NF16 - TP3 - Blockchain

Natan DANOUS & *Léo-Gad JOURNEL* - GI02 à l'UTC.

Le but de ce TP est d'implémenter une BlockChain simplifiée. Une blockchain permet le stockage d'informations (pas forcément financières) de manière décentralisée. C'est une base de données distribuée. L'intérêt de cette technologie est qu'elle est pratiquement impossible à altérer, en effet la liste des transactions étant chaînée, chaque bloc contient la signature numérique du bloc précédent (et par conséquent si l'on modifie un bloc, la signature des blocs change).

Choix d'architecture

Organisation du projet

Du point de vue de l'organisation du projet, nous avons choisis de séparer celui-ci en trois fichiers.

Le premier étant `blockchain.c` qui contient toutes les fonctions nécessaires au fonctionnement de la blockchain (dans le cadre d'une architecture MVC, ce serait le modèle).

Le second est `interface.c` qui agit comme l'interface entre l'utilisateur et la blockchain. Il contient toutes les fonctions permettant d'utiliser les 10 options demandées (ce fichier agit comme controller et comme vue).

Le dernier est le `main.c` qui initialise la blockchain et utilise les fonctions d'interface pour afficher le menu.

Lors du développement, nous avons ajouté nos tests dans le fichier `tests.c`, la fonction `initTests();` de ce fichier lance l'ensemble des tests.

Structure supplémentaire

Timestamp est une structure agissant comme un block d'une liste chainée :

```
typedef struct Timestamp {
    long timestamp;
    struct Transaction *transactions;
    struct Timestamp *nextTimestamp;

}
```

Il regroupe la liste des transactions pour un *timestamp* (date) donné. Nous avons eu besoin de construire une liste ordonnée des transactions lors de l'importation, cette structure nous a permis d'obtenir une solution d'organisation des transactions flexible.

Aussi, notre fichier d'exportation liste les transactions de la plus récente à la plus ancienne. Lors de l'importation, au meilleur cas, le fichier sera trié de la même manière que notre exportation. Ainsi lors de la lecture ligne par ligne du fichier, on pourra construire notre liste en commençant par les blocs les plus récents pour finir par les plus anciens. Une fois la liste construite, il suffit de lire de manière linéaire la liste pour reconstruire la blockchain, on s'épargne ainsi un tri, car ce dernier est effectué directement lors de la construction de la liste chainée et triée du plus ancien au plus récent (timestamp le plus petit au plus grand).

timestamp : **POSIX timestamp** Il s'agit du nombre de secondes écoulées depuis le 1er janvier 1970 00:00:00 UTC jusqu'à l'événement à dater, hors secondes intercalaires. (source Wikipedia - Heure Unix)

Modifications sur les fonctions

Nous avons modifié le prototype de la fonction `float totalTransactionEtudiantBlock(int idEtu, T_Block * b)`, nous utilisons un pointeur T_Block au lieu d'un passage de structure T_Block.

Fonctions supplémentaires

`blockchain.c`

- `T_Block *getBlock(int id, BlockChain bc);` qui permet de récupérer plus simplement un block en renseignant son `id`. Cette fonction nous est utile dans le fichier interface pour afficher les transactions d'un block spécifique.
- `void liberer();` qui permet de libérer l'espace mémoire occupé par la blockchain. Cette fonction parcourt la blockchain et toutes les transactions et libère celles-ci puis le block correspondant. Elle est appelée dans le `main.c` avec un `atexit()` ce qui permet, en cas d'arrêt normal du programme, de libérer la mémoire.
- `void freeTimestamp(T_Timestamp *timestamp);` de la même manière que `liberer()` cette fonction libère l'espace mémoire utilisé par la liste chainée de Timestamp utilisée lors de l'importation. Elle est appelée dans la fonction `importer(char* fileName)`.
- `int max(int a, int b);` est une fonction utilitaire permettant de retourner le maximum entre 2 entiers.
- `DatePlusDays(struct tm *date, int days);` est une fonction utilitaire permettant de modifier une date en lui ajoutant ou soustrayant un nombre de jours.
- `T_Timestamp *insert(long timestamp, T_Transaction *transaction, T_Timestamp *timestampList);` permet d'insérer un Block de Timestamp dans la liste chainée `timestampList`. Si cette liste est nulle, la liste est créée. Un `T_Timestamp` est une structure sous forme de liste chaînée qui permet de contenir de manière chronologique des transactions.

`interface.c`

Mise à part les fonctions correspondant aux 10 options et la fonction d'affichage du menu, nous avons crée une fonction `void boucle();` qui permet de demander à l'utilisateur si il veut quitter ou non à la fin de chacune des options. On réaffiche le menu si il souhaite continuer.

`tests.c`

Ce fichier contient l'ensemble des tests qui assurent le bon comportement de la Blockchain. Seules les fonctions de `blockchain.c` ont été testées de cette manière. Il est possible de lancer les tests avec la fonction `initTests();`.

Complexité des fonctions

Nous traiterons uniquement les fonctions relatives à la blockchain et non les fonctions d'interface. Les fonctions de tests n'ayant une utilité que lors du développement, leur complexité ne sera pas abordée non plus.

ajouterTransaction()

La fonction effectue des affectations simples. La complexité est donc : **O(1)**.

ajouterBlock()

La fonction effectue des affectations simples. La complexité est donc : **O(1)**.

totalTransactionEtudiantBlock()

Pour obtenir le total des transactions sur un Bloc, il faut parcourir les *n* transactions de ce bloc (boucle while), on a donc une complexité **O(n)**.

soldeEtudiant()

La fonction de solde doit dans tous les cas, consulter toutes les transactions de la Blockchain, pour se faire, il faut parcourir les *n* blocs de la Blockchain pour lire les *m* transactions qu'ils contiennent chacun (on prend m = le nombre maximum de transactions contenues dans un bloc). La double boucle `while` nous donne donc une complexité **O(n*m)**.

crediter()

Cette fonction présente une complexité constante : **O(1)**.

payer()

La fonction payer contient des instructions à coût constant, en revanche, elle fait appel à la fonction `soldeEtudiant()` de manière permanente et qui a une complexité de **O(n*m)**, sa complexité est donc **O(n*m)**.

consulter()

La fonction consulter doit rechercher un nombre de transactions pour un étudiant. Elle doit donc au pire cas parcourir les *m* transactions de chacun des *n* blocs de la Blockchain si l'étudiant n'a que 4 transactions ou s'il a exactement 5 transactions et que la première transaction de la blockchain était la sienne. La complexité de de cette fonction donc **O(n*m)**.

transfert()

La fonction de transfert d'argent tente d'abord de retirer l'argent du compte débiteur, pour cela elle fait appelle à la fonction `payer` qui a une complexité de **O(n*m)**, le reste des instructions ont un coût constant. La complexité est donc de **O(n*m)**.

afficherTransaction()

Cette fonction d'affichage présente une complexité constante : **O(1)**.

getBlock()

Pour obtenir le bloc dans la Blockchain constituée de *n* blocs (liste chainée), au pire cas, le bloc se situe en dernier, la complexité au pire cas est alors de **O(n)**.

exporter()

Pour exporter les transactions, on doit parcourir les *m* blocs, et pour chacun de ces blocs on parcourt les *n* transactions. On écrit dans le fichier chaque transaction ligne par ligne. Par conséquent, on a **O(n*m)**.

insert()

L'ajout d'une transaction à son timestamp dans une liste contenant *n* `Timestamp` triés de la plus ancien au la plus récent devra au pire cas parcourir les *n* `Timestamps` transactions (cas où la transaction est la plus récente de la liste), on a donc une complexité **O(n)**.

importer()

Pour un fichier faisant *n* lignes, on doit lire itérativement chaque ligne. Chaque ligne correspondant à une transaction, elles sont ajoutées à une liste chaînée et triée de transactions appelée `timestampList` à l'aide de la fonction `insert` qui présente une complexité de O(n). Après cela, on devra reparcourir chaque transaction pour l'ajouter à la blockchain. On a donc une complexité au pire cas (cas où le fichier ne serait pas trié et qu'il contiendrait une transaction par date) de O(2n*n) ~ **O(n²)**.

void liberer()

`liberer()` Parcours les *n* blocs de la liste et pour chacun de ces blocs, on parcours les *m* transactions qu'il contient. Au pire cas, on *m* maximum dans tous les blocs (tous les blocs contiennent le même nombre de transactions). Par conséquent, on a **O(n*m)**.

freeTimestamp()

Cette fonction ne s'occupe que de libérer l'espace mémoire des `Timestamp` et non de leurs transactions. Par conséquent pour une liste chainée de *n* `Timestamp` la complexité sera de **O(n)**.

DatePlusDays()

Cette fonction présente une complexité constante : **O(1)**.

max()

Cette fonction présente une complexité constante : **O(1)**.