MIAGE NANCY

Bibliographie et application
Projet applicatif

La blockchain

Élèves :
Benoit Cante
Nicolas Lamblin

Enseignant :
Olivier Perrin

3 septembre 2019

Remerciements

Avant toute chose, il nous parait opportun de remercier notre tuteur,
M. Olivier Perrin, qui nous a accompagné le long de cette période de projet, par son suivi régulier, ses conseils pertinents quant à l'organisation et l'avancement de notre étude et sa disponibilité pour répondre aux problèmes que nous avons rencontrés.

Contexte

Ce rapport est l'explication de la partie applicative de notre projet concernant la blockchain. Nous avions ici pour objectif de réaliser une blockchain simplifiée afin de comprendre le fonctionnement mais également les limites de cette technologie.

Vous trouverez les sources de notre application sur ce dépôt GitHub : `https://github.com/nlamblin/blockchain`.

Ainsi que le premier document concernant l'analyse bibliographique **ici**.

Table des matières

1 Introduction

La blockchain est un registre de transactions publiques créée en octobre 2008 suivi d'une publication des sources l'année suivante.
En effet, grâce à cette technologie les utilisateurs ont la possibilité de procéder à des échanges de diverses natures sans utiliser d'intermédiaire. L'ensemble des échanges réalisés depuis la création de la chaîne sont stockés dans un registre, ce qui permet d'identifier les auteurs des transactions. Enfin, le consensus distribué permet aux utilisateurs eux-mêmes de conserver le registre mais également de valider les transactions.
La blockchain a été rendue "populaire" avec l'essor des cryptomonnaies et notamment du Bitcoin. Elle est peut être utilisée pour répondre à diverses problématiques telles que les transactions financières ou encore les signatures de pétitions.

La blockchain, n'est comme son nom l'indique ni plus ni moins qu'une chaîne de blocs liés entre eux où chaque bloc stocke des transactions.

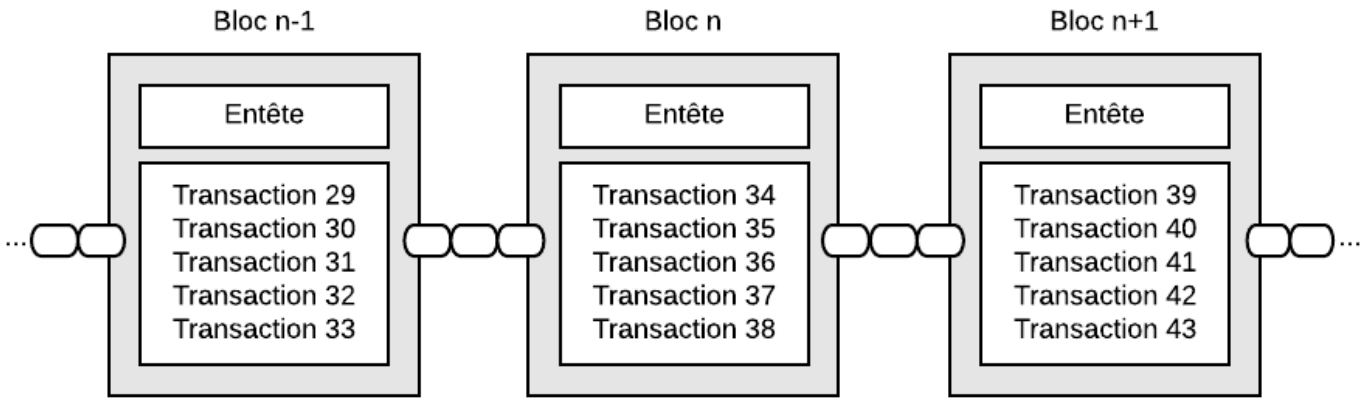

FIGURE 1 – Blockchain simplifiée

La blockchain répond à trois principes fondamentaux :
— Indépendance vis-à-vis des intermédiaires : aspect très intéressant pour le côté éthique (souhait de garder le minimum de personnes possibles impliquées dans un contrat), et pour l'aspect pratique (limiter les coûts et la lourdeur des transactions).
— Traçabilité : cela permet d'avoir plus aisément confiance dans le réseau, car les utilisateurs savent ce qu'il s'y est passé puisque nous savons qui a fait quoi et quand.
— Transparence : afin d'éviter l'aspect "boîte noire" et chaque utilisateur à la garantie que toutes les transactions qui ont lieu peuvent être visibles.

Dans ce rapport nous expliquerons dans un premier temps comment nous avons étudié et appréhendé le problème avec une conception reposant sur UML. Puis nous présenterons comment nous avons implémenté la blockchain. S'en suivra une analyse des résultats que nous avons pu récupérer afin de mettre en relation la difficulté [1] et le temps de minage d'un bloc. Enfin, nous terminerons par expliciter les limites de notre application.

1. La difficulté telle que nous l'avons définie est une mesure qui permet de déterminer combien il est difficile d'ajouter un bloc à la chaine. Notre implémentation de la difficulté se résume au nombre de 0 au début du hash du bloc.

2 Conception

Le fait d'avoir réalisé une étude bibliographique a grandement facilité notre compréhension de la blockchain. En effet, nous avions au fur et à mesure de la découverte de cette technologie, construit un diagramme de classe qui décrit l'ensemble des entités, les liens entre elles ainsi que les méthodes associées.

2.1 Diagramme de classe

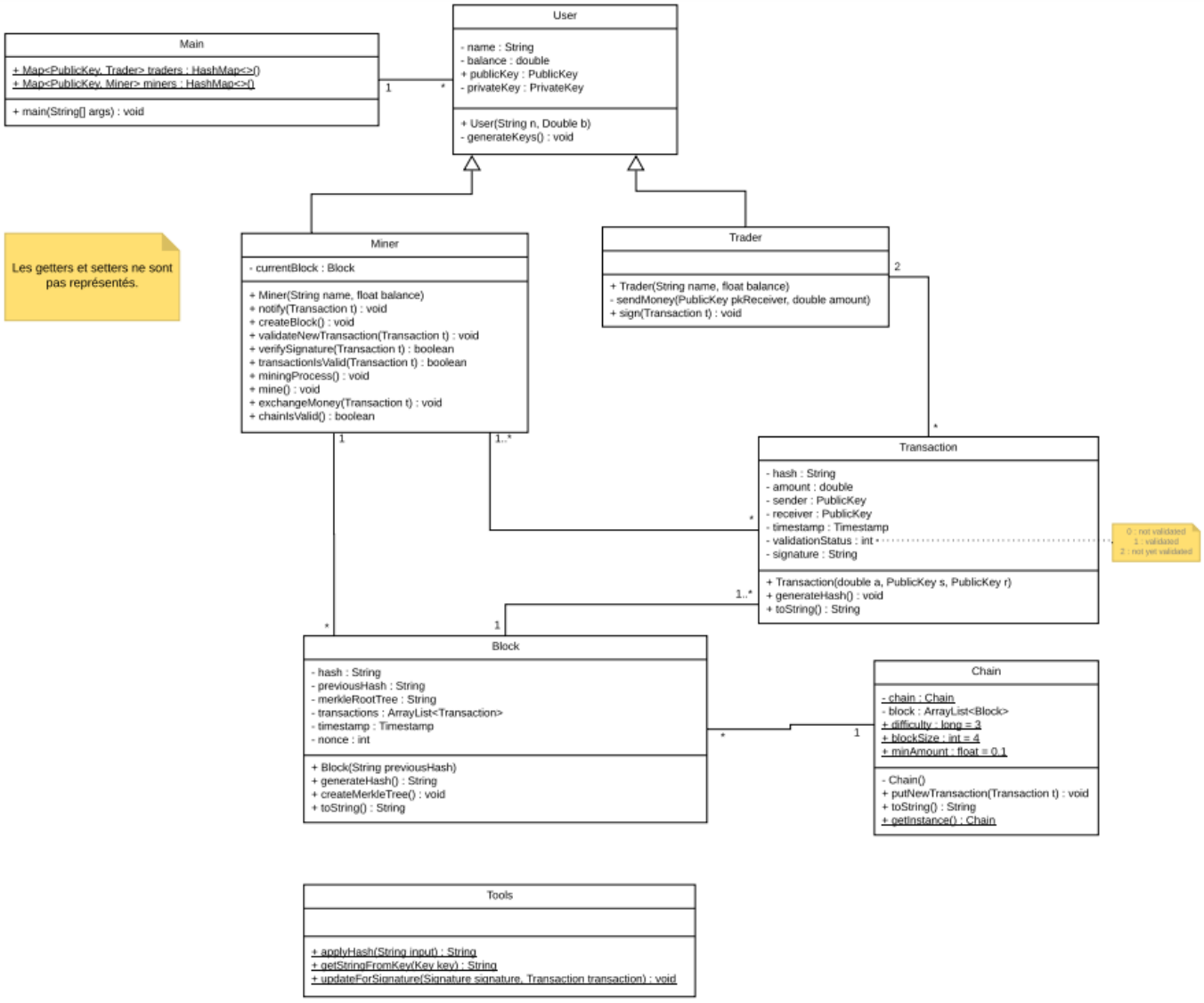

FIGURE 2 – Diagramme de classe

Les trois premières entités que nous avions identifiées étaient celles du coeur de l'application, soit les transactions, la chaine et les blocs. Nous avons également convenu que la chaine serait conçue en singleton afin de s'assurer qu'une seule instance de la chaine ne serait créée.

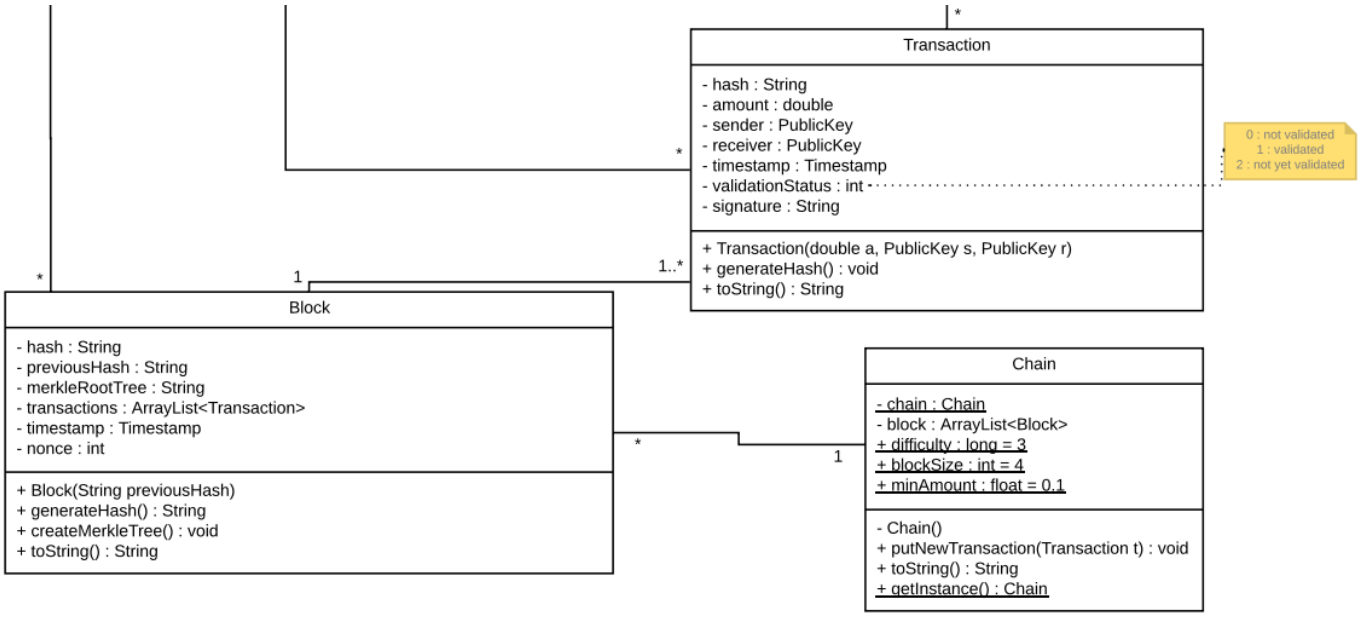

FIGURE 3 – Entités *Transaction*, *Chain*, *Block*

Ensuite, nous avons décidé de créer les entités correspondantes aux utilisateurs. Nous avons discerné deux types d'utilisateurs : les *traders* et les *miners*. Les traders vont être la représentation des personnes échangeant de l'argent. Et les mineurs les personnes qui vont vérifier les transactions et ajouter les blocs à la chaine existante.

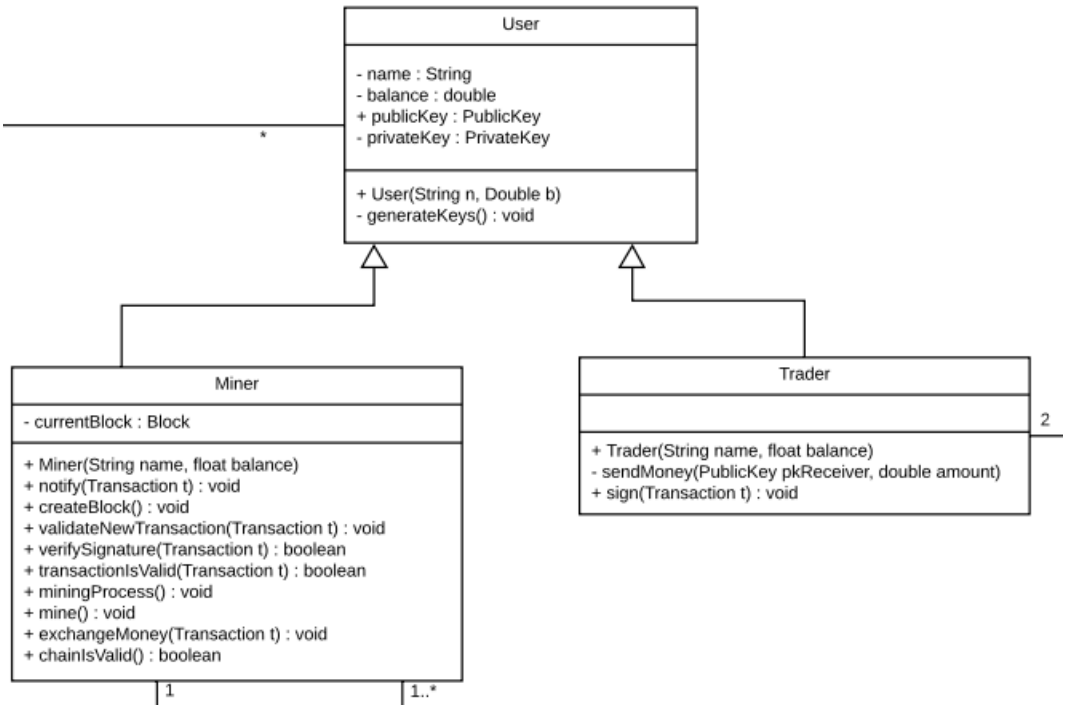

FIGURE 4 – Entités *User*, *Miner*, *Trader*

Enfin, nous utilisons une entité statique nommée *Tools* qui permet de définir des méthodes et de ne pas les dupliquer dans les différentes classes qui en ont besoin.

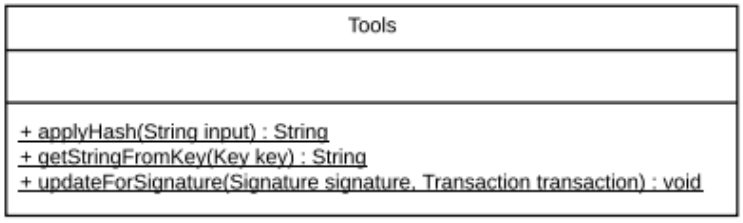

FIGURE 5 – Classe *Tools*

Ce diagramme nous aura servi de base à la première version de l'application. En effet, l'utilisation de thread aura quelque peu complexifié l'architecture initiale.

2.2 Diagrammes de séquence

Même si la blockchain est un système "concurrent", nous avons souhaité expliciter le problème à l'aide d'un diagramme de séquence afin de bien identifier les différentes étapes du processus.

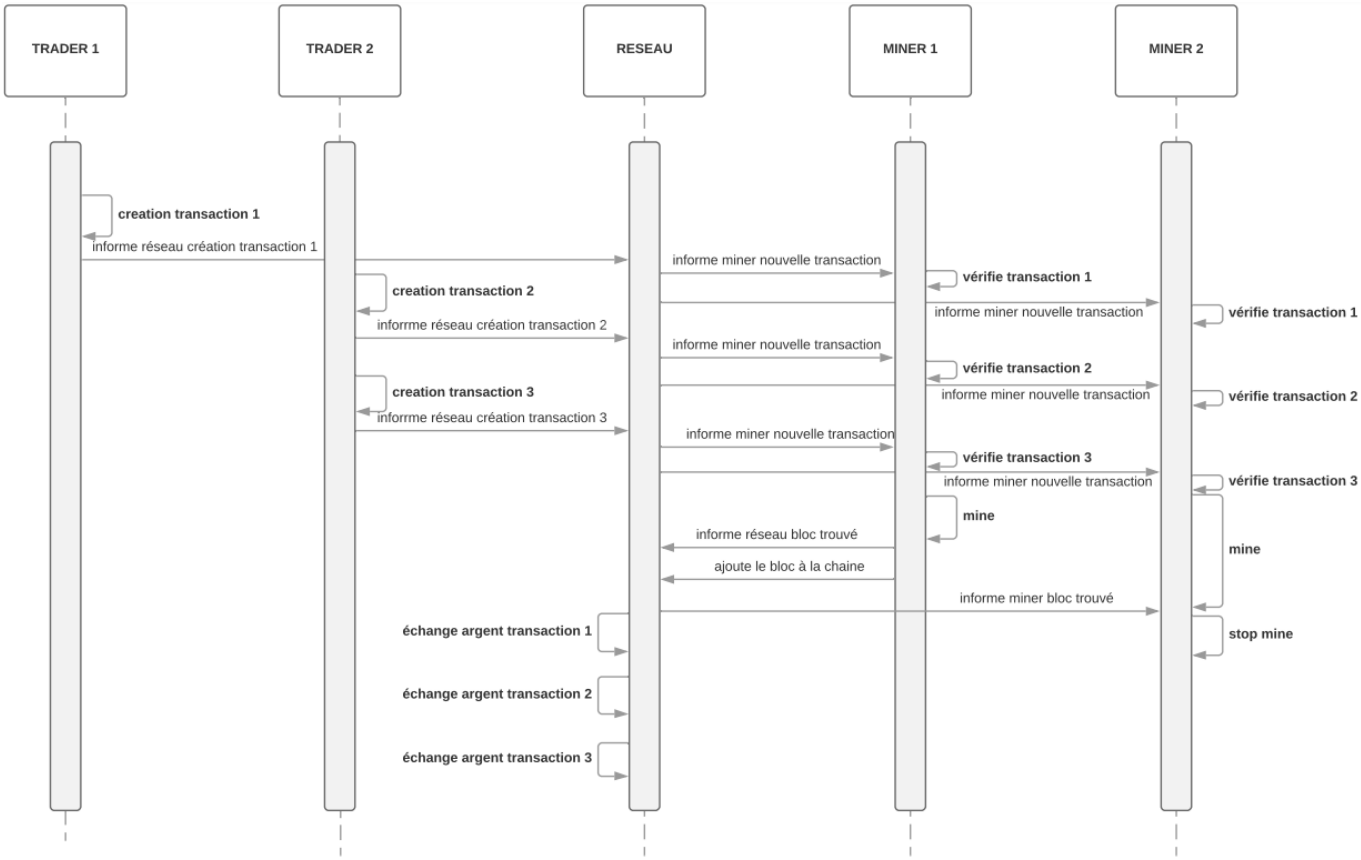

FIGURE 6 – Diagramme de séquence du processus général

3 Réalisation

Afin d'optimiser et d'avoir un suivi clair de l'avancée de notre travail, nous avons défini deux grandes phases de développement :

— La création d'une blockchain séquentielle
— La mise en place des threads

3.1 Création d'une blockchain séquentielle

Comme dit précédemment, cette première version repose sur le diagramme de classe réalisé lors de la phase de conception et d'étude bibliographique.

Création d'une transaction

Pour créer une transaction, autrement dit envoyer de l'argent via le réseau, un trader doit mentionner trois informations, sa clé publique [2] ainsi que celle du receveur et le montant de la transaction. Une fois ces informations saisies, la transaction est transmise à la chaine qui s'occupera de la transmettre aux mineurs afin de la valider, ou non.

Validation d'une transaction

Afin de valider un bloc, un mineur doit vérifier quatre conditions :

— Le receveur doit exister.
— L'envoyeur doit avoir un solde suffisant. Il ne pourrait pas envoyer 10 unités de monnaie s'il n'en possède que 8.
— Le montant minimum d'une transaction doit être respecté.
— La signature de l'envoyeur est correcte.

Création d'un bloc

La création d'un bloc suit un processus très précis. Ce processus se déclenche dès lors qu'un certain nombre de nouvelles transactions sont validées par les mineurs. La première phase correspond à l'initialisation du bloc. A noter que si c'est le premier de la chaine, il ne possède naturellement pas de bloc précédent.

Puis vient le processus de minage du bloc. Il s'agit du coeur de la blockhain car sans minage, la blockchain ne pourrait se construire. La première étape réalisée est la création de l'arbre de merkle. Ce dernier, aussi communément appelé arbre de hachage est une structure de données contenant un résumé des transactions présentes dans le bloc.

```
public void createMerkleTree() {
    int countNodesLeft = this.transactions.size();
    ArrayList<String> previousLayer = new ArrayList<String>();
    for(Transaction t : this.transactions) {
        previousLayer.add(t.getHash());
    }
```

2. Le système de signature numérique est basé sur la cryptographie asymétrique. Cette technique permet de chiffrer la transaction avec une clé dite privée et de déchiffrer avec une clé dite publique.

```
    ArrayList<String> layer = previousLayer;
    while(countNodesLeft > 1) {
        layer = new ArrayList<String>();
        int i = 0;
        while (previousLayer.size() - i > 1) {
            layer.add(Tools.applyHash(previousLayer.get(i)
                + previousLayer.get(i + 1)));
            i += 2;
        }
        if(previousLayer.size() - i == 1) {
            layer.add(previousLayer.get(previousLayer.size()-1));
        }
        countNodesLeft = layer.size();
        previousLayer = layer;
    }
    this.merkleRootHash = layer.get(0); // merkle root hash
}
```

Cette implémentation permet de gérer des listes de transactions de n'importe quelle taille. Cela était nécessaire car le nombre de transactions dans un bloc n'est pas fixe.

L'étape suivante est la recherche du hash en fonction de la difficulté. Notre gestion de la difficulté se résume au nombre de 0 au début du hash.

```
while(!found) {
    Random random = new Random();
    nonce = random.nextInt(Integer.MAX_VALUE);
    this.currentBlock.setTimestamp(
        new Timestamp(System.currentTimeMillis()));
    this.currentBlock.setNonce(nonce);
    hash = this.currentBlock.generateHash();
    if(hash.substring(0, Chain.DIFFICULTY)
        .matches("^[0]{"+ Chain.DIFFICULTY+"}$")) {
        found = true;
    }
}
```

L'objectif ici est de trouver un nombre entre 0 et la valeur maximale d'un nombre entier qui permet de satisfaire la difficulté. Après avoir renseigné la date courante, une expression régulière vérifie si le hash est correct ou non. Si ce n'est pas le cas, un nouveau nombre aléatoire est généré et ainsi de suite.

Enfin, les deux dernières étapes sont quant à elles beaucoup plus simples car elles ne visent qu'à procéder à l'échange de l'argent entre les parties et à ajouter le bloc à la chaîne.

Validation de la chaine

Les mineurs ont également la possibilité de vérifier l'intégrité de la chaine. Cela vise à contrôler que le hash du bloc à la position n de la chaine est identique à la valeur du hash précédent du bloc à la position n+1.

```
public boolean chainIsValid() {
    boolean result = true;
    int i = 1;
    while(i < Chain.getInstance().getBlocks().size() && result) {
        if (!Chain.getInstance().getBlocks().get(i).getPreviousHash()
                .equals(
                Chain.getInstance().getBlocks().get(i-1).getHash())) {
            result = false;
        }
        i++;
    }
    return result;
}
```

Les tests unitaires

Dans une démarche qualité, nous avons implémenté plus de 30 tests unitaires afin de couvrir l'ensemble de nos méthodes. Cela nous a permis d'avoir un code fonctionnel à chaque modification ou évolution et éviter ainsi toute régression. Prenons par exemple la méthode de construction de l'arbre de Merkle présent dans un bloc. Cette méthode est couverte par 4 tests unitaires afin de tester toutes les possibilités. Grâce à ces tests nous avons gagné un temps considérable, notamment en évitant les bugs de régression.

La structure d'un test est la suivante :

— Avant de lancer n'importe quelle classe de test les utilisateurs (mineurs et traders) sont réinitialisés.

— Puis, avant chaque test, la blockchain est également réinitialisée. Autrement dit, tous les tests vont s'éxecuter sur une blockchain vide. Cela n'a en effet aucune importance car l'objectif est de vérifier le comportement de chaque méthode indépendamment.

3.2 Mise en place des threads

Une fois la blockchain séquentielle fonctionnelle, nous avons décidé de rendre la blockchain plus "vivante" et simuler au mieux l'activité réelle. Pour cela, nous nous sommes appuyés sur des threads à plusieurs niveaux.
En premier lieu, nous avons décidé de traduire en threads tous les mineurs, car l'intérêt principal de notre application est de montrer les calculs parallèles faits entre les mineurs pour résoudre les calculs. A cet effet nous avons commencé par faire une blockchain ou seul un mineur pouvait miner [3] en même temps sur la chaîne, avec une méthode *mine*.

Lors de la modification du programme pour incorporer les threads, nous avons dû revoir le code existant. En effet, il fallait que la méthode de minage se retrouve dans une classe à part, afin que le mineur puisse avoir le contrôle sur son GPU, et éventuellement le couper. Nous avons donc décidé de créer une classe GPU qui simule la carte graphique du mineur, et qui va essayer différents hashs possibles jusqu'à ce qu'elle mine le bloc. Il est aussi possible d'envoyer un message aux mineurs pour qu'ils arrêtent de miner.

Le fonctionnement global de l'application est le suivant :

1. Au début, les mineurs sont initialisés.
2. Ensuite, le serveur va se mettre en attente de transactions.
3. Dès qu'il en a assez pour faire un bloc, il transmet ces requêtes aux mineurs et lance les threads des mineurs. Ces derniers vont dans leurs méthodes lancer chacun un thread. Tous les mineurs sont alors en concurrence pour miner un bloc qui contient les quatre mêmes transactions.
4. Dès qu'un mineur a trouvé un hash valide, celui-ci transmet le bloc nouvellement miné au serveur. C'est ensuite ce dernier qui va l'ajouter à la chaîne, sans tenir compte des autres mineurs qui auraient pu trouver un hash valide pendant ce temps.

Cette procédure sera répétée pour toutes les transactions à venir.

Nous avons également décidé de créer un thread annexe pour la génération de nouvelles transactions. Toutes les 2 secondes, le thread va créer une transaction et l'ajouter au pool de transactions [4] du serveur.

Thread safety

Pour s'assurer que notre application reste thread-safe, autrement dit, qu'elle n'ait pas d'effets indésirables quand plusieurs threads sont lancés (comme des situations de compétition [5]) , nous avons pris diverses mesures.

Nous nous sommes notamment assurés que les blocs critiques exécutés par le serveur soient exécutés par ce dernier uniquement. Ainsi, une fois que le serveur a reçu un bloc correct, les autres blocs envoyés sont ignorés (grâce à la méthode *invokeAny* qui récupère

3. Action de chercher un hash satisfaisant la difficulté
4. Liste dans laquelle les nouvelles transactions sont stockées
5. https ://fr.wikipedia.org/wiki/Situation_de_comp%C3%A9tition

le premier objet qui arrive à son exécution. Toutes les méthodes d'ajout à la chaîne et d'échanges de fonds sont exécutées par le serveur. Nous sommes ainsi garantis que l'échange n'a lieu qu'une fois. Certaines variables (comme *pool*, la liste des transactions en attente) sont déclarées volatiles pour s'assurer qu'elles ne puissent être accédées que par un thread à la fois.

Version "Preuve de concept"

Comme dit précédemment, nous avons décidé de faire une première version de la blockchain dite séquentielle. Il nous a été particulièrement difficile de la rendre concurrente d'un coup, alors nous avons décidé après quelques essais de faire une version "Preuve de concept", où le minage consistait à compter le nombre de caractères de différents mots avec des délais d'attente aléatoires. Ceci nous a permis de nous familiariser avec les classes *ExecutorService* qui permettent de gérer plus facilement les threads qu'en appelant ces derniers directement.
Notre "Preuve de concept" est donc disponible sur la branche GIT *POC-Multithreading-with-callables*.

4 Analyses et estimations de performances

4.1 Temps de minage selon la difficulté

Comme dit précedemment, notre gestion de la difficulté consiste à trouver des hashs qui commencent par une certaine séquence de caractères (ici des 0). Un hash satisfaisant pour une difficulté n commence donc par n 0. Nous nous sommes rapidement rendu compte que cette manière de gérer la difficulté manquait de subtilité : nous constations des temps de minage très rapide jusqu'à une difficulté de 4, de quelques secondes à 5, et de l'ordre de l'heure pour une difficulté de 6.

Nous avons donc décidé de faire quelques calculs pour avoir un ordre d'idée des temps de calculs nécessaires selon nos machines, et voir quel était l'impact d'autoriser d'autres caractères (par exemple des 1) dans le calcul.

Pour cela, nous avons fait tourner nos ordinateurs sur du calcul de hash afin d'estimer le nombre de hash par minute qu'ils pouvaient effectuer (environ 250 000 000) et nous avons effectué pour toutes les difficultés (de 1 à 20) les calculs suivants (dans ce modèle, nous souhaitons que tous les n premiers caractères soient un 0) :

Proportion de hash satisfaisant la difficulté :

$\frac{1}{16}^n$ où n est la difficulté.

Nombre de hashs à effectuer avant d'en trouver un satisfaisant la difficulté (en moyenne) :

$\frac{1}{x}$ où x est la proportion de hash satisfaisant la difficulté.

Nombre de résultats corrects par minute (en moyenne) :

$\frac{250000000}{y}$ où y est le nombre de hashs à effectuer avant d'en trouver un satisfaisant la difficulté (en moyenne)

Nombre de minutes à attendre avant d'avoir un résultat correct (en moyenne) :

$\frac{1}{z}$ où z est le nombre de bons résultats par minute (en moyenne)

Nous comprenons alors très vite pourquoi le temps augmente de manière considérable. D'après nos calculs, pour une difficulté de 6, nous obtenons 1,5 résultat en moyenne par minute. Résultat qui passe à 0,01 par minute en difficulté 7, et finalement 0,005 en 8.
Il est important de mentionner que les résultats observés sont différents des résultats estimés avec le hash rate calculé, mais les ordres de grandeur restent comparables. Les différences entre estimation et observation s'expliquent notamment par le fait que l'application blockchain s'exécute différemment que le code qui sert à calculer des hashs.

# of 0	Proba	Average # of hashes before finding a good one (1 / Proba)	HIT / minut	Average min before hit (1 / HpM)
1	0,0625	16,00	1561524,388	6,404E-07
2	0,00390625	256,00	97595,27422	1,02464E-05
3	0,000244141	4 096,00	6099,704639	0,000163942
4	1,52588E-05	65 536,00	381,2315399	0,002623078
5	9,53674E-07	1 048 576,00	23,82697124	0,041969245
6	5,96046E-08	16 777 216,00	1,489185703	0,671507924
7	3,72529E-09	268 435 456,00	0,093074106	10,74412679
8	2,32831E-10	4 294 967 296,00	0,005817132	171,9060286
9	1,45519E-11	68 719 476 736,00	0,000363571	2750,496457
10	9,09495E-13	1 099 511 627 776,00	2,27232E-05	44007,94332
11	5,68434E-14	17 592 186 044 416,00	1,4202E-06	704127,0931
12	3,55271E-15	281 474 976 710 656,00	8,87624E-08	11266033,49
13	2,22045E-16	4 503 599 627 370 500,00	5,54765E-09	180256535,8
14	1,38778E-17	72 057 594 037 927 900,00	3,46728E-10	2884104573
15	8,67362E-19	1 152 921 504 606 850 000,00	2,16705E-11	46145673173
16	5,42101E-20	18 446 744 073 709 600 000,00	1,35441E-12	7,38331E+11
17	3,38813E-21	295 147 905 179 353 000 000,00	8,46504E-14	1,18133E+13
18	2,11758E-22	4 722 366 482 869 650 000 000,00	5,29065E-15	1,89013E+14
19	1,32349E-23	75 557 863 725 914 300 000 000,00	3,30666E-16	3,0242E+15
20	8,27181E-25	1 208 925 819 614 630 000 000 000,00	2,06666E-17	4,83872E+16

FIGURE 7 – Estimation du temps de minage avec des 0

# of 01	Proba	Average # of hashes before finding a good one (1 / Proba)	HIT / minut	Average min before hit (1 / HpM)
1	0,125	8,00	3273443,125	3,05489E-07
2	0,015625	64,00	409180,3906	2,44391E-06
3	0,00195313	512,00	51147,54883	1,95513E-05
4	0,00024414	4 096,00	6393,443604	0,00015641
5	3,0518E-05	32 768,00	799,1804504	0,001251282
6	3,8147E-06	262 144,00	99,8975563	0,010010255
7	4,7684E-07	2 097 152,00	12,48719454	0,080082039
8	5,9605E-08	16 777 216,00	1,560899317	0,640656312
9	7,4506E-09	134 217 728,00	0,195112415	5,125250496
10	9,3132E-10	1 073 741 824,00	0,024389052	41,00200397
11	1,1642E-10	8 589 934 592,00	0,003048631	328,0160317
12	1,4552E-11	68 719 476 736,00	0,000381079	2624,128254
13	1,819E-12	549 755 813 888,00	4,76349E-05	20993,02603
14	2,2737E-13	4 398 046 511 104,00	5,95436E-06	167944,2083
15	2,8422E-14	35 184 372 088 832,00	7,44295E-07	1343553,666
16	3,5527E-15	281 474 976 710 656,00	9,30368E-08	10748429,33
17	4,4409E-16	2 251 799 813 685 250,00	1,16296E-08	85987434,63
18	5,5511E-17	18 014 398 509 482 000,00	1,4537E-09	687899477
19	6,9389E-18	144 115 188 075 856 000,00	1,81713E-10	5503195816
20	8,6736E-19	1 152 921 504 606 850 000,00	2,27141E-11	44025566528
21	1,0842E-19	9 223 372 036 854 780 000,00	2,83926E-12	3,52205E+11
22	1,3553E-20	73 786 976 294 838 200 000,00	3,54907E-13	2,81764E+12
23	1,6941E-21	590 295 810 358 706 000 000,00	4,43634E-14	2,25411E+13
24	2,1176E-22	4 722 366 482 869 650 000 000,00	5,54543E-15	1,80329E+14
25	2,647E-23	37 778 931 862 957 200 000 000,00	6,93179E-16	1,44263E+15

FIGURE 8 – Estimation du temps de minage avec des 0 et des 1

4.2 Estimation du temps de minage selon le nombre de mineurs

Nous nous sommes ensuite penchés sur l'intérêt de rajouter de plus en plus de mineurs à la blockchain.

Les pistes que nous avions identifiées étaient les suivantes :

— Plus il y a de mineurs qui essayent de trouver un hash correct, plus le temps pour qu'un mineur trouve une bonne solution diminue. En effet, trouver un hash pertinent se résume à tester aléatoirement des possibilités. Si plusieurs personnes essaient en parallèle des combinaisons différentes, nous devrions couvrir le champ des possibles plus rapidement.

— De la même manière, nous nous attendons à ce que le temps total pour trouver un hash augmente. En effet, le programme doit gérer de plus en plus de ressources, et les différents mineurs auront individuellement moins de temps pour miner, car il faudra partager le temps de processeur avec de plus en plus d'acteurs.

Les graphiques suivants visent à montrer l'évolution du temps d'exécution global pour miner un bloc, mis en rapport avec le temps d'exécution utilisé exclusivement par le mineur "gagnant".

Il est important de mentionner que tous les 10 blocs générés, un mineur est rajouté à la blockchain.

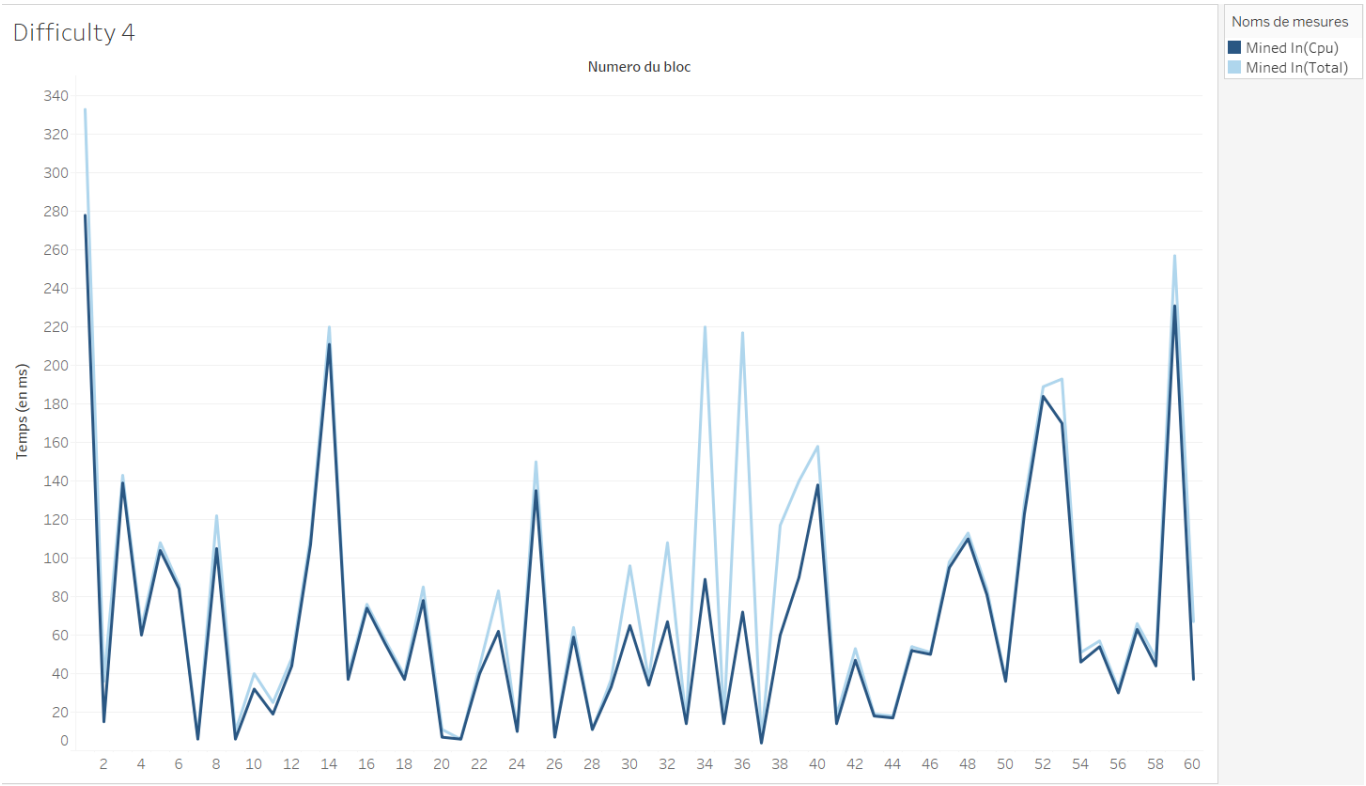

FIGURE 9

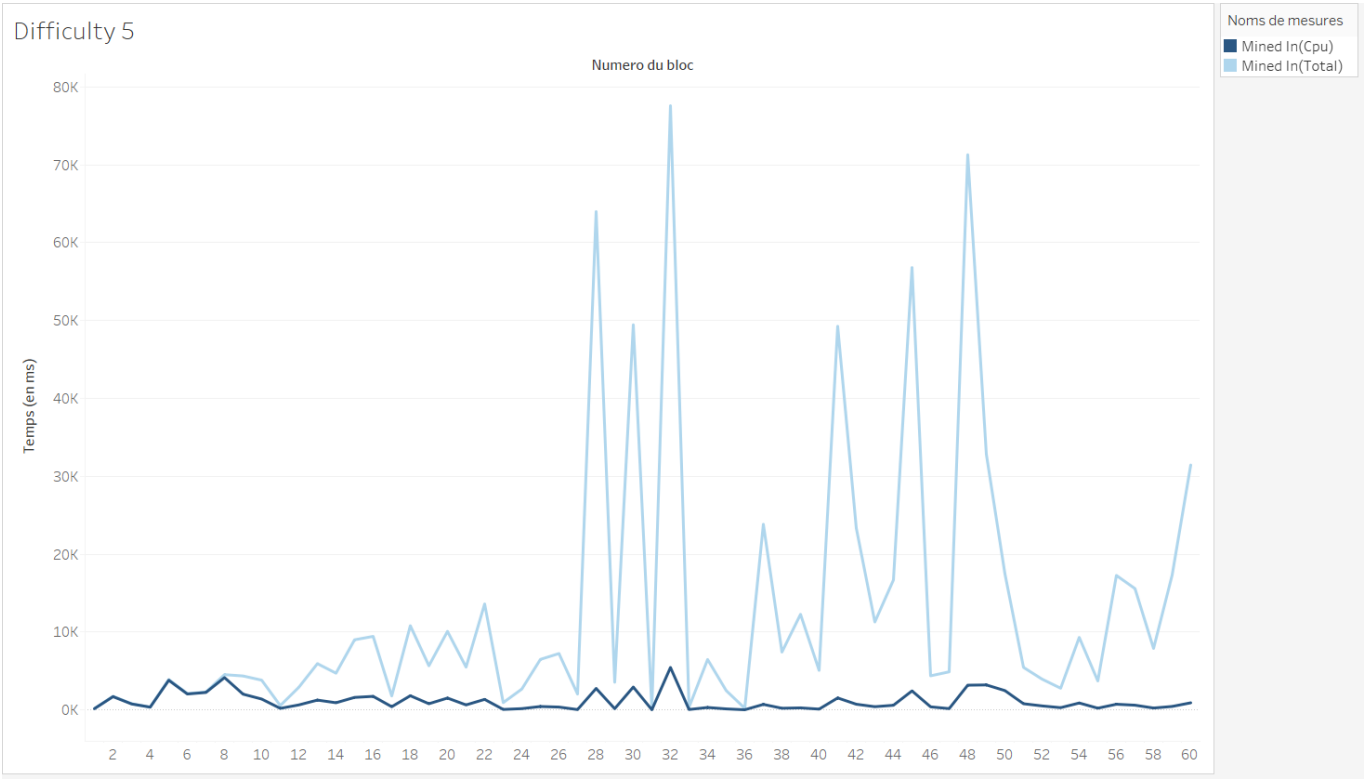

FIGURE 10

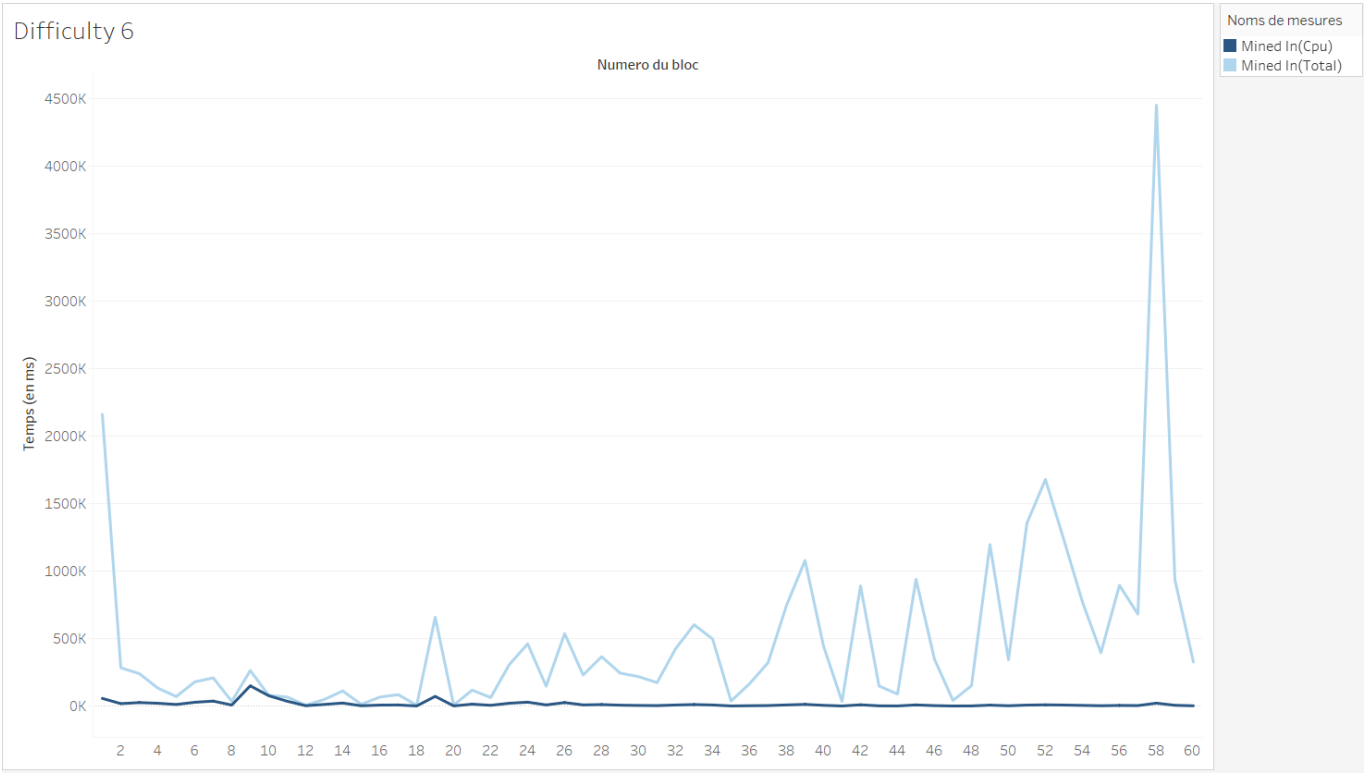

FIGURE 11

Nous constatons sur ces graphiques, que le temps pour trouver un bloc est globalement en "dents-de-scie", ce qui s'explique par la nature aléatoire du minage de bloc. Il est important de montrer qu'à cause des échelles, l'aspect "dent-de-scie" est moins visible pour le temps propre à un mineur lorsque la difficulté augmente. Nous constatons également que le temps global d'exécution augmente au fur et à mesure que l'on rajoute des mineurs. Ce qui confirme notre hypothèse préalable. Cependant, il n'y a pas de baisse significative du temps avant de trouver le bon hash par un mineur, contrairement à ce que nous avions préalablement imaginé.

5 Limites de notre application

Nous avons eu de grandes difficultés avec les threads. N'étant pas familiers avec ces derniers, nous avons commencé en essayant de créer directement des threads grâce à l'interface *Runnable*. Cependant, nous avions des difficultés pour arrêter puis relancer les threads, notamment en fournissant des arguments différents. De plus, la communication entre Serveur, GPU, et Mineurs étaient compliqués à mettre en place. Nous nous sommes alors orientés vers la classe *ExecutorService*, qui rendait la manipulation des threads beaucoup plus simple. Cela vient malheureusement au détriment des performances, car la gestion des threads induit un fort overhead.

Il est particulièrement difficile de faire des tests unitaires quand le programme devient concurrentiel. Nous avons pu faire régulièrement des tests quand la création de la chaîne ne dépendait que d'un seul mineur, mais il est bien plus difficile d'en faire sur la création de la chaîne grâce au minage de plusieurs threads.

L'estimation et l'ajustement de la difficulté sont particulièrement difficiles compte tenu de nos équipements et de la méthode du calcul de difficulté. Cela nous empêche de réajuster efficacement la difficulté pour atteindre un temps de minage optimal, et la difficulté sera toujours largement trop haute ou trop basse. Dans cette implémentation, nous avons fait le choix de rester sur une difficulté "simple", déterminée par le nombre de 0.

Finalement, il convient de mentionner les limites qui ont été définies dès le début du projet : Notre blockchain est sur un modèle clients / serveur, ce qui est contraire à l'idée de décentralisation du réseau. Il n'est pas possible de faire un "fork" sur la chaîne, car tous les mineurs accèdent aux mêmes transactions au même moment, et seul le bloc miné par le mineur le plus rapide est transmis au serveur pour ajout. Finalement, la difficulté est calculée de manière plus triviale que dans une cryptomonnaie.

6 Conclusion

Comme expliqué précédemment, les plus grandes difficultés que nous avons rencontrées lors de la réalisation de ce projet sont sans aucun doute liées aux threads. En effet, leur comportement est assez difficile à tester ce qui nous a obligé à repenser l'architecture de l'application.

Outre ces problèmes, implémenter une blockchain est le meilleur moyen pour en comprendre son fonctionnement. Etant une technologie d'avenir, y avoir été sensibilisés ne peut être qu'un avantage pour nous. Quand nous voyons le nombre d'entreprises et les types d'entreprises (télécoms, banques, assurances, ...) qui s'y intéressent, nous comprenons la chance que nous avons eu d'obtenir un projet tel que celui-ci.

Références

[1] https://anders.com/blockchain/blockchain.html

[2] https://medium.com/programmers-blockchain/create-simple-blockchain-java-tutorial-from-scratch-6eeed3cb03fa

[3] https://medium.com/programmers-blockchain/creating-your-first-blockchain-with-java-part-2-transactions-2cdac335e0ce

[4] https://keyholesoftware.com/2018/04/10/blockchain-with-java

[5] https://medium.com/@JB_Pleynet/le-chiffrage-à-clés-publiques-privées-expliqué-aux-non-initiés-1a0eed15934f

[6] https://niels.nu/blog/2016/java-rsa.html