Key-Value Store: Project 2018

Andrea Rar, Ryan Siow, Jonas Epper

IN.4022 Operating System Course, University of Fribourg
andrea.rar@unifr.ch, ryan.siow@unifr.ch, jonas.epper@unifr.ch

Table des matières

1 Introduction

Ce projet a pour but d'implémenter un stockage clé-valeur avec des accès simultanés via TCP. Ce type de donnée est appelé Key-Value Store (KV Store). Le stockage utilise un tableau associatif comme modèle. Dans ce dernier, les données sont représentées comme une collection de pairs tel que chaque clé est unique dans la liste. L'utilisation de sémaphore ou de mutex étudiée en cours permet de pallier au problème de "race condition".

2 Description du problème

Plusieurs problèmes doivent être traités durant ce projet. Ce dernier doit permettre d'écrire les clés avec les valeurs via TCP sockets. Lors du processus de lecture, il doit être possible de lire les valeurs via TCP sockets en fournissant la clé (un message d'erreur s'affiche si la clé entrée n'existe pas). Le programme doit utiliser des threads multiples et un accès simultané doit être garantit aux "readers" et "writers". Enfin, des scripts permettant des tests automatiques doivent être fournis afin de faciliter la tâche de révision du code.
Durant ce projet, diverses décisions d'implémentation ont dû être prises pour rendre la solution unique et proche de l'efficacité optimale.
La première étape consiste à créer une structure permettant de recevoir des paires de clés et de valeurs. Cette dernière doit être dynamique, c'est-à-dire grandir à mesure que des paires clé-valeurs sont ajoutées dans le but d'utiliser le moins de mémoire possible. En outre, cette structure doit supporter des actions telles qu'ajouter, modifier, lire ou supprimer des paires.
La deuxième étape se concentre sur la gestion de la communication entre le serveur et ses clients. Le serveur doit être transparent pour les clients, et ceux-ci doivent pouvoir envoyer des requêtes simples au serveur qui se charge de retourner les résultats.
Finalement, les accès à la structure de donnée doivent être sécurisés afin qu'aucun conflit ne surgisse. Par exemple, il est interdit que deux clients puissent modifier en même temps la même pair, ceci pouvant générer des comportements inattendus et imprévisibles.

3 Solutions et décisions prises

Dans cette partie, les solutions élaborées pour résoudre chaque problème mentionné auparavant sont décrites.

3.1 KV store: array dynamique de struct

Pour ce projet, nous avons opté pour la construction d'un tableau redimensionnable. Les valeurs ainsi que les clés sont ajoutées les unes après les autres dans le tableau. Lorsque les éléments atteignent la taille maximale du tableau qui lui est initialement attribuée, le tableau est redimensionné à deux fois sa taille initiale, et ainsi de suite.
Les fonctions pouvant manipuler le tableau permettent d'ajouter, supprimer, modifier et lire les éléments de ce dernier. Lorsque un élément est supprimé du tableau, il laisse un trou qui peut être complété par la prochaine valeur ajoutée, au lieu que cette dernière s'ajoute à la fin du tableau. Cela permet d'optimiser l'espace (trou complété) en allouant le moins possible de mémoire.
Voici la structure du tableau:

```
typedef struct{
    int key;
    char *value;
    size_t used; //indicate where we are in the array
    size_t size; //indicate the size of the array
}KVstore;
```

Ce tableau dynamique comporte 3 fonctions permettant d'initialiser, ajouter des éléments ainsi que de libérer la mémoire allouée.
Bien que ces fonctions méritent d'être étudiées plus en détail, seule une description de "insertKV()" est donnée. En effet, cette fonction est la plus intéressante car c'est dans celle-ci que l'optimisation de l'espace alloué est réalisée.
Tout d'abord, la fonction redimensionne le tableau si celui-ci en a besoin grâce à la méthode realloc(). La valeur "newvalue" alors passée en argument, si elle n'est pas nulle, pourra être

insérée dans le tableau, soit dans un espace libre s'il y en a, soit à la fin de la série d'éléments déjà présents. Voici un extrait de code explicite:

```
if (newvalue != NULL) {
    size_t length = strlen(newvalue);
    //check if there are holes, if yes add in the hole
    for(int j = 0; j<=kv->used; j++){
        if(kv[j].key == -1 || kv[j].value == NULL ){
            size_t length1 = strlen(kv[j].value);
            kv->used++;
            //clear the memory in this index
            memset(kv[j].value, 0, length1);
            // insert new value
            strncpy(kv[j].value, newvalue,length);
            kv[j].key = newkey; // indicate key
            break;
        }
        //if no holes, just add at the end of the series
        if(j == kv->used){
            kv->used++;
            strncpy(kv[j].value, newvalue,length);
            kv[j].key = newkey; // indicate key
        }
    }
}
```

Le tableau redimensionable a été choisit pour sa simplicité d'implémentation. Les tableaux dynamiques bénéficient de nombreux avantages des tableaux, notamment la bonne localisation de la référence et l'utilisation du cache de données, la compacité (faible utilisation de la mémoire) et l'accès aléatoire. Ils ont généralement seulement une petite surcharge supplémentaire fixe pour stocker des informations sur la taille et la capacité. Cela rend les tableaux dynamiques un outil attrayant pour créer des structures de données compatibles avec le cache.
Toutefois, tel qu'il est implémenté dans ce code, ce tableau comporte un inconvénient: il n'est pas de possibilité de réduire la mémoire allouée (la taille du tableau exacte) pour le tableau si les éléments sont supprimés. Les éléments sont ajoutés dans les trous créés par les éléments supprimés afin de pallier au problème et optimiser l'utilisation de la mémoire. Il est aussi déconseillé d'allouer la mémoire à nouveau lorsque la taille du tableau est réduite, car une copie de ce dernier doit être faite à chaque processus de réduction. Des pertes d'éléments peuvent en être la conséquence.
Une meilleure implémentation aurait été une liste liée (linked list), ainsi, le problème de redimensionnement lors d'une suppression d'élément disparaît. Par rapport aux listes chaînées, les tableaux dynamiques ont une indexation plus rapide (temps constant $O(1)$ par rapport au temps linéaire $O(n)$). Cependant, les tableaux dynamiques nécessitent un temps linéaire pour insérer ou supprimer à un emplacement arbitraire, car tous les éléments suivants doivent être déplacés, tandis que les listes chaînées peuvent le faire à un instant constant (temps de recherche + $O(1)$).

3.2 Communication server - clients

La communication entre le serveur et le client a été résolue par une liaison TCP utilisant les sockets [3]. Ceux-ci sont initialisés dans le serveur et utilisent les librairies *sys/socket.h* et *arpa/inet.h*.

```
    // Creating socket
    socketdesc = socket(AF_INET, SOCK_STREAM, 0);
    if (socketdesc == -1)
        perror("Error on Socket");
    // Editting Server socket
    srv.sin_family = AF_INET;
    srv.sin_port = htons(PORT);
    ...
    // et ainsi de suite...
```

```
10      // ensuite, pour accepter un client et etablir une
            connection:
        clsock = accept(socketdesc, (struct sockaddr *) &clt, (
            socklen_t*) & structsize))
```

L'envoi et la réception de messages entre le serveur et ses clients se fait avec de simples commandes *recv* et *send* qui doivent savoir sur quelle liaison recevoir ou envoyer les messages ainsi que le message et sa taille. Étant donné que la fonction *accept* est bloquante, nous avons distribué les différentes connections sur des *pthreads* afin que chacun gèrent la communication avec un seul client et que le serveur ne soit pas bloqué. Un thread a également été ajouté afin de recevoir les entrées du côté du serveur. Ce dernier thread servait essentiellement à tester rapidement les fonctionnalités ainsi que de pouvoir arrêter le serveur.
Du côté du client, celui-ci n'a en principe besoin que de ce connecter au serveur par la commande *connect* et dès qu'il reçoit confirmation du serveur,il peut commencer à envoyer ses requêtes. Le serveur arrive à recevoir des connections de plus de 50 clients. Nous n'avons pas trouvé pertinent d'essayer un nombre supérieur mais sommes persuadés que le serveur arrive à supporter beaucoup de clients.
Une fois que la commande a été traitée par le serveur, ce dernier affiche le résultat dans sa propre fenêtre et de plus envoie la réponse au client. Afin d'envoyer la réponse au client, chaque fonction traitant la commande modifie un string global se trouvant dans le serveur. Ainsi une fois que la commande a été entièrement traitée, le string est envoyé en une seule fois au client afin de maximiser les performances. En effet l'envoie de messages est plus lent que la modification d'un string global et le client reçoit ainsi qu'un seul message contenant l'intégralité de la réponse.

3.3 Lire, écrire et modifier des valeurs

Afin de vérifier qu'une commande reçue par le serveur (qu'elle soit envoyée par le client ou écrite directement dans la fenêtre du serveur) corresponde à la structure décrite dans le fichier README.txt, l'utilisation de regex s'est imposée. Si la commande ne correspond pas au regex, elle est tout simplement rejetée. L'utilisateur est informé que sa requête n'est pas valide et il peut en insérer une autre.

Les fonctions permettant de manipuler le KV Store grâce aux commandes sont les suivantes:

```
1 void addpair(int newkey, char* newvalue);
  void deletepair(int key, char* value);
3 void modifyPair(int key, char* value1, char* value2);
  void readpair(int key, char* value);
5 void printKV();
```

Les trois premières fonctions sont de type "write" et les deux dernières de type "read". Il n'est pas permit d'ajouter, supprimer ou de modifier la même pair par peur d'avoir des comportement imprévisible et ingérable. En revanche, il est tout à fait possible de lire en même temps.

- "addPair" permet d'ajouter une valeur avec une clé spécifique, ou alors d'ajouter une valeur avec une clé générée si celle-ci n'est pas spécifiée.
- "deletePair" permet de supprimer une valeur (ou plusieurs fois les mêmes valeurs identiques) ainsi que la/leurs clé(s), ou alors de supprimer une valeur spécifique si la clé est mentionnée.
- "modifyPair" permet de modifier une valeur (ou plusieurs fois les mêmes valeurs identiques), ou alors de modifier une valeur spécifique si la clé est mentionnée.
- "readpair" permet de lire une entrée dans le Key-Value Store autant bien si elle reçoit une valeur ou une clé. Elle parcourt l'intégralité du tableau à la recherche de l'entrée désirée. Le parcourt se fait en deux parties délimitées par les mutex. En effet, la partie protégée par les mutex doit tenir compte des entrées ajoutées, modifiées et supprimées au même moment.
- "printKV" fonctionne exactement sur le même principe que la fonction readpair à la différence qu'elle affiche l'intégralité du Key-Value Store et non pas qu'une entrée désirée.

3.4 Accès sécurisé des lecteurs/rédacteurs [1]

Lorsqu'un client veut écrire (writer) dans le Key-Value Store, aucun autre client ne peut écrire en même temps. Cependant, des clients voulant lire (reader) les données stockées, peuvent le faire mais ne peuvent cependant pas lire la donnée qui est sur le point d'être insérée, modifiée ou supprimée. Cette dernière est affiché une fois que la modification est terminée. Lorsque plusieurs clients veulent lire en même temps le Key-Value Store, rien de spécial n'est effectué étant donné qu'aucun ne modifie les entrées. Ainsi, un nombre indéfini de clients peut lire le Key-Value Store en même temps.

Afin de réaliser les fonctionnalités cités plus haut, la tableau est divisé en 2 parties principales, la partie qui n'est pas modifiée actuellement et la partie modifiée en ce moment. La deuxième partie est donc protégée par des mutex afin d'éviter d'être lu lorsqu'un client est en train de la modifier. Afin de délimiter cette seconde partie, il est nécessaire de prendre en compte les index dans le tableau où une clé est ajoutée, modifiée ou supprimée. Ainsi, cette seconde partie est protégée par des mutex.

La solution implémentée utilise 4 mutex différents: rmutex, wmutex, readTry et resource. Ceux-ci sont utilisés dans 3 sections: Entry Section, Critical Section (section où les opérations critiques sont réalisées) et Exit Section. Avant d'entrer dans la "Critical Section", chaque client doit d'abord entrer dans l'"Entry Section". Cependant, il peut y avoir qu'un seul client dans l'"Entry Section" à la fois. Ceci est fait pour éviter les race condition (ex: deux readers incrémentent le readcount au même moment, et les deux essayent de bloquer la ressource, provoquant le bloquage d'un reader).

De plus, les mutex sont implémenté de façon à ce que les clients en écriture aient la priorité. Explication: si le reader R1 bloque l'accès au writter W et que R2 arrive, il ne doit pas pouvoir passer devant W sinon ce serait injuste et ce dernier mourra. Ainsi, une contrainte est ajoutée afin qu'aucun writer, une fois ajouté à la queue, ne doive attendre plus qu'absolument nécessaire.

Cette contrainte est réalisée en obligeant chaque reader à bloquer et relacher le semaphore "readtry" individuellement. Le writer au contraire ne doit pas le bloquer individuellement. Seulement le premier writer bloque le "readtry" et ensuite tous les writers ultérieures peuvent simplement accéder à la ressource quand elle est libérée par le writer précédent. Le dernier writer doit relacher le semaphore "readtry" afin d'ouvrir la possibilité aux readrers d'essayer d'écrire.

De l'autre côté, si un reader a bloqué le semaphore "readtry", ceci va indiquer à tous les writers potentiels qu'il y a un reader dans la section d'entrée. Ainsi, le writer va attendre que le reader relâche le "readtry" et ensuite le writer bloquera immédiatement le "readtry".

Le semaphore "resource" peut être bloqué par le reader ou le writer. Ils peuvent le faire uniquement après avoir bloqué le semaphore "readtry", ce qui peut être fait uniquement par l'un d'entre eux à la fois.

"rmutex" et "wmutex" permettent d'éviter les race conditions pour les "reader" et les "writers" pendant qu'ils sont dans la section d'entrée et de sortie.

Problème de cette solution: dans certains rares cas, elle peut résulter en une "starvation" des readers étant donné qu'ils n'ont pas la priorité. Afin de palier à ce problème, il aurait fallu que l'opération de bloquage des ressources partagées se termine toujours après un certain temps donné. Ainsi il aurait été nécessaire de stocker les readers et les writers dans une queue (FIFO) et obliger les semaphore à maintenir cet ordre.

3.5 Scripts de tests

Afin de pouvoir tester les différents accès automatiquement, nous avons ajouté plusieurs scripts *.sh* dans le dossier /scripts. Le fichier README.txt indique comment lancer ces scripts. Ils fonctionnent de la manière suivante: les programmes sont d'abord nettoyés et compilés puis le serveur est lancé dans un nouveau terminal et un client fictif ajoute des valeurs arbitraires au KV Store afin que les lecteurs puissent lire des valeurs. Ensuite, selon le script que l'on a choisit, cela lance des lecteurs ou des rédacteurs dans un ordre précis afin de montrer par exemple que des lecteurs peuvent continuer à lire des parties du KV Store malgré la présence de rédacteurs dans la partie critiques.
Ces scripts s'occupent de lancer des petits scripts qui eux lancent le serveur ou le client. Enfin, les petits scripts envoient des entrés d'utilisateur automatiquement aux clients ce qui enverra des requêtes au serveur.

```
1 ./client << EOF # servant a feed automatiquement les programmes
      lances
  a test6
3 m 1 test7 # les commandes
```

```
d 2
...
EOF # indique la fin des inputs au programme
```

Afin de lancer ces scripts dans une nouvelle fenêtre, il fallait aussi utiliser une commande qui peut varier selon le système d'exploitation (exemple ici pour Linux Ubuntu):

```
gnome-terminal -x sh -c "./scriptserver.sh"
```

l'option -x permet d'exécuter dans un nouveau terminal les commandes indiquées qui suivent. Ici *sh* permet d'exécuter le script qui suit et l'option -c permet d'indiquer que le script se trouve dans un string.

4 Apprentissage

Andrea: N'ayant aucune connaissance concernant les semaphores et les mutex avant ce cours, j'ai du énormément m'informer à ce sujet. Ensuite, il a fallut décider d'une stratégie adéquate au sein de l'équipe pour l'implémentation des accès sécurisés entre readers et writers. Cette partie a causé quelques soucis notamment au niveau de la séparation du tableau en deux parties. Afin de mener à bien l'intégralité du projet il a été important d'avoir une bonne communication au sein de l'équipe et une bonne répartition des tâches. J'ajouterais que le projet est fort intéressant à réaliser et très divertissant.
Jonas: Je n'étais pas du tout à l'aise avec les connections en TCP via des sockets, j'ai donc énormément appris concernant le fonctionnement pour ce genre de connections. Les threads ne sont pas évidents à gérer non plus car il m'était plus difficile de comprendre leur fonctionnement, cela restait très abstrait pour moi au début. Finalement, je m'attendais à ce qu'écrire des scripts en bash soit plus difficile mais cela n'a finalement pas pris autant de temps que prévu. Ayant déjà suivi énormément de cours qui traitaient des matières proches tels que les systèmes concurrents en Erlang, je n'ai pas eu trop de peine non plus à acquérir les différents concepts. Finalement, ce qui rendait parfois ce projet difficile était qu'à 3, nous ne savions parfois pas d'où provenait une erreur et c'était donc difficile de régler certains bugs. Il faut parfois rester simple au début et être sûr que cela fonctionne avant de passer plus loin et implémenter des choses moins importantes ce qui est parfois frustrant car on veut avancer rapidement dans le projet.
Ryan: N'ayant jamais fait de tableau dynamique auparavant, j'ai rencontré plusieurs problèmes lors de la construction de la structure ainsi que lors de l'implémentation de la dynamicité du tableau; spécialement lors de la suppression d'un élément, laissant ainsi un trou dans le tableau. Néanmoins je crois avoir accumulé une certaine expérience non négligeable dans ce domaine. En outre, il est difficile de parfois comprendre ce qu'un autre membre du groupe a codé, cela implique d'avoir une structure de travail qu'il faut suivre prudemment.

5 Conclusion

La solution présentée ainsi que le code répondent à la consigne, c'est-à-dire aux 5 fonctionnalités minimales. En outre, les décisions d'implémentation concernant la priorité des "readers"/"writters", de la structure de donnée ainsi que des accès simultanés semblent efficaces et optimisés dans leur forme, bien que ces solutions ne soient des plus idéales.

References

[1] *Mutex.* https://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem

[2] *Faire un tableau dynamique en C.* https://stackoverflow.com/questions/3536153/c-dynamically-growing-array?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa, Last visited: 16.04.2018

[3] *Liaison TCP via des sockets en C.* Silver Moon https://www.binarytides.com/server-client-example-c-sockets-linux/, Last visited: 16.04.2018

A Manuel d'utilisateur

Le fichier README explique de manière détaillée comment compiler et exécuter le programme.