2020/2021

Projet S4

Rapport de soutenance

Groupe TREG

CAVALIÉ Margaux
LITOUX Pierre
PINGARD Adrien
RUIZ Hugo
VEYRE Thimot

Encadrant : VERNAY Rémi

TABLES DES MATIÈRES

Abstract

Depuis plus de dix ans, la conception et l'utilisation de crypto-monnaies connaissent un grand essor. Ces dernières, qui ne sont contrôlées par aucune institution, sont dorénavant acceptées comme moyen de paiement. En développant un réseau en peer to peer et une blockchain, TREG a créé sa propre crypto-monnaie, le TEK. Ce réseau, développé en langage C, est accessible depuis une interface web en Javascript, exploitant l'environnement Node.js. Au travers de ce projet, TREG souhaite mettre en avant les crypto-monnaies, qui constitueront probablement les échanges monétaires de demain.
#crypto-monnaie #blockchain #décentralisation #transactions #token

Introduction

Ce cahier des charges présente notre projet de S4, un logiciel dans lequel l'algorithmique occupe une part très importante. Le sujet est libre, et encadré par le docteur en informatique et expert auprès de la cour d'appel de Toulouse Rémi Vernay. Le projet sera développé sous Linux, et codé en langage C.

Ce cahier contient une présentation générale du projet, en plus de la répartition des tâches et du planning détaillé de réalisation. Notre groupe est actuellement constitué de 5 personnes : Cavalié Margaux, Litoux Pierre, Pingard Adrien, Ruiz Hugo et Veyre Thimot. Il est fort probable que Pierre Litoux parte en séjour à l'international, c'est pourquoi nous sommes un groupe de cinq personnes, et non quatre comme le prévoit le sujet.

Pour notre projet, nous avons choisi de recréer la technologie d'une blockchain. Cette technologie est décrite pour la première fois dès 1991 par les chercheurs Stuart Haber et W. Scott Stornetta, qui cherchaient à horodater des documents numériques pour que ceux-ci ne soient jamais antidatés ou altérés. Cette technologie tombera dans l'oubli, et perdra son brevet en 2004, quatre ans avant la création du Bitcoin.

En 2004, un activiste cryptographique, Hal Finney, lance le système appelé "preuve de travail réutilisable", qui permet de conserver le registre de propriété des informations envoyées pour permettre à n'importe quel utilisateur du réseau de vérifier l'exactitude et l'intégrité des données en temps réel. Ce n'est qu'en 2008 que Satoshi Nakamoto fait naître le Bitcoin en reliant ces deux technologies. Il envoya dix Bitcoins à Hal Finney, et offrit cinquante Bitcoins de récompense à celui qui minerait le bloc. Le Bitcoin et la Blockchain comme nous les connaissons sont ainsi créés.

L'objectif de notre projet est de récréer cette technologie. Pour cela, il nous faudra nous intéresser à la cryptographie et à la création d'un réseau qui mettrait en relation tous les différents utilisateurs. Le dernier rendu du projet est prévu pour la semaine du 14 juin 2021, nous avons ainsi trois mois et demi pour porter notre projet à terme.

État de l'art

Fonctionnement de la blockchain

Le terme blockchain (ou chaîne de blocs) désigne la technologie de consensus décentralisé. La blockchain est donc une technologie de stockage et de transmission d'informations, transparente, sécurisée, fonctionnant sans organe central de contrôle.

Toute blockchain publique fonctionne nécessairement avec une monnaie ou un jeton programmable. Les transactions effectuées entre les utilisateurs du réseau sont regroupées par blocs.
Avant d'être approuvé, chaque bloc est validé par les nœuds du réseau (appelés les "mineurs"), selon des techniques qui dépendent du type de blockchain. Dans la blockchain du bitcoin cette technique est appelée le "Proof-of-Work" (preuve de travail), et consiste en la résolution de problèmes algorithmiques.
Une fois le bloc validé, il est horodaté et ajouté à la chaîne de blocs. La transaction est alors visible pour le récepteur et pour l'ensemble du réseau. Au cours du temps, des blocs sont ajoutés à la chaîne. Une blockchain publique est donc comparable à un grand livre comptable public : anonyme et infalsifiable.

A l'inverse, une blockchain privée est contrôlée par une entité totalement centralisée dans le réseau, et les membres participants doivent avoir été acceptés et déclarés par cette entité. Les transactions peuvent être émises par chacun des nœuds, mais elles sont validées et ajoutées à la chaîne par le nœud central autorisé à le faire. Il n'existe aucun mécanisme de consensus, et les règles de fonctionnement sont spécifiques au dispositif et aux accords passés par les membres de la communauté d'utilisateurs.

La blockchain contient l'historique de tous les échanges effectués entre ses utilisateurs depuis sa création. Cette base de données est sécurisée et distribuée, c'est-à-dire qu'elle est partagée par ses différents utilisateurs, sans intermédiaire, ce qui permet à chacun de vérifier la validité de la chaîne.

Ce qui rend la blockchain si fiable et sécurisée, c'est l'ensemble des méthodes qui permettent aux participants du réseau distribué de se mettre d'accord, sans recourir à un tiers de confiance.

Il existe actuellement deux grandes familles de blockchain. Leur différence réside dans la manière dont les blocs sont validés. La première est la validation par "preuve de travail" (proof of work), et la deuxième la validation par "preuve d'enjeu" (proof of stake).

Validation par proof of work

La validation par preuve de travail est la plus simple à mettre en place. En effet, dans ce système les mineurs doivent résoudre un problème mathématiques très complexe, plus le nombre de mineurs augmente, plus cette complexité augmente également, pour valider un bloc. Ce problème dépend du bloc précédent, de cette manière pour modifier un bloc donné dans la chaîne, il faut également recalculer tous les blocs suivants.

Cette méthode de validation à l'inconvénient majeur d'être extrêmement énergivore et pose un gros problème environnemental.

Validation par proof of stake

Le deuxième principe de validation est la preuve d'enjeu. Cette méthode choisit un ou plusieurs mineurs parmi ceux disponibles et ceux-ci valident le bloc sans calcul complexe. Les mineurs sélectionnés peuvent être choisis selon plusieurs critères (leur ancienneté, leur quantité de monnaie, etc). Ce système permet de réduire grandement la consommation énergétique lié au minage mais il a l'inconvénient majeur d'être de ce fait un peu moins sécurisé

Ces méthodes, pour fonctionner à grande échelle, doivent être rémunérées pour que le réseau continue à fonctionner. En effet, les membres du réseau doivent pouvoir rentabiliser les dépenses liées aux vérifications.

Notre projet

Notre cryptomonnaie utilisera deux familles d'acteurs pour fonctionner.
Les premiers sont les nœuds de gestion, qui s'occupent de recevoir les transactions et de créer les blocs. Les seconds sont les nœuds de minages, qui sont appelés par les nœuds de gestion pour résoudre un problème mathématique permettant de sécuriser la blockchain.
Les nœuds de gestion devront intégrer des sécurités permettant d'éviter la création de blocs malicieux.

Le réseau devra être constitué d'au minimum un nœud de gestion et un nœud de minage. Même si cela implique de gros problèmes de sécurité, nous souhaitons que ce soit possible car c'est un projet, et que le réseau ne sera constitué que de quelques ordinateurs de confiance lors de nos tests.

Pour le système de validation, nous avons décidé de nous orienter vers la validation "proof of work", qui nous semble plus simple à implémenter. Étant donné que le projet restera à petite échelle, l'impact environnemental sera minime. Néanmoins, s'il venait à prendre de l'importance, il nous faudrait le faire fonctionner en "proof of stack" pour réduire son impact énergétique.

La blockchain permet de stocker et transmettre des données, mais les utilisateurs de celle-ci auront besoin d'une interface pour réaliser leurs transactions et consulter l'historique des transactions déjà réalisées. Comme pour la blockchain du Bitcoin, il nous faudra un site ou toutes ces informations sont accessibles en quelque clic , il devra donc intégrer une gestion de compte ou chaque utilisateur aura accès à ses données personnelles tel que visualiser le solde du compte et réaliser des transactions avec les autres utilisateurs. Un site avec une certaine sécurité devra alors être mis en place pour éviter que d'autres personnes y accèdent de façon illégitime et détourne nos fonds.

Nous coderons notre site en Javascript et en CSS à l'aide Node.js un environnement Java qui exécute du code sur le serveur du site , pour la gestion des comptes utilisateurs nous utiliserons MySQL, qui permet de créer puis gérer les base de données. Les utilisateurs pourront ainsi créer leur compte et obtenir un portefeuille qui leur permettra d'échanger des données avec les autres utilisateurs du réseau en toute sécurité.

Planning

Répartition des tâches

	Réseau	Noeud de minage	Noeud de gestion	Site web
Margaux		⊕		+
Pierre	+	+	+	
Adrien			⊕	
Hugo	+			⊕
Thimot	⊕			

Légende :

⊕ → Responsable

\+ → Suppléant

Planning détaillé de réalisation

Première soutenance

Réseau :

- Simulation d'un réseau en multi-thread
- Exécution des différent noeuds sur différents process
- Début de mise en réseau sur plusieurs machines

Site web :

- Ébauche de site web

Noeuds :

- Début du noeud de gestion
- Fonction de hash primaire et minage brute force
- Interaction noeud de minage/noeud de gestion primaire

Deuxième soutenance

Réseau :

- Réseau fonctionnel

Site web :

- Amélioration de l'esthétique du site
- Création de compte
- Intégration interface de transactions

Noeuds :

- Utilisation du noeud de minage pour valider un bloc
- Intégration de plusieurs transactions par bloc
- Mise en place de liste de transactions en cours

Soutenance finale

Réseau :

- Réseau fiable

Site web :

- Site web esthétiquement abouti
- Système de compte et de connexion sur le site

Noeuds :

- Echange de données avec les autres noeuds pour mettre à jours les transaction et la blockchain

Avancement du projet

Pour la première soutenance nous avions été amenés à changer quelque peu la répartition des tâches, en fonction des goûts personnels de chacun et de la charge de travail que chaque partie nécessitait.
Ainsi Pierre s'était finalement positionné sur la gestion des nœuds de minage au vue de l'annulation de son départ à l'international, plutôt que d'être disponible comme soutien sur les autres tâches. Margaux quant à elle avait finalement rejoint Adrien sur la gestion des nœuds de gestion, après avoir réalisé que cette partie nécessitait une charge de travail plus importante que ce que nous pensions.

Pour cette deuxième soutenance, nous n'avons pas changé la répartition des tâches. Pierre est resté sur la partie minage et l'a terminée, et va maintenant rejoindre Thimot pour l'aider. Ce dernier a continué son travail sur la partie réseau, qui fonctionne en partie mais souffre d'un très grand manque de stabilité et de nombreux bugs. Margaux a avancé sur la partie nœud de gestion et va pouvoir commencer à tester la communication en réseau. Adrien a lui aussi travaillé sur le nœud de gestion mais s'est également penché sur le site web. Hugo, lui, n'a pas avancé, néanmoins il prévoit, pour la prochaine soutenance, de réaliser une API pour permettre à quiconque d'utiliser le réseau en temps réel.

	Réseau	Noeud de minage	Noeud de gestion	Site web
Margaux			⊕	
Pierre		⊕		
Adrien			⊕	⊕
Hugo				
Thimot	⊕			

Tableau indiquant sur quelles parties du projet chaque membre a travaillé depuis la première soutenance

Noeud de gestion

Pour la première soutenance, nous avions commencé par définir deux nouvelles structures : BLOCK et TRANSACTION.

Une structure **TRANSACTION** possède trois variables : sender, receiver et amount.

- “sender” est une chaîne de caractères, correspondant à l’identifiant (le nom) de la personne envoyant l’argent de la transaction.
- “receiver” est une chaîne de caractères, correspondant à l’identifiant (le nom) de la personne recevant l’argent de la transaction.
- “amount” est un entier, correspondant au montant de la transaction.

La taille d'une chaîne de caractères contenant un identifiant (sender et/ou receiver) a été fixée à 20. Plus tard, l’identifiant ne sera plus simplement un nom, mais la valeur de hachage de l’identifiant de l’utilisateur.

A ce moment-là, le montant maximum d’une transaction avait été fixé à la taille maximum d’un “integer” sur 32 bits, c’est-à-dire $2^{31} - 1$ tokens.

Une structure **BLOCK** possède trois variables : previusHash, transactions et blockHash.

- “previusHash” est une chaîne d’octets, qui contient la valeur de hachage du bloc précédent.
- “transactions” est une liste de TRANSACTION.
- “blockHash” est une chaîne d’octets, qui contient la valeur de hachage du bloc actuel.

Nous souhaitions utiliser la structure BLOCK afin de générer les blocs de notre blockchain. Pour la première soutenance, nous avions fixé le nombre de transactions par bloc à dix. Nous pensions changer ce système plus tard, pour qu’un bloc ne soit pas créé toutes les dix transactions, mais toutes les x minutes.

Par la suite, nous avions travaillé sur les fonctions de hachage. Pour implémenter ces différentes fonctions il nous avait fallu décider de la méthode de hachage que nous souhaitions utiliser. Nous avions opté pour la plus connue : le sha256, qui avait en plus l’avantage d’être sécurisé.

Nous avions récupéré une librairie créée par Brad Conte sur internet, à l'aide de laquelle nous avions donc créé notre propre fonction de hachage.
La fonction, sha256, prend une chaîne de caractère et un buffer en paramètre. La chaîne de caractère correspond au texte à hacher, et c'est dans le buffer que la valeur de hachage du texte sera stockée. Le buffer, tout comme la valeur de hachage trouvée, a une taille prédéfinie, qui sera constante quelque soit la taille du texte. Dans notre cas, cette constante, nommée SHA256_BLOCK_SIZE, est de 32 octets.
Le texte passé en paramètre ne peut qu'être une chaîne de caractères contenant des caractères ASCII strictement supérieurs à zéro.

La première fonction, très courte, est ***txsToString***. Elle prend une TRANSACTION et un buffer en paramètre, et transforme les trois variables de la structure en chaîne de caractère, afin de permettre, plus tard, de les envoyer à nos autres fonctions. La chaîne de caractère est stockée dans le buffer donné en paramètre, qui doit avoir une taille suffisante pour la stocker.

La deuxième fonction est ***getMerkleHash***, qui prend en paramètre une structure BLOCK, et une liste d'octet de taille SHA256_BLOCK_SIZE.
Les transactions du bloc sont d'abord transformées en chaînes de caractères grâce à *txsToString*. Ces chaînes sont ensuite concaténées, et le tout est haché avec sha256.
De cette façon, il est presque impossible que deux valeurs de hachage soient identiques.
Le hash calculé, qu'on appelle le merkleHash, est stocké dans le buffer donné en paramètre. Cette valeur ne correspond pas à la valeur de hachage finale de notre bloc, mais elle sera utilisée pour calculer cette dernière.

Pour finir, la fonction qui nous permet de calculer le hash définitif d'un bloc est ***getHash***, qui prend en paramètre un BLOCK et une liste d'octets de taille SHA256_BLOCK_SIZE, tout comme *getMerkleHash*.
getHash récupère le hash du bloc précédent (stocké dans previusHash), puis calcule le merkleHash du bloc actuel. Pour finir elle concatène ces deux hashs, applique la fonction sha256 sur cette nouvelle chaîne, et stocke le résultat obtenu, qui correspond au Hash final du bloc actuel, dans la chaîne d'octets donnée en paramètre.
Cependant, nous avions rencontré un problème auquel nous ne nous attendions pas en implémentant cette fonction.

La fonction sha256 permet de générer une chaîne d'octets compris entre 0 et 255.
En hexadécimal, chaque octet contient deux caractères hexadécimaux (de 0 à F).
Si on dédouble chaque octet de la chaîne pour avoir sa représentation hexadécimale on aura donc chaque octet compris entre 0 et 15. Or, comme on l'a dit précédemment la chaîne de caractères que prend la fonction sha256 en paramètre ne peut pas contenir de caractère égal à zéro.
Pour palier ce problème nous avions donc dû implémenter une nouvelle fonction : sha256ToAscii. Celle-ci prend en paramètre une chaîne d'octets et un buffer, et stocke dans le buffer la valeur ASCII d'un hash, c'est-à-dire qu'elle transforme chaque octet de la liste en sa correspondance en ASCII.

Par la suite, nous avions créé la structure **BLOCKCHAIN**, qui possédait au moment de la première soutenance deux variables :

- "blocks", qui pointe vers le premier BLOCK de la blockchain.
- "blocksNumber", un entier indiquant le nombre de blocs contenus par la blockchain.

Nous savions que nous aurions besoin d'ajouter de nouvelles variables à l'avenir.
Afin de créer et gérer la blockchain, nous avions implémenté quatre nouvelles fonctions.

Il nous fallait tout d'abord pouvoir récupérer le bloc le plus récent d'une blockchain, alors nous avons créé ***getLastBlock***, qui prend une BLOCKCHAIN en paramètre et retourne son tout dernier bloc.

Ensuite nous avions créé ***addBlock***, une fonction plus longue qui nous permet d'ajouter un bloc à une blockchain, tous deux passés en paramètre, en utilisant notamment *getLastBlock* afin de remplir la variable *previusHash* de notre nouveau bloc.

Par la suite, pour pouvoir initialiser notre blockchain, il nous avait fallu créer la fonction ***createGenesis***, qui initialise le genesis, c'est-à-dire le premier bloc de la blockchain, la sentinelle, qui permet d'initialiser la liste.
Ce bloc ne contient donc pas de transaction, mais il faut quand même lui attribuer une valeur de hachage pour que le bloc suivant puisse renseigner ce hash dans *previusHash*. Pour cela nous avions appliqué sha256 à une chaîne de caractère choisie arbitrairement. Les genesis de toutes nos blockchains auront donc le même hash, mais cela ne pose pas de problème. Enfin, nous avions implémenté ***initBlockchain***, qui utilise les trois fonctions précédentes pour créer une nouvelle blockchain, et initialiser sa sentinelle.

Pour cette deuxième soutenance, nous avons décidé de changer le fonctionnement de nos blocs de transactions. En effet, jusqu'alors un BLOCK était composé d'une variable transactions (une liste de structures TRANSACTION), qui avait donc une taille statique. Cette organisation n'a posé aucun problème au début de l'implémentation de notre système de blockchain, mais nous avons finalement décidé qu'il fallait la revoir, puisqu'elle n'était pas parfaitement adaptée à nos besoins.
Nous avons ainsi décidé de créer une nouvelle structure : **TRANSACTIONS_LIST**. Celle-ci possède trois variables :

- "transactions", qui pointe vers la première structure TRANSACTION de la liste,
- "size", un entier indiquant le nombre de structures TRANSACTION dans la liste,
- "capacity", un entier indiquant le nombre de places allouées.

Pour coïncider avec ce nouveau fonctionnement, nous avons changé la variable "transactions" des structures BLOCK : c'est maintenant une TRANSACTIONS_LIST.

Grâce à ce nouveau système, nous n'avons pas à définir une taille fixe pour nos groupes de transactions, nous pouvons donc dorénavant envoyer dans la blockchain des groupes de transactions de tailles différentes.

Une fois l'architecture de nos groupes de transactions modifiée, nous avons commencé à implémenter de nouvelles fonctions.

Tout d'abord nous avons créé ***initListTxs*** qui ne prend pas de paramètre. Cette dernière initialise une liste de transactions, et assigne aux variables *transactions*, *size* et *capacity* les valeurs NULL, 0 et 1 respectivement. Elle alloue également la place nécessaire pour accueillir 1 transaction. La fonction renvoie ensuite la TRANSACTIONS_LIST qui vient d'être créée.

Ensuite, nous avons implémenté ***addTx***. Cette fonction a pour but d'ajouter une transaction à notre liste de transactions, qui toutes deux passées en paramètre.
Si jamais la liste est pleine (c'est-à-dire si les variables *capacity* et *size* sont égales), et donc ne peut pas accueillir une transaction supplémentaire, alors l'espace dédié à la liste est réalloué. Ensuite, la nouvelle transaction est ajoutée à la suite des autres.

Pour finir, nous avons créé une dernière fonction, ***clearTxsList***, qui nous permet de supprimer toutes les transactions contenues dans une liste dont le pointeur est donné en paramètre. Après être passé par la fonction, le pointeur indiquant l'emplacement de la liste

pointe vers une TRANSACTIONS_LIST vierge, c'est-à-dire dont les variables *transactions*, *size* et *capacity* valent respectivement NULL, 0 et 1.

Pour améliorer la sûreté et la fiabilité de notre blockchain, il nous fallait par la suite implémenter une fonction permettant de vérifier, à tout instant, que la blockchain n'a pas été corrompue, et qu'aucun problème n'est survenu durant l'ajout d'un groupe de transactions par exemple.
Pour cela, nous avons créé ***checkBlockchain***, qui prend en paramètre la blockchain que l'on souhaite vérifier. Pour chaque bloc contenant des transactions, deux vérifications vont être effectuées :

- Tout d'abord on vérifie que la variable *previusHash* de chaque bloc est bien égale à la variable *blockHash* du bloc le précédent dans la chaîne. De cette façon, on vérifie que les blocs sont bien "attachés" les uns aux autres dans le bon ordre.
- Dans un même temps, et pour plus de sécurité, on recalcule la valeur de hachage de chacun des blocs, afin de vérifier qu'aucune modification n'a été apportée à ces derniers. On vient donc ensuite vérifier que la valeur *blockHash* de chacun des blocs est bien égale à son hash, que l'on vient de recalculer.

Pour l'instant, pour calculer la valeur de hachage d'un bloc nous faisons appel à *getHash*, qui réalise une opération sur le *merkleHash* du bloc actuel, concaténé au hash du bloc précédent. Plus tard, nous modifierons notre fonction *getHash* pour qu'elle applique le cryptage sur une string non plus composée seulement du *merkleHash* et du hash du bloc précédent, mais aussi de la preuve de travail (cf. Noeuds de minage).

Dans un autre temps, nous avons commencé à essayer de relier site internet et nœuds de gestion, dans l'optique de pouvoir plus tard consulter et effectuer des transactions dans la blockchain depuis le site.
La première difficulté à laquelle nous avons été confrontés a été "d'envoyer" notre blockchain à notre site internet, puisque envoyer la structure en elle-même serait inefficace, car elle serait illisible par le site. Il nous a donc fallu trouver un moyen de mettre en forme la blockchain.

Nous avons décidé d'utiliser le format json, puisqu'il est simple de le décoder depuis un serveur en Javascript (node.js).
Ainsi, nous avons créé trois fonctions relatives à la conversion de la blockchain :

Tout d'abord ***txsToJson***, qui prend une TRANSACTION en paramètre, et renvoie sa valeur sous forme de chaîne de caractère en format Json.

Ensuite, ***blockToJson***, qui, en faisant appel à *txsToJson*, prend un BLOCK en paramètre, et renvoie sa valeur sous forme de chaîne de caractère en format Json.

Pour finir, ***blockchainToJson*** fait appel aux deux fonctions précédentes, et permet de convertir l'entièreté de la blockchain en un format Json lisible par notre site internet.

Noeud de minage

Le nœud de minage manipule le sha256 afin de hacher les blocs tout en générant une preuve. L'utilisation du sha256 permet d'avoir des hachages sécurisés de taille constante. La fonction de hachage nous permet dans un premier temps de créer les nouveaux blocs de cette manière :

$$sha256(\ hash\ bloc\ précédent\ +\ index\ +\ transaction)\ =\ nouveau\ hash.$$

Néanmoins, ce système peut se trouver être trop rapide car hacher un bloc ne prend pas particulièrement beaucoup de temps. Or nous voulons éviter la double dépense, car si le hachage des blocs est trop rapide un utilisateur peut se permettre de dépenser deux fois un même jeton.
Pour le minage, nous devons donc fournir un code capable de hacher un bloc tout en générant une preuve associée, afin de ralentir la création de nouveaux blocs.
Pour cela, une fonction brute force hache en boucle jusqu'à ce qu'une preuve soit valide.
Le principe est le suivant pour chaque bloc :

$$sha256(\ hash\ bloc\ précédent\ +\ index\ +\ transaction\ +\ preuve)\ =\ nouveau\ hash.$$

Ceci représente un système avec une inconnue et une pseudo inconnue. "preuve" est l'inconnue qu'il nous faut trouver, le problème étant qu'on ne connaît pas "nouveau hash" non plus. Nous sommes donc obligés de fixer une spécificité à "nouveau hash".
Par exemple, dans notre implémentation, on demande une preuve telle que le nouveau hash finisse par "0".
Ainsi, il nous faut tester de nombreuses preuves jusqu'à en trouver une validant le critère. La sortie d'une fonction de hachage étant presque impossible à anticiper, la preuve doit être calculée par un algorithme de brute force.

Les fonctions de hachage sont aussi capables de s'aligner à un indice de difficulté. Plus cet indice est élevé, plus la preuve sera compliquée à trouver. Ceci permet d'éviter au réseau de calculer trop vite les blocs, et donc d'avoir des preuves trop faciles à produire.
La méthode la plus simple pour augmenter la difficulté est de rendre les critères de validité du nouveau hash encore plus spécifiques. Par exemple, si le critère de validité est

que le nouveau hachage doit terminer par un zéro, on peut augmenter la difficulté en augmentant le nombre de zéros.

Le but du minage est aussi de ralentir le réseau lors de la génération des nouveaux blocs, afin d'empêcher les doubles dépenses. En effet, si ce système n'est pas implémenté alors un client pourrait dépenser un jeton à deux personnes en même temps. Néanmoins, l'obligation de créer une preuve et un hash assez long à calculer empêche de créer facilement plusieurs blocs différents dans lequel on dépenserait plusieurs fois le même jeton.

Lors de la dernière soutenance il avait été convenu de régler et d'implémenter les points suivants :

- Le réglage de la difficulté était trop abrupt, et un meilleur système était envisageable.
- Le minage ne se faisait que sur un seul thread, ce qui posait des problèmes d'optimisation.
- La fonction de hachage acceptait certains arguments spécifiques, néanmoins elle pouvait causer certains problème de conversions sur des arguments plus généraux
- Le nœud minage se devait évidemment de communiquer avec le réseau : nous pensons pour l'instant recevoir et envoyer les blocs à miner sous forme de chaînes de caractères sur des file descriptors, mais nous ne sommes pas encore certains de la manière dont les noeuds de minage vont communiquer avec les noeuds de gestion.

Pour la première soutenance nous avions donc implémenté un minage simple, mais non pas dénué de défauts. Le fonctionnement du nœud a été modifié, le code a été adapté aux demandes et exigences de la mise en réseau et du multi-threading, et les erreurs les plus évidentes ont été debugées.

Tout d'abord, la génération des preuves a été modifiée. En effet, nous avons retenu une autre méthode pour générer les hashs et les preuves. Cette méthode utilise

previushash et *merkleHash* (cf. Noeud de gestion), et calcule les preuves avec cette nouvelle formule :

$$sha256(preuve + previushash + Merklehash) = nouveau\ hash.$$

La validation des preuves est toujours la même, c'est à dire qu'on demande un nouveau hash finissant par n zéros, n correspondant à la difficulté du hash, puis on recherche une preuve valide grâce à un minage brute force.
La fonction de minage a néanmoins été grandement améliorée, puisqu'elle utilise maintenant plusieurs threads afin de multiplier la puissance de calcul.

Comme expliqué précédemment, le but du minage est de ralentir le réseau afin d'éviter les problèmes de double dépense par exemple. Implémenter un multi-thread peut donc sembler contre productif, puisqu'il améliore la vitesse du minage. Néanmoins, il faut garder à l'esprit que les utilisateurs peuvent coder eux-même leurs fonctions de minage, donc si le système est prévu pour fonctionner avec des fonctions plus lentes, quelqu'un utilisant le multithread aura un grand avantage sur les autres.

A la première soutenance nous avions jugé la gestion du niveau de difficulté trop abrupte, mais nous avons tout de même décidé de garder celle-ci, puisque rendre le système plus précis s'est révélé inutile. Nous avons besoin que la durée de calcul suive un ordre de grandeur, et non pas forcément une durée précise.

Il peut demeurer des bugs que nous ne connaissons pas encore, mais nous considérons que le nœud de minage est terminé. Il ne manque plus qu'à le mettre en réseau avec la partie de gestion.

Site web

Pour la première soutenance, notre site internet était très simple. Pour comprendre le fonctionnement de l'HTML et du CSS utilisés, nous avions suivi un cours sur OpenClassroom qui nous avait permis de comprendre les fondements de ces langages de balisage.

Nous utilisions le HTML pour gérer la structure, le texte, les images, les téléchargement et les changements de pages. Le CSS quant à lui nous permettait d'ajouter des menus déroulants, ajuster les couleurs, la taille et l'emplacement de la police...
Nous avions codé nous même la partie en HTML5, en suivant les instructions données par le cahier des charges sur ce que devait contenir le site. Nous avions téléchargé la mise en forme CSS sur ce site : https://html5up.net.

A ce moment-là, le site ne contenait qu'une page d'accueil faisant une présentation des actualités concernant notre projet, un onglet "Notre projet" dans lequel nous présentions notre idée, ses origines, tout en présentant les fondateurs des premières blockchains. Nous avions ensuite implémenté un onglet "Avancement", qui visait à présenter l'évolution de notre projet à chaque soutenance. Finalement, un onglet "Téléchargement" était disponible, dirigeant vers une page rappelant les consignes du projet, le cahier des charges, et la version finale du projet en lui-même.

Notre objectif était de réussir à lier, plus tard, le site et la blockchain, afin de pouvoir réaliser des transactions directement depuis le site.

Nous travaillions également sur un système de gestion de comptes utilisant PHP, HTML et MySQL, l'objectif étant de permettre aux utilisateurs de créer un compte sur le site, pour utiliser la blockchain depuis celui-ci.
Pour cela, il nous a fallu créer une page où les clients peuvent s'inscrire avec leur adresse mail en créant un mot de passe. Ensuite, nous savions qu'il nous faudrait utiliser une base de données pour stocker les données de connexion des utilisateurs, afin qu'ils puissent se reconnecter. Nous avions décidé d'utiliser MySQL, qui a l'avantage d'être gratuit et assez simple d'utilisation. De plus, une grande quantité de tutoriels est disponible sur internet.

Pour relier les informations fournies par les utilisateurs et la base de données nous utilisions des fichiers codés en PHP.. Nous n'avions pas encore implémenté ces différentes fonctionnalités lors de la première soutenance, car il nous fallait un serveur capable de lire le PHP, chose sur laquelle nous travaillions encore.

Nous avions prévu au départ d'utiliser GitHub, qui permet de publier un site web, mais nous avions réalisé que cela ne fonctionne qu'avec du HTML, c'est-à-dire des sites statiques, chose qui ne nous convenait pas. Nous utiliserons donc un serveur privé, ce qui nous permettra d'implémenter le code en PHP. Ensuite nous aurons à mettre en relation notre blockchain et le site web.

Pour la deuxième soutenance, nous avons décidé de changer complètement la conception du site, et recommencer depuis le début. Il nous fallait utiliser un système plus efficace afin de coder les différentes parties de notre site, telles que la création de compte ou la gestion des connexions au réseau de la blockchain.
Nous savions déjà qu'il nous serait impossible d'utiliser Github, qui ne permet pas de mettre en place un site dynamique en PHP. Nous avons donc transféré le site sur un serveur nous le permettant. Sur ce dernier, nous avons installé MySQL, un système de gestion de base de données, et Node.js, un environnement JavaScript bas niveau qui permet d'exécuter du code côté serveur.

Tout d'abord, nous avons implémenté le système de comptes.
Lors de la création d'un nouveau compte client, l'identifiant et le mot de passe choisis par l'utilisateur sont enregistrés dans la base de données MySQL, en plus de deux clés RSA (publique et privée), qui permettent, lorsqu'on les couple, d'encoder et de décoder des messages. Plus tard, lorsqu'une transaction sera créée, un message crypté sera ajouté à la transaction, et fera office de "signature". En effet, pour décoder le message en question et donc valider la transaction, il faudra utiliser les clés RSA de l'envoyeur, ce qui empêchera les utilisateurs de réaliser des transactions sous le nom de quelqu'un d'autre.

Plus tard, nous souhaitons ajouter une page qui permettra d'afficher certaines données de la blockchain, parmi lesquelles les transactions temporaires. Pour ce faire,

nous allons devoir modifier un des nœuds de gestion, afin qu'il puisse recevoir des requêtes depuis le site, et ainsi lui envoyer les informations qu'il demande.

Nous voulons également, pour la dernière soutenance, mettre en place une page permettant aux utilisateurs de réaliser des transactions. Chaque utilisateur pourra consulter son solde depuis son compte client.

Réseau

Dans cette partie, nous allons expliquer comment nous avons pensé le réseau, comment celui-ci fonctionne, et ses possibles évolutions.
Nous avons décidé de réduire les communications entre les serveurs à trois choses. Ils peuvent échanger leurs connaissances du réseau, les transactions et la blockchain. Ce simple échange autour de ces trois données permet de poser les bases de la décentralisation. Le principe théorique peut s'avérer plutôt simple mais l'implémentation est beaucoup plus complexe.

Pour simplifier la communication entre le serveur et la partie gestion, nous avons décidé que la communication entre le nœud de gestion et la gestion du réseau doit se faire au moyen de deux files descriptors, un pour la lecture et l'autre pour l'écriture. Cela permet de réduire grandement la complexité du côté du nœud de gestion. Le nœud reçoit tous les messages sans savoir de qui ils proviennent. Inversement lorsque l'on souhaite envoyer un message, celui-ci est envoyé à tous les autres serveurs connus sans distinction.

Le réseau s'occupera de recevoir et envoyer les connaissances du réseau, le reste sera à la charge du nœud de gestion. Il doit donc se charger d'agrandir le réseau connu en partageant ses connexions avec les autres serveurs et inversement très régulièrement. Si nous avons le temps nous pourrons essayer de mettre un place un système de ping pour savoir si les serveurs sont toujours en ligne.

Le réseau repose sur une liste de clients connectés. Pour chaque client, nous enregistrons plusieurs informations, telle que l'adresse IP, son statut (en cours de connexion, connecté, terminé, erreur...) ou encore les files descriptors sur lequel nous pouvons le contacter.
Cette liste originellement vide est remplie au démarrage par une liste de serveurs auxquels nous devons nous connecter. Par la suite, les nouveaux clients sont ajoutés à cette liste et supprimés s'ils se déconnectent. Elle permet de connaître à tout moment le statut et diverses informations sur les clients.

Suite à la première soutenance, plusieurs modifications ont été apportées. La principale étant le passage d'une liste contiguë à une liste chainée pour la gestion des serveurs connectés. En effet, une liste chaînée permet de ne pas avoir à réallouer et donc bloquer tous les clients durant la réallocation. Ceci apporte en stabilité et en performances. La création d'un client ne peut impacter que très légèrement le dernier serveur de la liste, lors de la modification d'un pointeur sur le dernier élément de la liste La suppression d'un client apporte le même gain, il ne faut modifier que deux pointeurs.

Pour continuer, la fonction principale est séparée en deux threads majeurs. L'un pour accepter les connexions entrantes, et l'autre pour maintenir ces connexions ouvertes et transmettre l'information.

Le premier agit de manière classique, il accepte toutes les connexions entrantes sur un port précis. Il enregistre ensuite le client sur la liste avec l'état "connexion en cours".
Le thread chargé de maintenir les connexions ouvertes trouve tous les clients "en cours de connexion", et réalise un thread pour chaque client. Au sein de celui-ci, deux threads sont générés pour pouvoir lire et écrire en permanence de manière asynchrone. L'état du client est alors passé à "connecté".
Ces nombreux threads vont surement être sujet à modifications. En effet, il est fort probable qu'à terme nous n'ayons plus l'utilité de certains.

Nous avons également implémenté l'envoi et la réception de messages. Ceux-ci sont divisés en deux parties : l'entête et les données.
L'entête est composée de neuf octets. Le premier est le type de message, qui indique ce qu'il contient. Les huit autres sont un long entier non signé, correspondant à la taille, en octets, des données à lire.
Il n'y pas de taille maximum théorique pour la taille de la partie donnée d'un message, mais nous serons peut-être amenés à implémenter une taille maximum, et donc segmenter la partie des données, pour augmenter la stabilité des communications.

La gestion du réseau côté serveur est en passe d'avoir sa première version terminée. En effet, une grande partie a été terminée, et la correction des bugs est en cours.

Nous avons eu des difficultés à mettre en place cette première version, car il nous a fallu utiliser toutes les notions que nous avons récemment apprises (fork, thread, mutex...). De plus, le correction des bugs risque d'être assez ardue car le processus lance plusieurs threads. Néanmoins, les premiers tests nous montrent que l'algorithme en lui-même semble fonctionner correctement.

Actuellement, le réseau fonctionne en partie entre deux ordinateurs. Néanmoins, dès que nous augmentons le nombre de serveurs dans le réseau, la stabilité est toute relative. En effet, les serveurs ne plantent pas, mais de nombreuses déconnexions non voulues surviennent régulièrement. Nous avons également des problèmes de connexions "fantômes" : certains serveurs considèrent avoir plus de connexions qu'il n'y a d'ordinateurs sur le réseau.
Nous avons également un problème avec la transmission des messages : si ceux-ci dépassent une certaine taille, le client va se déconnecter pour une raison inconnue.

Pour conclure, notre but pour la dernière soutenance sera de corriger ces problèmes, et donc d'améliorer la stabilité du réseau. Éventuellement, si nous avons le temps, nous souhaiterions également pouvoir résoudre des noms de domaines pour que le réseau puisse fonctionner en dehors d'un réseau local.

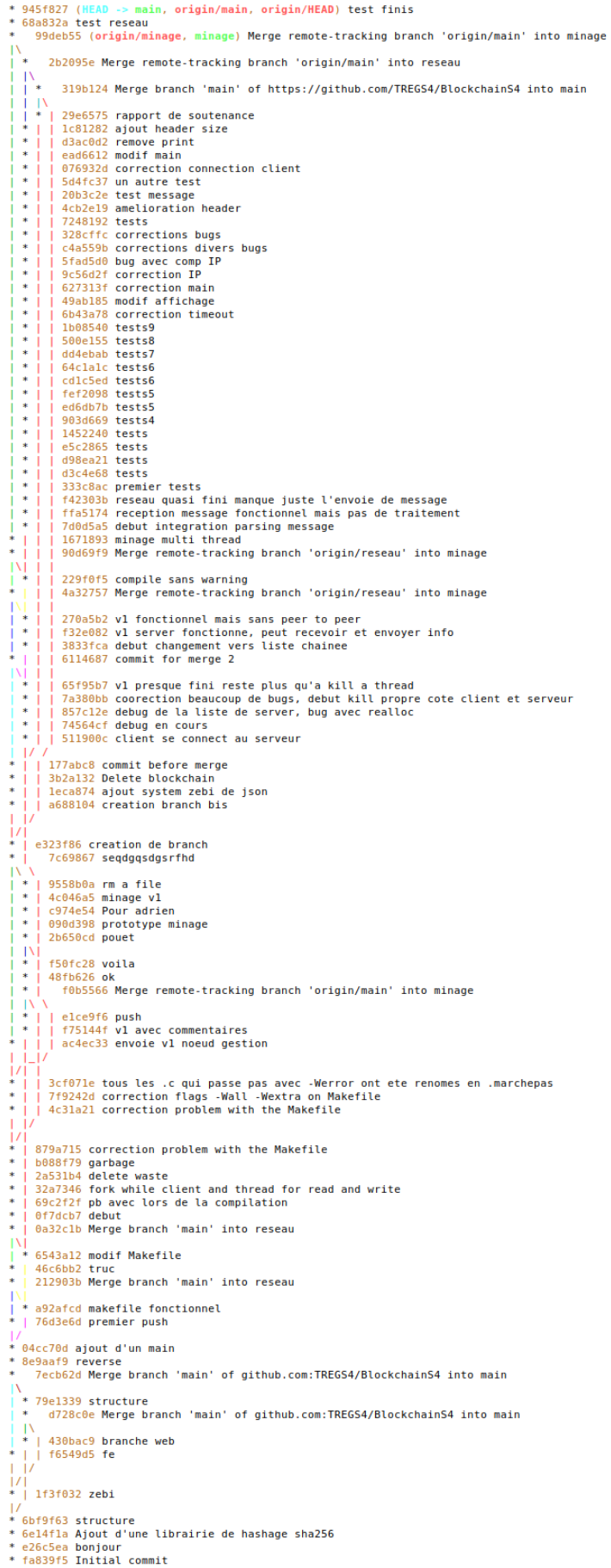
Conclusion

Lorsque nous avons choisi de recréer une blockchain décentralisée, nous ne connaissions que vaguement les principes techniques utilisés pour la faire fonctionner. Depuis la validation du cahier des charges nous avons appris beaucoup, néanmoins ce sujet implique des notions simples, mais difficiles à assimiler.

Depuis la dernière soutenance, le projet a bien avancé. Bien que certaines notions demeurent floues, nos connaissances quant à la blockchain et son fonctionnement ne cessent de croître. Les objectifs que nous nous étions fixés ont été respectés, à l'exception de ceux concernant le site, puisque l'interface de visualisations des transactions en cours n'a pas encore été implémentée. En revanche, nous avons pris de l'avance sur la mise en forme du site et son esthétique, ainsi que sur la création des comptes.

Il nous reste peu de choses à implémenter pour la soutenance finale :
Après avoir réussi à mettre en réseau les nœuds de gestion, il nous faudra rendre notre blockchain directement utilisable depuis le site internet en créant une API.
Néanmoins, nous savons que la mise en commun de tous les éléments mettra en lumière de nombreux problèmes qui sont déjà existants, mais que nous ne pouvons pas identifier tant que tous les éléments ne sont pas connectés.
Nous sommes confiants quant à la suite du déroulement de notre projet, et sommes déterminés à rendre un projet dont nous serons fiers.

Annexes



Historique des commits sur notre répertoire de travail