Version 1.0.0 2021/04/06

Projet S4 - Crypto monnaie

Rapport 2ème soutenance

Jules Lefebvre	<jules.lefebvre@epita.fr>	(chef de projet)
Jeanne Cai	<jeanne.cai@epita.fr>	
Baptiste Bucamp	<baptiste.bucamp@epita.fr>	
Vincent Barbier	<vincent1.barbier@epita.fr>	

Groupe: Stonks Industry

Table des matières

1. Introduction

Durant ce cours laps de temps entre les deux soutenances, nous avons pu progresser à bon rythme dans notre projet. Cependant, nous avons toujours une grande difficulté à faire avancer le projet comme indiqué dans le cahier des charges. Nous rencontrons pas mal de problèmes surtout liés à l'utilisation du langage de programmation C. Nous nous efforçons malgré tout d'avancer dans le projet et de le rendre fiable au possible. Nous prenons donc beaucoup de temps sur l'aspect technique et le fond du projet, plutôt que sur la forme de ce dernier.

1.1. Présentation du groupe

1.1.1. Jules Lefebvre

Passionné depuis tout petit par l'informatique. J'ai découvert ce domaine à l'âge de 11 ans lors d'une colonie de vacances. J'ai continué avec les cours du site "du Zero" (actuellement Openclassrooms) où j'ai appris le C++ ainsi que tous les langages web. Je me suis spécialisé dans le développement web. Compétences que j'utilise tout au long de ma scolarité à travers différents projets comme le TPE, le projet de science de l'ingénieur ou le projet d'informatique et science du numérique. J'étais très motivé à l'idée d'intégrer EPITA, je m'y suis directement plu. J'ai pu vraiment m'améliorer pendant les travaux pratiques tout en m'amusant à chercher les optimisations. J'avais déjà essayé d'implémenter un OCR en Javascript mais j'avais abandonné par manque de motivation.

1.1.2. Baptiste Bucamp

À mon arrivée en SPE j'avais un niveau moyen en programmation et des bases solides dans différents langages tels que le Python, le C# et le bash que j'avais acquises durant mes années de lycée avec OpenClassroom et durant mon année de SUP. J'ai également déjà eu des expériences de travail en groupe à moyen terme comme le TPE et un projet d'ISN en terminale qui était un jeu en Python. Ainsi que mon projet de S2 que j'ai fait avec Vincent et qui était un jeu réalisé en C# et avec l'aide d'Unity.
J'ai ainsi pu me familiariser avec le travail de groupe et comprendre que cela me plaisait beaucoup plus que de travailler seul sur un projet. Ce projet devant être réalisé principalement en C, je vais pouvoir mettre en pratique tout ce que j'avais appris jusqu'ici à Epita lors du projet de l'OCR ou durant les TP de programmation réalisés tout au long de l'année.

1.1.3. Vincent Barbier

Intéressé par l'informatique et le développement depuis plus de quatre ans, jusqu'à l'arrivée à l'EPITA je travaillais seul de mon côté, et l'interaction que j'avais avec les autres développeurs consistait surtout à s'aider plutôt que d'avancer ensemble dans un projet commun. Arrivé à l'EPITA en septembre 2019, j'ai bien plus appris en ces différentes années, que dans tout le reste de ma scolarité. Depuis EPITA, j'ai découvert de nombreux projets qui nous suivent tout au long de l'année, notamment le projet de S2 en C# qui fut le premier gros projet en groupe. Suivi du projet d'OCR qui fut mon premier projet avec un aspect vraiment technique. J'ai tendance à préférer les aspects plus techniques et algorithmiques des projets, ce projet de crypto monnaie me correspond donc très bien.

1.1.4. Jeanne Cai

À partir du lycée, j'ai commencé à m'intéresser à l'informatique suite aux cours de mathématiques qui m'ont fait découvert l'algorithmie ainsi que la programmation. J'ai alors décidé de prendre la spécialité ISN en Terminale et j'ai adoré pendant toute l'année à coder des mini-jeux en Javascool. Après cette période, j'ai voulu en apprendre beaucoup plus à coder différents langages et en rejoignant Epita, j'ai de plus en plus aimé la programmation. Les projets de S2 et de S3 m'ont appris énormément de notion sur le travail personnel, mais particulièrement en équipe. J'ai apprécié les concepts de créations surtout comprendre les réalisations ainsi pouvoir peut-être réaliser moi-même de nouveaux. Je poursuis avec le projet de S4 sur les cryptomonnaies, un thème qui demande des recherches et qui apporte des nouvelles notions, c'est ce qui m'a séduit.

1.2. Qu'est ce qu'une crypto-monnaie ?

Une crypto monnaie est un système bancaire décentralisé sécurisé par tous ses utilisateurs. Notre crypto monnaie se base sur une Blockchain et un réseau Pair-à-pair pour fonctionner.

Le réseau est constitué d'utilisateurs qui jouent un rôle de "minage". Le terme "miner" se réfère à l'ajout de nouveaux blocs dans la blockchain, cela permet d'enregistrer de nouvelles transactions. Lors de la création d'un nouveau bloc, le mineur ajoute dans la liste de transactions une rémunération d'un montant fixe pour se récompenser du travail qu'il a effectué. Dans ce contexte, le minage est l'unique moyen de créer de la monnaie. Pour éviter que le nombre de devises augmente, la récompense suit une loi géométrique.

Soit $C \in \mathbb{R}$

$$\int_1^{+\infty} x^{-2} dx = 1 \tag{1}$$

La sécurité des transactions est assurée par la cryptographie asymétrique.
Chaque utilisateur possède un portefeuille (appelé Wallet). Ce portefeuille est constitué d'une clé privée. Toutes les transactions sont signées grâce à la clé privée des utilisateurs. Quant à la clé publique, elle est librement partagée sur le réseau et permet de vérifier que les transactions sont bien valides.

Un bloc est constitué de transactions ainsi que du hash du précédent bloc. Pour éviter que la création de bloc soit trop rapide, un mineur doit montrer une preuve de travail. Cette preuve de travail consiste à résoudre un problème qui demande beaucoup de calcul et de temps. Une preuve de travail pourrait être trouvée un nombre que l'on rajoute aux données du bloc de telle sorte à ce que le hash de celui-ci commence par 4 zéros. La preuve de travail doit long à trouver, mais rapide à vérifier.

1.3. Pourquoi réaliser une crypto-monnaie ?

Comme vous pouvez peut-être le savoir ces derniers temps, le monde des crypto-monnaies évolue à très grande vitesse. De part les événements internationaux qui se succèdent et influencent le cours des crypto-monnaies, où des traders et investisseurs gagnent de l'argent (et bien sûr en perdent aussi). Mais sans s'attarder sur l'aspect financier qui tourne autour de ces nouvelles monnaies virtuelles, nous nous sommes penchés sur l'aspect technique et informatique qui se trouve derrière. L'idée d'une crypto-monnaie n'est donc pas là que par hasard, certes cela à première vue est très abstraite, mais avec quelques recherches nous en avons appris de plus en plus.

1.4. Les aspects algorithmiques

Dans la réalisation d'une crypto-monnaie, nous avons plusieurs aspects algorithmiques à réaliser (qui sont détaillés dans les pages suivantes):

- Une hash table
- Crypto asymétrique
- Queue
- Blockchain
- Graphe

---

2. Ce qui est fait

Étant donné que nous créons un produit qui demande une certaine rigueur pour assurer une stabilité et une bonne sécurité. Nous avons préféré créer des modules de test unitaire et une structure de projet fiable ce qui nous a retardées. Par conséquent, pour l'ensemble des modules, nous ne nous permettons pas de laisser quelques rares cas d'erreurs. Ainsi nous perdons du temps sur des détails, mais notre code est bien plus fiable.

Pour cette raison, toutes nos fonctions possèdent des codes d'erreur permettant de gérer leurs apparitions et de les traiter sans faire sauter tout le programme. En effet, cela pourrait devenir très agaçant pour les clients et d'autant plus problématique pour les serveurs.

2.1. Table de Hachage

Une fois lancer dans la conception de la table de hashage, il a fallu l'implémenter correctement, cette fois-ci le TP de s3 n'a pas servi à grand chose si ce n'est pour les structures ainsi que pour le constructeur.

Les deux structures principales de la table de hachage :

```
1  // A pair is something where we save some informations about an element and his hash
   keys
2  struct s_pair
3  {
4      Buffer key;          // The key
5      Buffer hkey;         // The resulting of the hash of key
6      void *value;
7      struct s_pair *next;
8  };
9
10 typedef struct s_pair *_Pair;
11
12 // Our struct of the hash table
13 struct s_htab
14 {
15     size_t capacity;  // the current capacity of the hash table
16     size_t size;      // number of pairs in the hash table
17     Pair data;        // a pointer on the beginning of the array of pairs
18 };
19
20 typedef struct s_htab *Htab;
```

L'implémentation du constructeur fut relativement simple. Quant à celui du destructeur il a fallu s'occuper de tout les cas possibles comme par exemple le besoin de détruire l'ensemble des "_Pair" qui constitue notre table de hachage.

```
1  int htab_constructor(Htab *new_htab);
2  void htab_destructor(Htab htab, Destructor destructor);
```

Pour le destructeur, la fonction prend un paramètre de type `Destructor` *qui est une fonction qui permet de détruire la value d'une Pair.*

Une fois ces deux fonctions réalisées nous avons dû nous occuper de l'implémentation d'une fonction pour inserrer des clés (`Buffer key`) dans notre table de hashage. Car l'insertion de clés est un peu particulière, il faut d'abord "hacher" la clé pour ensuite l'insérer à la place correspondante. Cette clé et son hachage correspondant son stocker dans une structure `Pair` avec une valeur "choisit".

Cependant une table de hachage ne doit jamais être "remplie", dans notre cas dés que la `size` de notre table de hachage dépasse 80% de la capacité courante de la table, nous doublons la capacité de la table. Cela demande donc de replacer toutes les anciennes Pairs à leur nouvelle place (toujours en fonction de leur clé de hashage correspondante).

```
1   int htab_insert(Htab htab, Buffer key, void *value);
```

Maintenant que nous avons une table de hachage et que l'on peut y ajouter des Pairs, il faudrait que l'on puisse l'utiliser cette table.

```
1   int htab_get(Htab htab, Buffer key, void **value);
2   int htab_remove(Htab htab, Buffer key, Destructor destructor);
3   int htab_pop(Htab htab, Buffer key, void **value, Destructor destructor);
```

C'est pour cela que l'on a dû implémenter ces 3 fonctions, la première permet de récupérer la `value` associé à la clé si elle est présente dans notre table de hachage. Forcément nous devons recalculer la clé de hashage correspondant à la clé de base, pour ensuite voir si notre clé de hachage est présente (au bon endroit) dans notre table. Biensûr nous devons vérifier qu'il n'y a pas deux clés différentes qui ont la même clé de hachage. Une fois trouver on peut récupérer la `value` associée.

Pour les 2 autres fonctions montrées au-dessus, elles servent à enlever d'une façon ou d'une autre une Pair de notre table de hachage. Forcément comme lors de l'ajout, on a un pourcentage minimum à ne pas dépasser lors du retrait. Ce pourcentage est de 20, donc lorsque l'on est inférieur a 20% de la capacité, nous devons diviser la table de hachage par 2 et ainsi recalculé l'emplacement de l'ensemble des Pairs restantes. *Au passage lors de ces fonctions de retrait, nous utilisons en paramètre un `Destructor` afin de détruire la value.*

Comme pour les autres parties une suite de tests a du être créée afin de rendre notre table de hachage la plus fiable possible.

De plus à des fins purement visuel, nous avons créé une fonction d'affichage d'une table de hachage et de ce qui la compose.

2.2. Big int

Le but de ce type est de stocker et de manipuler des nombres de tailles variables parfois bien supérieures au type de base du C. Il contiendra par exemple les clés privées et publiques du client. De ce fait, il est un dérivé des buffers et bénéficiera au module de cryptographie asynchrone.

Comme les types de base, il possède le bit de poids faible à gauche ce qui nous permet de l'utiliser et de convertir très facilement les opérations de base du C. Ainsi cela nous permet d'utiliser les opérations de base du C et de profiter de leur vitesse.

Grâce à notre suite de tests, nous avons pu très rapidement réimplémenter les algorithmes en fonction de la place du bit de poids faible, cela nous a aussi permis de rester cohérents et d'être sur de toutes nos fonctions.

Comme les `BigInt` sont de taille variables nous ne pouvons pas utiliser le codage avec complément à 2 pour les négatifs. L'addition ne marcherait pas à cause du complément à 2 qui change en fonction de la taille du `BigInt` .C'est pour cela que nous avons pensé à utiliser un attribut "sign" qui représente de signe du bigint.

Les bigints possèdent aussi d'autres propriétés afin d'accélérer les opérations et de réduire leur empreinte memoire.Les bigint sont normaliser, à leur création le constructeur tronque les octets null de poids forts.

valeur décimal	buffer	sign	exhibitor
0	`[]`	POSITIVE	0
1	`[ 0x01 ]`	POSITIVE	1
10	`[ 0x0a ]`	POSITIVE	4
385218	`[ 0xc2, 0xe0, 0x05 ]`	POSITIVE	24
3852181624961359865334	`[ 0xf6, 0xe5, 0xe2, 0x5c, 0x2b, 0x5a, 0xc2, 0xd3, 0xd0 ]`	POSITIVE	72
-1	`[ 0x01 ]`	NEGATIVE	1
-10	`[ 0x0a ]`	NEGATIVE	4
-1000	`[ 0xe8, 0x3 ]`	NEGATIVE	10

Pour cette soutenance nous avons réussi à implémenter :

- des constructeurs :
  1) `bigint_constructor_null` : créer un bigint vide
  2) `bigint_constructor_array` : créer un bigint à partir d'un tableau de valeurs
  3) `bigint_constructor_buffer` : créer un bigint à partir d'un buffer
  4) `bigint_constructor_from_int` : créer un bigint à partir d'un entier signé
  5) `bigint_constructor_bigint` : créer un bigint à partir d'un autre bigint
- un destructeur permettant d'effacer les données du buffer d'un bigint et de libérer l'espace mémoire.
- des getter :
  1) `_bigint_get_buffer_exhibitor`
  2) `_bigint_get_array_exhibitor`
- des opérations de conversions :
  1) `bigint_to_bool` : convertit un bigint en un booléen
  2) `bigint_to_int` : convertit un bigint en un entier signé
  3) `bigint_to_long_long_int` :convertit un bigint en un entier compris entre `-9 223 372 036 854 775 807` et `+9 223 372 036 854 775 807`
  4) `bigint_to_buffer` : convertit un bigint en un buffer
  5) `bigint_to_string` : convertit un bigint en une chaîne de caractères
- des opérations de comparaison entre 2 bigint qui renvoie un booléen:
  1) `bigint_greater_than` : renvoie vrai si le premier bigint est plus grand que le deuxième
  2) `bigint_less_than` : renvoie vrai si le premier bigint est plus petit que le deuxième
  3) `bigint_equal_than` : renvoie vrai si le premier bigint est égal au deuxième
  4) `bigint_not_equal` : renvoie vrai si le premier bigint est différent du deuxième
  5) `bigint_less_or_equal` : renvoie vrai si le premier bigint est inférieur ou égal au deuxième
  6) `bigint_greater_or_equal` : renvoie vrai si le premier bigint est plus grand ou égal au deuxième
- des opérations arithmétiques entre 2 bigint et qui renvoie un nouveau big int:
  1) `bigint_addition` : fait l'addition entre 2 nombres
  2) `bigint_substraction` : fait la soustraction entre 2 nombres
  3) `bigint_shift` : décale tous les bit d'un nombre de bit (decalage positif <=> decalage vers la

gauche)
4) `bigint_shift_left` : décale vers la gauche de n bit
5) `bigint_shift_right` : décale vers la droite de n bit
6) `bigint_multiplication` : fait la multiplication entre 2 bigint

2.3. Blockchain

Une blockchain est une chaîne de blocs qui contient l'historique des transactions entre utilisateurs ainsi les blocs contiennent la liste des transactions.
Le "Genesis block" est le tout premier bloc à la création de la blockchain qui présente les mêmes caractéristiques qu'un sentinel vu en cours de programmation.

- Chaque bloc est numéroté dans l'ordre d'arrivée dans la blockchain : afin de différencier chaque bloc pour la démonstration et les exemples.
- A la création d'un bloc, celui-ci récupère et stocke le hash du précédent bloc dans son bloc.
- Il stocke principalement une liste des transactions effectuées. Pour le moment, les données sont des chaînes de caractères en attendant d'implémenter les transactions dans la blockchain.
- A la fin des stockages de toutes les données du bloc, on doit vérifier et valider le bloc s'il est correct, c'est-à-dire la preuve de travail.
- La preuve de travail a deux paramètres importants : le nonce qui est un entier initialisé à zéro et un autre entier n.

La preuve de travail a pour objectif de chercher un entier qui permettra d'obtenir un hash commençant par n zéros. Si le hash ne correspond pas à l'objectif alors on incrémente le nonce et ainsi de suite.

Cette performance prendra de plus en plus de temps si le n'augmente d'où l'appellation de preuve de travail car il est difficile de trouver le nonce très rapidement.

Les personnes qui cherchent le nonce avec leur machine sont appelées mineurs. Les mineurs cherchent le nonce pour qu'un bloc soit validé et ainsi un butin si la preuve de travail est effectuée. Cette pratique permet d'éviter que plusieurs blocs arrivent simultanément en même temps pour être validés dans la blockchain. Le nonce est un nombre qui est long à chercher surtout car le moindre changement dans les données change totalement le hash.

La validation d'un bloc commence d'une part par voir si le hash commence avec suffisamment de zéros paramétrés par la preuve de travail et d'un autre part si les hash (le hash du bloc et le hash du précédent bloc) corresponde bien dans la blockchain.

Pour cette soutenance, il n'y a pas encore de transaction donc on a des données de type void *.

Les structures d'un bloc et d'une blockchain :

```
1  struct block
2  {
3      Buffer previousHash;
4
5      size_t index;
6      long nonce;
7
8      void *data;
9
10     Buffer hash;
11     struct block *previousBlock;
12 };
```

```
13
14  struct blockchain
15  {
16      struct block *block;
17  };
18
19  typedef struct blockchain *Blockchain;
20  typedef struct block *Block;
```

Voici la liste des implémentations de cette soutenance pour le module “Blockchain” :

- création des structures : la création d'une blockchain et des blocs en allouant de la mémoire et initialiser les données de la blockchain et du bloc.
- ajout d’un bloc dans une blockchain : un bloc n'est pas ajouté à la blockchain s'il n'est pas valide.
- hachage d'un bloc : à l'aide du module `hash`, du module `buffer` et la librairie `strings.h` pour la concaténation des données du bloc hormis le hash du bloc.
- minage : miner à la place des mineurs pour cette soutenance.
- proof of work : la preuve de travail c’est-à-dire la vérification et la validation à la création d’un bloc, mais aussi un parcours de tous les blocs dans la blockchain et vérifier en même temps.
- destruction des structures : une fonction pour libérer la blockchain et des blocs de la mémoire.

Pour les exemples suivants data qui est en type void * sera en char * c'est-à-dire en chaîne de caractère.

1. Un exemple d'un "Genesis block" : (data : void * -> char *)

```
1       [0x5605641602a0] // (1)
2
3       index         : 0,
4       nonce         : 914464,
5       data          : Genesis block,
6       previousHash  : (null),
7       hash          : 000000c5300b24c7, // (2)
8       previousBlock : (nil)
```

(1) : Adresse du bloc
(2) : Adresse du précédent bloc

2. Un exemple d'une blockchain. On ajoute 4 blocs dans cette blockchain avec N_PROOF = 5 avec chacun une donnée (data) différente sauf pour le dernier bloc :

```
1       [0x56057e690aa0]
2
3       index         : 4,
4       nonce         : 2979149,
5       data          : Hello 1,
6       previousHash  : 00000c027b88f624,
7       hash          : 00000c2ae9fc6bb7,
8       previousBlock : 0x56057a9fe2c0
9
10
11      [0x56057a9fe2c0]
12
13      index         : 3,
14      nonce         : 360877,
15      data          : Hello 3,
16      previousHash  : 00000831b709fec8,
17      hash          : 00000c027b88f624,
18      previousBlock : 0x56056d4f20d0
19
```

```
20
21      [0x56056d4f20d0]
22
23      index         : 2,
24      nonce         : 1269296,
25      data          : Hello 2,
26      previousHash  : 0000009fec37e68c,
27      hash          : 00000831b709fec8,
28      previousBlock : 0x56056d3e1980
29
30
31      [0x56056d3e1980]
32
33      index         : 1,
34      nonce         : 6339,
35      data          : Hello 1,
36      previousHash  : 000000c5300b24c7,
37      hash          : 0000009fec37e68c,
38      previousBlock : 0x5605641602a0
39
40
41      [0x5605641602a0]
42
43      index         : 0,
44      nonce         : 914464,
45      data          : Genesis block,
46      previousHash  : (null),
47      hash          : 000000c5300b24c7,
48      previousBlock : (nil)
49
50  Blockchain is correct ! // (3)
51
52  real  0m3.532s
53  user  0m3.251s
54  sys     0m0.280s
```

(3) : Message pour la vérification de la blockchain

- On remarque que chaque bloc possède bien un hash commençant par 5 zéros (= N_PROOF) et que le nonce. Le temps de créer, de miner, de valider les blocs les uns après l'autre a au moins mis 3 secondes, cela représente un long temps pour une machine. D'où le fait que miner prend du temps à trouver le bon nonce pour avoir un hash qui commence par des N_PROOF zéros.
- Le dernier bloc et le premier bloc ajouté ont la même donnée : `"Hello 1"`. Même s'ils ont la même donnée on voit bien que leur hash n'est pas pareil et diffère : `"0000009fec37e68c" (index : 1)` est différent de `"00000c2ae9fc6bb7" (index : 4)`.
- Le `nonce` varie et peut ainsi être un faible nombre mais aussi un grand nombre allant d'en des millions sur cette blockchain.

2.4. Database

La database est une base de données qui va stocker la blockchain dans un fichier. Ici, la database sera en fichier cvs. Depuis ce fichier, il permet aussi de retranscrire le contenu de ce fichier en structure Blockchain.

Pour le moment, la Database nous permet :

- D'initier la blockchain
- De récupérer toutes les informations des fichiers dans le dossier prévu à cet effet.
- D'ajouter les informations récupérées d'un fichier et de les écrire dans un bloc de la blockchain.

Header utilisé :
<dirent.h>: celui-ci nous permet de naviguer dans un dossier plus facilement et d'effectuer un certain nombre d'actions sur des fichiers.

```
opendir : ouvre et renvoie un pointeur sur le flux répertoire à l'aide du chemin du dossier

closedir : ferme le flux du répertoire du dossier

readdir : renvoie un pointeur sur une structure dirent à partir du flux répertoire

structure dirent : permet d'accéder au fichier suivant dans le flux répertoire
```

Fichier en entrée : test.cvs

```
transaction 1, transaction 2, transaction 3, transaction 4;
transaction 5, transaction 6, transaction 7, transaction 8;
transaction 9, transaction 10;
```

Ici dans le fichier test.csv, chaque ligne représentera une donnée à mettre dans un bloc dans la blockchain. La donnée sera stocker comme une chaîne de caractère.

Blockchain en sortie :

```
    [0x5593f3f9c130]

    index         : 3,
    nonce         : 1866034,
    data          : transaction 9, transaction 10,
    previousHash  : 000000a6ce3b50a7,
    hash          : 0000030ca8688451,
    previousBlock : 0x5593dbfa7240


    [0x5593dbfa7240]

    index         : 2,
    nonce         : 1797355,
    data          : transaction 5, transaction 6, transaction 7, transaction 8,
    previousHash  : 000009aaf4c6fff4,
    hash          : 000000a6ce3b50a7,
    previousBlock : 0x5593d3e48d10


    [0x5593d3e48d10]

    index         : 1,
    nonce         : 605590,
    data          : transaction 1, transaction 2, transaction 3, transaction 4,
    previousHash  : 000000c5300b24c7,
    hash          : 000009aaf4c6fff4,
    previousBlock : 0x5593cabbe2a0


    [0x5593cabbe2a0]

    index         : 0,
    nonce         : 914464,
    data          : Genesis block,
    previousHash  : (null),
    hash          : 000000c5300b24c7,
    previousBlock : (nil)
```

Dans cet exemple, étant donné qu'il y avait 10 transactions dans le fichier la fonction a créée 3 blocs : une pour chaque ligne.

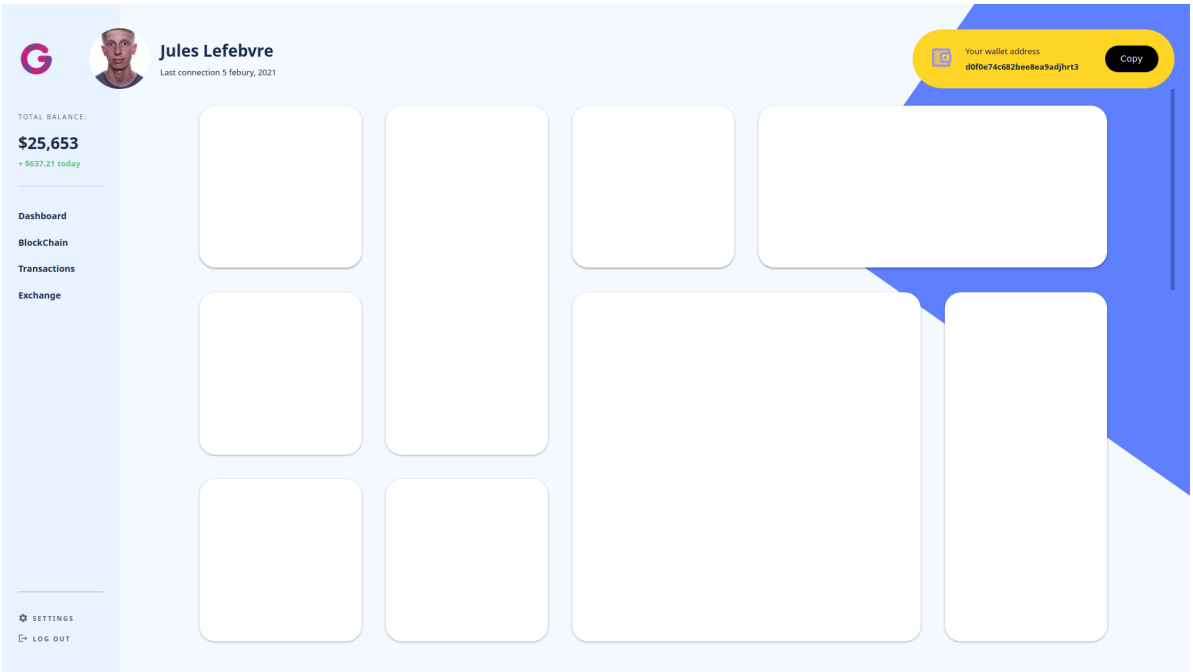
2.5. Transaction

Cette partie étant créer que récemment, nous avons pu implémenter uniquement les structures ainsi que le constructeur et le destructeur.

```
1   struct s_transaction
2   {
3       Buffer from;        // Key public of the expeditor
4       Buffer to;          // Key public of the beneficiary
5       double amount;      // The amount of the transaction (how many cryptos)
6       Buffer signature;   // The signature
7       Buffer date;        // When the transaction is treated
8   };
9
10  typedef struct s_transaction *Transaction;
```

La particularité de ce module est la `signature` qui pour le moment est le résultat du hachage entre l'expéditeur, le destinataire, la clé privée de l'expéditeur et la date de la transaction.

2.6. Site Web



Le but final est d'interfacer le site avec notre crypto monnaie pour suivre en temps réel le cours et les transactions de celle-ci. Le site sera donc une application nomo-page écrite en Javascript.

Pour cette première soutenance, nous avons créé le squelette de notre application en pure Html, Javascript et CSS. Nous avons créé un système de boîtes responsive (qui s'adapte à la taille de l'écran).

Vous pouvez retrouver notre site sur: https://julesdecube.github.io/UnCoin

2.7. Les plannings

2.7.1 Planning de réalisation

En résumé, voici le tableau de ce que l'on avait prévu de faire à la base sur le cahier des charges comparé à ce que l'on a réellement fait.

Tâche\Soutenance	Ce qui était prévu	Ce que l'on a réalisé
Hash table	✔	✔
BigInt	✔	✔
Crypto Asymétrique	✔	
File Database	✔	🏗
Block Chain	✔	✔
Transaction	✔	🏗
Cryptomonnaie	🏗	
Consorsium algo	🏗	
Crypto protocole	🏗	
Pear to pear	🏗	
Crypto client		
Crypto server		
Site web	🏗	🏗

✔ : Tâches finies
🏗 : Tâches en cours de réalisation

2.7.2 Planning cahier des charges mis à jour

Tâche\Soutenance	1ère	2ème	Finale
Buffer	✔		
Queue	✔		
BigInt	🏗	✔	
Hachage	🏗	✔	
Hash table	🏗	✔	
Block Chain		✔	
File Database	🏗	🏗	✔
Transaction		🏗	✔
Cryptomonnaie		🏗	✔
Crypto Asymétrique			✔
Pear to pear			✔
Consorsium algo			!?
Crypto protocole			!?
Crypto client			!?
Crypto server			!?
Site web	🏗	🏗	✔

✔ : Tâches finies
🏗 : Tâches en cours de réalisation
!? : Optionnel

Comme le montre ce tableau, nous avons du retard. Nous avons tout de même réussi à le rattraper en partie mais il est toujours présent. Nous passons dans l'ensemble beaucoup de temps sur la réalisation des tests unitaires afin de créer notre code le plus fiable possible.
Nous avons choisi de passer en "optionnel" certains de nos modules car ils restent assez imposants et nous savons qu'il ne faut pas avoir les yeux plus gros que le ventre.

2.7.3 Répartition des charges pour cette soutenance

Tâche	Vincent	Jeanne	Baptiste	Jules
Hash table	☆			💝
BigInt			💝	☆
File Database		💝	☆	
Block Chain		☆	💝	
Transaction	☆	💝		
Site web			💝	☆

☆: Responsable
💝: Suppléant(e)

3. Avancement du projet : retards et difficultés

3.1. Table de Hashage

Malgré le fait que cette partie était déjà commencée, elle a prit beaucoup de temps à l'un de nos membres, il a d'abord fallu se renseigner sur comment correctement l'implémenté et préréaliser les differents tests. Certes pour le début, le TP de s3 a pu aider, mais finalement nous avons fait beaucoup plus que ce qui est fait dans ce TP.
Une fois les bases créées il a fallu s'occuper de tout ce qui est des éléments à ajouter et recupérer, cependant comme dit précédemment cela ne se fait pas naturellement et suit un procédé précis. De plus, nous nous sommes attardés sur la correction de cas possibles qui serte sont très rare, mais aujourd'hui, ne font pas crash le programme.

3.2. BigInt

L'addition et la soustraction nous on pris beaucoup de temps notamment, notamment à cause du fait que l'on doit crée un buffer assez grand pour éviter les dépassements et du coup avoir un segfault. Pour pouvoir choisir entre l'addition et la soustraction nous devons d'abord savoir lequel des nombre est le plus petit (notamment si l'un des chiffres est négatif). Après cela choisir le signe nous a pose pas mal de problème. Enfin pour optimiser le temps nous avons ajouté des gardes notamment si les chiffres sont égale ou nulle.

Le décalage de bit a aussi été difficile à cause du calcul de la nouvelle taille. De plus le fait que des octets puissent se retrouver entre 2 nouveaux octets nous oblige à garder en mémoire le précédant octet.

La multiplication n'a pas été trop dure étant donné qu'elle reprend le décalage et l'addition. Point notable nous faisons la somme des bit qui sont à 1 pour savoir quel nombre va être parcourue et lequel va être additionné.

3.3. Block Chain

Lors de la soutenance précédente, nous n'avons pas pu commencer cette partie car elle dépend directement de la partie "Transaction" qui dépend de la partie "Crypto Asymétrique" et qui dépend de la partie "BigInt".

Malgré les dépendances, nous avons fini une grande partie sur la Blockchain sans les transactions. Donc, il nous reste à implémenter les transactions dans la structure de blockchain pour la soutenance finale.

Dans cette partie, les difficultés rencontrées étaient la gestion des chaînes de caractères pour le hash qui était assez compliqué, c’est-à-dire correctement allouées à la mémoire car on n'allouait pas suffisamment de mémoire ou trop de mémoire pour stocker des nouvelles chaînes de caractères localement. En effet, le hash demande le changement de type des données dans chaque bloc en chaînes de caractère (donc sauf le hash du bloc) et de les concaténer en un seul chaîne de caractère pour pouvoir hacher celle-ci.

Néanmoins, la structure d'une blockchain en elle-même n'était pas difficile à comprendre et à représenter.

3.4. File Database

Au début de l'implémentation de la database, nous avons d'abord commencé avec le chemin du fichier csv déjà défini. Pour cette partie, nous avons stocké les blocs de la blockchain dans un seul fichier en incluant un bloc par ligne pour avoir un début de database.

Cependant nous nous sommes rendu compte qu'il faudrait plutôt stocker un bloc par fichier et une transaction par ligne pour plus de clarté. Cela ne devrait pas être très dur à changer dans la mesure où l'on accès au hash du bloc précédent sans avoir à le chercher dans le dossier.
C'est ainsi que nous avons commencé à implémenter une fonction pour chercher plusieurs fichiers csv dans un dossier, en oubliant la première option choisie pour trouver chemin d'un fichier.

Pour le moment, les transactions ne peuvent toujours pas être encryptés et donc les blocs de la blockchain ne sont pas totalement complets car il manque les transactions.

Ainsi la récupération des informations d'un bloc de la blockchain pour les sauvegarder dans un fichier n'a pas pu être faite pour le moment.

3.6. Transaction

Pour cette partie nous commençons juste à l'implémenter c'est pour cela qu'elle n'est encore que peu developpée. De plus nous avons besoin d'un autre module qui n'est pas encore fini.

3.5. Crypto Asymétrique

Cette partie n'a pas été commencé car elle nécessite le building type Bigint qui n'a pas encore été terminé.

4. Conclusion

Comme lors de la première soutenance, nous avons du retard, mais heureusement pour nous, nous avons réussi à le réduire ! Comme précédemment nous passons beaucoup de temps sur la création de nos nouveaux modules et l'implémentation de nos types personnalisés. Puisque l'on n'utilise quasiment pas de librairies, et que nous cherchons à créer le code le plus correct possible, nous perdons toujours du temps par rapport au plan initial. Cependant nous approfondissons un maximum nos différends modules et apprenons beaucoup en les réalisant. Sur l'ensemble, nous nous en sortons tout de même bien et ce projet reste enrichissant.