Projet S4 - Crypto monnaie

Rapport 1ère 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>	

Table des matières

1. Introduction

Lors de cette première partie, le projet d'une crypto monnaie a été un peu particulier. Tout d'abord nous avons dû nous pencher bien plus en détails sur chaque aspect de la réalisation du projet, car il faut le dire nous n'y connaissions pas grand-chose jusqu'à présent. En effet nous avons donc pris un certain de temps pour faire des recherches et de l'approfondissement. Nous avons également remarqué que certaines parties étaient bien plus complexes qu'elles n'y parraissaient et cela a donc pas mal retardé l'avancement du projet. Notre projet a eu du mal à démarrer, mais depuis quelques temps nous avançons à bonne allure.

1.1. Présentation du groupe

1.1.1. Jules Lefebvre

Passionné depuis tout petit par l'informatique. J'ai découvert l'informatique 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'implementer un OCR en Javascript mais j'avais abandonné par manque de motivation.

1.1.2. Baptiste Bucamp

A 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

A 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 trouvé 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

Etant 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 beaucoup retardées. Par conséquent, le nombre de modules n'est pas à la hauteur de ce que nous avions prévu dans le planning.

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.

2.0 Les structures utilisées

Pour notre projet, nous avons décidé d'utiliser plusieurs structures afin de pouvoir respecter les standards d'une cryptomonnaie.
Nous avons donc implémenté :

- des utilitaires tels que : des alias sous forme de booléen, des nom des types d'erreurs

```c
typedef int bool;
#define true 1
#define false 0
```

```c
// Task is a success
#define SUCCESS 0
// Error that is not normaly possible
#define INTERNAL_ERROR 1
// Not enough space
#define NO_SPACE 2
// The main object is null
#define NO_SELF 3
// The provided index is greater than the size
#define OUT_OF_RANGE 4
// The value is not valid
```

- une structure pour utiliser des buffers

```c
struct s_buffer
{
    size_t size;
    u_char *data;
};
```

- une structure pour utiliser des files

```c
struct _s_queue_block
{
    void *element;
    struct _s_queue_block *next;
    struct _s_queue_block *previous;
};

struct s_queue
{
    size_t size;
    struct _s_queue_block *first;
    struct _s_queue_block *last;
};
```

- une structure "bigint" qui nous sert à stocker des entiers de très grands tailles

```c
struct s_bigint
{
    size_t exhibitor;
    bool sign;
    Buffer buffer;
};
```

2.1. Buffer

Il est le composant principal de tous nos types de donnée, c'est une version protégée des tableaux en C. Il sont constitués d'un pointeur de donnée et de la taille de celle-ci. Il bénéficie d'un grand nombre de constructeurs nous permettant de l'utiliser dans beaucoup d'autres types comme (`array` , `struct` , `int` , `size_t` , ...).

Il nous offre une façon sécurisée lors de sa destruction afin d'éviter les `Segmemtation fault` et les fuites de données. De plus, il possède un destructeur qui réinitialise toutes les informations du buffer avant sa suppression pour éviter toute attaque par lecture de mémoire post `free` .

Il possède aussi son propre moyen d'affichage en hexadécimal afin de pouvoir le débuger et partager son contenu à l'utilisateur. Une version en base 64 est en cours de test avant d'être finalisé.

2.2. Queue

Queue est une structure que nous appris en cours d'algorithme et que nous utiliserons pour le projet. Cette structure fonctionne comme une file d'attente. Le but est d'enfiler des éléments et de défiler des éléments dans l'ordre d'arrivée des éléments dans la file d'où le "premier arrivé, premier sorti". Ici, elle est implémentée en tant que liste chaînée.

Queue possédera deux structures : la file et le bloc où seront stockés les éléments enfilés. Elles sont nommées respectivement `s_queue` et `_s_queue_block` .

- La structure de la file est composée du nombre d'éléments, un pointeur sur l'élément en tête dans la file et un pointeur sur le dernier élément enfilé dans la file.
- La structure du bloc est composée d'un pointeur sur l'élément, un pointeur sur le précédant bloc depuis le bloc actuel et un pointeur sur le prochain bloc qui suit dans la file depuis le bloc actuel.

La particularité de la structure est que les éléments que nous enfilons dans la file peuvent être de n'importe quel type, elles sont implémentées en type `void *` (nommé "element" dans la structure `_s_queue_block`). Mais aussi, pouvoir effectuer des opérations dans un ordre ordonné (va être déployé dans les prochaines soutenances comme permettre de gérer les transactions dans une unique file partagée entre chaque client).

Il y a ainsi les fonctions implémentées :

- `Queue queue_init()` : crée une file. La fonction alloue la file et initialise les valeurs de la file: les pointeurs à NULL et le nombre d'éléments dans la file à zéro, et renvoie la file.
- `bool queue_is_empty(Queue queue)` : renvoie vrai si la file est vide sinon faux.
- `void queue_enqueue(Queue queue, void *data)` : enfile l'élément "data" dans la file "queue". La fonction alloue le bloc, initialise ses valeurs, réinitialise les valeurs de la file vu que le premier et dernier élément peuvent changer de bloc et incrémente le nombre d'élément.
- `void *queue_dequeue(Queue queue)` : défile l'élément en ordre d'arrivée. La fonction désalloue le bloc et récupère l'élément en réinitialisant les valeurs de la file et en décrémentant le nombre d'éléments.
- `void queue_free(Queue queue, Callback callback)` : appelle une fonction proccess qui appelle la fonction dequeue pour défiler tous les éléments dans la file, et désalloue la file.

L'affichage de la file est sous la forme de :

```
==========================================
Queue: nombre d'elm (add du 1er bloc <> add du dernier bloc)
------------------------------------------
add du 1er bloc   : add du 1er elm   ((nil) <> add du 2e bloc)
add du 2e bloc    : add du 2e  elm   (add du 1er bloc <> add du 3e bloc)
add du 3e bloc    : add du 3e  elm   (add du 2e bloc  <> add du 4e bloc)
...
add du nième bloc : add du nième elm (add du n-1 ième bloc <> (nil))
==========================================
```

- add : addresse
- elm : élément

Voici quelques exemples de tests : (le programme ne prend pas d'argument)
A chaque execution d'une fonction, on affiche la file fonction (cf liste de fonction implémentée, affichage de la file dans Queue).

```
$ make tests_build
gcc ...
$ ./build/tests/queue
================================================
||                                            ||
||                    INIT                    ||
||                                            ||
================================================
[Init q] // Initialise une file nommée "q"
==========================================
Queue: 0 ((nil) <> (nil))
------------------------------------------
==========================================

[Init q1] // Initialise une file nommée "q1"
==========================================
Queue: 0 ((nil) <> (nil))
------------------------------------------
```

```
==========================================

[Init q2] // Initialise une file nommée "q2"
==========================================
Queue: 0 ((nil) <> (nil))
------------------------------------------
==========================================

================================================
||                                            ||
||                   ENQUEUE                  ||
||                                            ||
================================================
[Enqueue q "Hello"] // Enfile une chaîne de caractères en char * dans q
==========================================
Queue: 1 (0x564abb484710 <> 0x564abb484710)
------------------------------------------
0x564abb484710:0x564aba8c720f ((nil) <> (nil))
==========================================

[Enqueue q "1"] // Enfile un chiffre en caractère en char * dans q
==========================================
Queue: 2 (0x564abb484710 <> 0x564abb484730)
------------------------------------------
0x564abb484710:0x564aba8c720f ((nil) <> 0x564abb484730)
0x564abb484730:0x564aba8c7225 (0x564abb484710 <> (nil))
==========================================

[Enqueue q "400000"] // Enfile un nombre en chaîne de caractère en char * dans q
==========================================
Queue: 3 (0x564abb484710 <> 0x564abb484750)
------------------------------------------
0x564abb484710:0x564aba8c720f ((nil) <> 0x564abb484730)
0x564abb484730:0x564aba8c7225 (0x564abb484710 <> 0x564abb484750)
0x564abb484750:0x564aba8c723c (0x564abb484730 <> (nil))
==========================================

[Enqueue q1 "Hellooooooooo"] // Enfile une chaîne de caractère en char * dans une
autre file (q1)
==========================================
Queue: 1 (0x564abb484770 <> 0x564abb484770)
------------------------------------------
0x564abb484770:0x564aba8c725f ((nil) <> (nil))
==========================================

[Enqueue q2 400000] // Enfile un int dans une autre file (q2)
==========================================
Queue: 1 (0x564abb484790 <> 0x564abb484790)
------------------------------------------
0x564abb484790:0x61a80 ((nil) <> (nil))
==========================================

================================================
||                                            ||
||                   DEQUEUE                  ||
||                                            ||
================================================
[Dequeue q "hello"]
Elm dequeue : "Hello"
==========================================
Queue: 2 (0x564abb484730 <> 0x564abb484750)
------------------------------------------
0x564abb484730:0x564aba8c7225 ((nil) <> 0x564abb484750)
0x564abb484750:0x564aba8c723c (0x564abb484730 <> (nil))
==========================================

[Dequeue q "1"]
Elm dequeue : "1"
==========================================
```

```
Queue: 1 (0x564abb484750 <> 0x564abb484750)
-------------------------------------------
0x564abb484750:0x564aba8c723c ((nil) <> (nil))
===========================================

[Dequeue q1 "Hellooooooooo"]
Elm dequeue : "Hellooooooooo"
===========================================
Queue: 0 ((nil) <> (nil))
-------------------------------------------
===========================================

================================================
||                                            ||
||                    FREE                    ||
||                                            ||
================================================
[free q]
[free q1]
[free q2]
```

On voit bien ici que les files sont indépendantes. Ici, on a utilisé des chaînes de caractères et des entiers pour les types des éléments, mais comme "élement" est un type `void *` il est aussi possible d'enfiler des `long` ou même une autre structure. Il faut bien gérer les types des éléments pour les afficher correctement.

2.3. Hashage

Pour cette partie, la principale connaissance que l'on a, est le dernier TP du troisième semestre. Cependant, ce TP a servi surtout comme source d'inspiration. En effet comme dis précédemment, notre projet tourne beaucoup autour de la structure dénommé "Buffer". Cette structure ayant ses propres particularités, il a fallu trouver une fonction de hashage qui fonctionne avec ce Buffer (car le hashage prend un buffer en entrée et retourne un buffer en sortie). N'en trouvant pas d'autres suffisamment intéressantes, nous avons donc réutilisé la fonction de hashage du TP de S3 (fonction de *one_at_a_time* de Jenkins). Cette dernière reste fonctionnel, meme s'il y a bel et bien fallu en partie la modifier.

Ainsi, nous obtenons pour ces différents mots, les résultats en héxadécimal après le hashage suivant:

```
"Hello World!" -> 8b9d610292cc51ba
"Hello world!" -> c02df41901edf271
"  "           -> 0024b3d61336d83b
" "            -> 000009041d191622
"test"         -> 4d79be2f15b21952
"te\0st"       -> d630c7f26e2eb9a8
```

Pour une seule petite modification dans la chaîne de caractère passée au hachage, on obtient un résultat complètement différent du premier. Ainsi on conserve l'intégrité de nos objets (dans ce cas des chaînes de caractères), c'est à dire que 2 objets très similaires mais avec une toute petite différence donnera 2 résultats de hashage complètement différents.

L'autre avantage du hashage est qu'à partir de fichiers très volumineux, on peut obtenir rapidement une unique clé de hashage sur 64 bits. Evidemment cela marche comme pour les petites chaines de caractères, si l'on change ne serait-ce qu'un bit dans le fichier, celui-ci donnera une clé de hashage complètement différente.
Pour des raisons évidentes, nous ne pouvons pas montrer d'exemples avec de fichiers de tailles très volumineuse. Mais cela marche tout aussi bien.

2.4. Table de Hashage

Ayant perdu pas mal de temps sur l'implémentation du hashage via l'utilisatiopn des buffers, cela a forcément retardé l'implémentation d'une table de hashage. Pour le moment on a uniquement la structure, le constructeur et un destructeur qui ne prend pas en compte toute les possibilités.

```
1  // A pair is something where we save some informations about an element and his hash
   keys
2  struct s_pair
3  {
4      Buffer hkey;
5      Buffer 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;
16     size_t size;
17     Pair data;
18 };
19
20 typedef struct s_htab *Htab;
```

2.5. 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 marcherais 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.

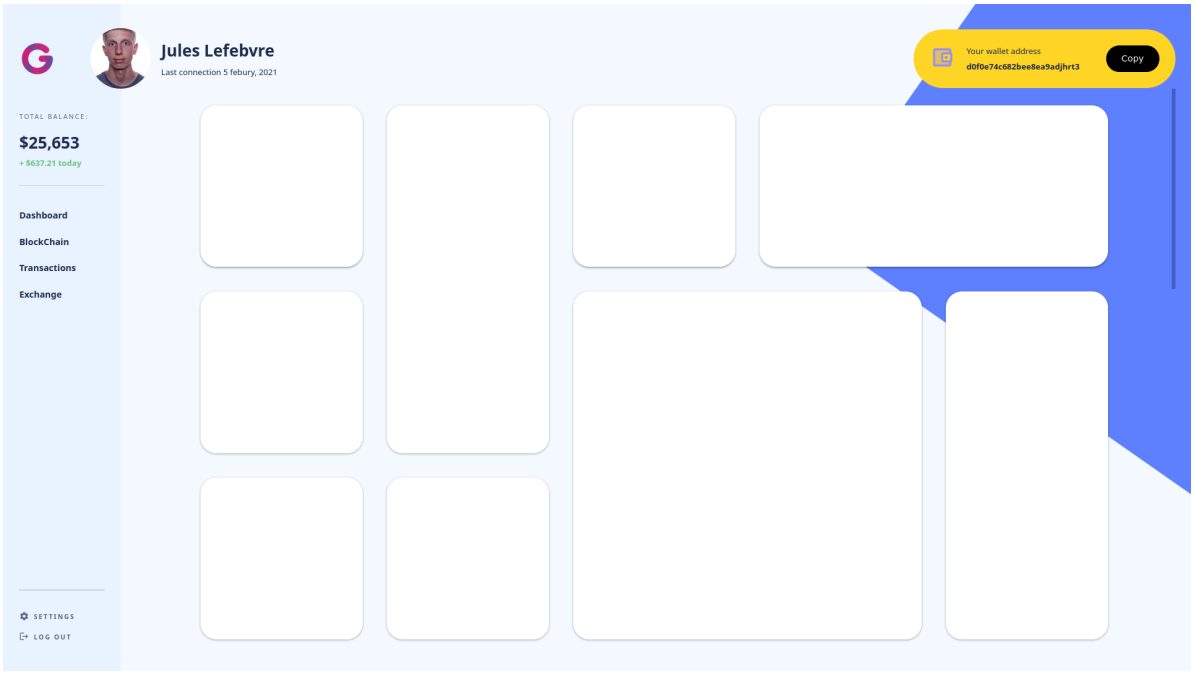
Les bigints possèdent aussi d'autres propriétés afin d'accélérer les operations et de reduire leur emprinte memoire.Les bigint sont normailiser, à leur création le constructeur tronque les octets null de poids forts.

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

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. 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
Buffer	✔	✔
Queue	✔	✔
Hachage	✔	✔
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

2.8. Planning cahier des charges mis à jour

Tâche\Soutenance	1ère	2ème	Finale
Buffer	✔		
BigInt	🏗	✔	
Hachage	🏗	✔	
Crypto Asymétrique		✔	
Queue	✔		
Hash table	🏗	✔	
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

Comme on peut le constater, nous n'avons pas réellement pu tenir le planning du cahier des charges pour cette première soutenance. Brièvement cela est du au manque de préparation et de connaissance de l'ampleur du projet. De plus, nous avons préféré passer plus de temps à avoir un code propre et le plus fonctionnel possible. C'est pour cela que l'on a passé beaucoup de temps à faire et créer des tests unitaire afin de rendre nos parties plus sûres.

2.9. Répartition des charges pour cette soutenance

Tâche	Vincent	Jeanne	Baptiste	Jules
Buffer				☆
Queue	💝	☆		
Hachage	☆	💝		
Hash table	☆	💝		
BigInt			☆	💝
Crypto Asymétrique			☆	💝
File Database		☆		💝
Block Chain	💝	☆		
Transaction		💝	☆	
Site web			💝	☆

☆: Responsable
💝: Suppléant(e)

3. Outil

3.1. Architecture

```
├── CONTRIBUTING.md # règles de contribution
├── Makefile # le Makefile
├── README.md # présentations et utlisation du logiciel
├── build # dossier ou les binaire sont crées
│   ├── client
│   ├── server
│   └── tests # dossier des binaire de tests
│       ├── module1
│       ├── module2
│       └── ...
├── obj # dossier contenant tous les objets compilés
│   └── ...
├── scripts # script de build pour les programmes et tests
│   ├── buidler.py
│   ├── build.py
│   └── tests.py
└── src # source du projet
    ├── builtins # les objets
    │   ├── bigint
    │   ├── buffer
    │   └── queue
    ├── module # modules
    │   ├── hash
    │   └── ...
    ├── client # source du client
    ├── server # source du server
    ├── test # fonctions de tests
    └── utils # fonction basiques et défintions de constantes
```

Tout les modules et builtins ont la structure suivante:

```
module
├── module.h # fichier d'entete contenent tous les déclarations de fonction
├── private # fonction privee necesaire au fichier publique
│   ├── destructor.c
│   └── operator.c
├── public # implémentations des fonctions publiques
│   ├── constructor.c
│   ├── destructor.c
│   ├── getter.c
│   └── operator.c
└── tests # tests unitaire
    ├── module_test.c # fichier d'enter des test du module
    ├── test_constructor.c # sous fichier de tests
    ├── test_constructor.h
    ├── test_destructor.c
    ├── test_destructor.h
    ├── test_getter.c
    ├── test_getter.h
    ├── test_operator.c
    └── test_operator.h
```

3.2. Script de build

Pour éviter d'avoir à écrire toutes les dépendances a la main, nous avons créé des scripts pour pouvoir chercher et compiler toutes les dépendances de binaire. Ces fichiers se trouvent dans le dossier `scripts` de notre projet. Il y a 3 scripts :

3.2.1. `buidler.py`

C'est le script qui contient la majorité de la logique. Grâce à un parcours de profondeur, il va chercher dans chaque fichier ses dépendances (`#include`) et essaye de les résoudre ainsi que ses dépendances. Quand il tombe sur un fichier d'entête (`.h`), il va essayer soit de trouver son fichier source attribué en remplaçant son extension par `.c` ou il va essayer de l'importer comme un module en incluent et compilent tous les fichiers source qui se trouverait dans un sous dossier `public` et `private` .

Après avoir construit l'arbre de dépendance, il va calculer les dates de dernier modification et va recompiler toutes les objects (créé en fonction des `.c`) pour recréer l'exécutable.

3.2.2. `build.py`

Simple script qui permet d'utiliser le script `builder.py` en invité de commande il permet notamment de changer le dossier source, destinations, des objects ainsi que les flags de compiler.

3.2.3. `tests.py`

Il est chargé de chercher tous les suites de test des modules et de les construire. Pour cela, il parcourt tous les dossiers de fichier source et cherche tous les dossiers contenant un fichier `moduleName/tests/moduleName_test.c` qu'il utilisera comme points d'enter pour construire les binaire de tests.

3.2.4 Makefile

Le makefile ne construit pas directement lui-même les binaire, il demande au script de constuction vu plus haut de les construire pour lui.

Il possède notamment une aide (`make help`) qui liste toutes les commandes possible :

```
$ make help
Usage: {Action}[_{Target}]

if no {Target} is specify apply {Action} to all {Target}

Action:
- help: show this help
- clean: remove all the bin and object files
- build: build the target
- debug: build in debug mode the target
- run: run the target
- tests: build and run tests
- tests_build: build tests
- tests_run: run tests

Target: server client
```

3.3. Test suite

Tout module a le droit à sa propre test suite. Cela permet de valider et garantir la sécurité du code tout au long de ces modifications. C'est un point essentiel de la stabilité de notre projet. Comme dit plus haut tout module possède sa suite de test. Pour faire nos tests, nous avons développé une librairie qui permet de normaliser et contrôler les tests que nous faissons.

---

4. Avancement du projet : retards et difficultés

4.1. Buffer

Ce module était plutôt simple et il n'a pas posé de problèmes, cependant il a permis de poser les bases ainsi que les prérequis pour chaque module et aussi pour le formatage du code.

4.2. Queue

Comme précédemment décrit dans la partie Queue, la difficulté venait à gérer les types des éléments enfilés pour l'affichage et tester si tout fonctionnait correctement. On pouvait créer des sous-fonctions pour chaque type ou créait une variable pour prendre en compte le type qui est casté en `void *` mais avec des `char *` et des `int` cela semble suffisant pour ce début de projet. Il est toujours possible de vérifier avec d'autres structures pour les soutenances à venir.

4.3. Hashage

La plus grande difficulté sur le hashage a été de bien comprendre la partie buffer et d'utiliser comme il faut ce dernier, car de part son implémentation un peu particulière, il a fallu réorganisé plusieurs fois l'implémentation de la fonction de hashage. De plus le concept de hashage n'étant pas totalement maitriser il a fallu aussi se l'approprié un peu plus. Puisque le TP de S3 n'a pas suffit et avait été raté pour la plupart des membres du groupe. Ensuite la réalisation de tests unitaires a prit beaucoup de temps pour cette partie, ce qui a retardé encore le projet.

4.4. Table de Hashage

Pour réaliser la table de hashage, il fallait mieux avoir fini le hashage en lui-même, et ce dernier ayant été retardé, c'est que tardivement que nous avons pu commencer la table de hashage. Pour le moment, il n'y a pas eu une grosse rencontre de difficulté.

4.5. BigInt

La mise en place de toutes les fonctions de ce building type a pris plus de temps que prévu à cause de la quantité importante de sous-fonctions ainsi que de tests unitaires à réaliser pour chaque fonction. En effet, avant de réellement commencer à écrire les premières fonctions nous avons du commencer à écrire énormément de fonctions "annexes" que nous n'avions pas pris en compte lors de la création du planning.

De plus, nous avons pris du temps changé plusieurs fois la représentation des nombres. En effet, nous avons inversé l'octet de poids faible plusieurs fois. Etant donné que la représentation des nombres sous format de chaines de caractères est inversé par rapport à sa représentation mémoire nous avons inverser le sens du tableau d'octet.

De plus, une fois toutes les fonctions annexes créées, il a fallu implémenter un environnement complet pour tester nos fonctions avec des tests unitaires le plus exhaustif possible.

4.6. Crypto Asymétrique

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

4.7. File Database

La réalisation du database dépendait énormément des fonctions du blockchain. En effet, le but étant de :

- récupérer les données dans un fichier qui seront les blocs de la blockchain. Il faut alors la fonction qui permet de créer et initialiser ces blocs.
- créer la blockchain suivant la récupération des données. Il faut donc la fonction de création de blockchain.
- écrire dans le fichier les blocs de la blockchain. Il faut pouvoir récupérer les données qui nous intéressent.
- pouvoir libérer la mémoire du blockchain et de ses blocs après utilisation des données.

4.8. Block Chain

Nous n'avons pas pu commencer cette partie car elle dépend directement de la partie "Crypto Asymétrique".

4.9. Transaction

De même pour cette partie qui dépend aussi de la partie "Crypto Asymétrique".

---

5. Conclusion

De manière général, les bases de notre projet sont solides, ce qui a été fait n'aura pas à être modifié ou corrigé par la suite.
Cependant la mise en place de l'environnement de tests, des sous-fonctions a pris un temps considérable que nous n'avions pas pris en compte lors de la réalisation du cahier des charges.
Pour mener notre projet jusqu'à son terme, nous allons devoir redoubler d'efforts pour rattraper le retard pris dans de nombreux modules.