`.

Nous ne considérons pas que la requête `CLO` fait partie du nombre de requêtes que devrait envoyer un client.

Certaines sections du code ont été enlevées (par exemple, la fonction `st_signal`).