2020/2021

Projet S4

Rapport de projet

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, et constitueront probablement les échanges monétaires de demain. En développant un réseau en peer to peer, c'est-à-dire sans serveur central, 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.

#crypto-monnaie #blockchain #décentralisation #transactions #token

Since more than ten years, the design and use of cryptocurrencies has experienced a great boom. These currencies, which are not controlled by any institution, are now accepted as a payment method, and will probably constitute tomorrow's monetary exchanges. By developing a peer to peer network, i.e. without a central server, and a blockchain, TREG has created its own cryptocurrency, the TEK.
This network, developed in C language, is accessible via a web interface in Javascript, using the Node.js environment.

#cryptocurrency #blockchain # decentralization #transactions #token

Cahier des charges

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 était 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

<u>Première soutenance</u>

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

Dernière soutenance

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 changé quelque peu la répartition des tâches. Pierre s'était finalement positionné sur la gestion des nœuds de minage après de l'annulation de son départ à l'international. Margaux quant à elle avait 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 la deuxième soutenance, Pierre était resté sur la partie minage et l'avait terminée. Thimot avait continué son travail sur la partie réseau, qui fonctionnait en partie mais souffrait d'un très grand manque de stabilité et de nombreux bugs. Margaux avait avancé sur la partie nœud de gestion. Adrien avait lui aussi travaillé sur le nœud de gestion, mais s'était également penché sur le site web. Hugo, lui, n'avait pas avancé.

Pour la version finale du projet, la répartition des tâches n'a que très peu évolué. Thimot a entièrement reconstruit le réseau, qui était trop complexe et trop peu fiable. Pierre a continué de travailler sur le minage pour le rendre compatible avec les nœuds de gestion. Il a également réglé certains bugs que nous avons découvert au fur et à mesure. Le nœud de gestion étant terminé, Margaux a assisté Adrien sur l'interface et la partie interne du site web. Hugo, lui, n'a pas avancé.

	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

Noeud de gestion

Les nœuds de gestion sont les tâches principales (les threads principaux) de notre système : sans ceux-ci, le programme ne pourrait faire partie d'aucun réseau d'une blockchain.

Chaque noeud a différents buts :

- Gérer la communication entre les différents nœuds d'un même réseau d'une blockchain. Chaque nœud de gestion envoie les informations qu'il possède, c'est-à-dire sa version des transactions non validées et de la blockchain.
  Il reçoit ces mêmes informations de la part des autres nœuds du réseau.

- Vérifier les informations reçues. Chaque nœud de gestion doit vérifier les informations qu'il reçoit pour savoir si elles sont obsolètes, ou bien erronées.

- Lancer les tâches secondaires du système. Chaque nœud de gestion s'occupe de lancer les tâches (threads) de minage et d'API si le nœud en question le souhaite.

Avant de se lancer dans la création du programme principal du nœud de gestion, nous avions besoin de mettre en place différentes choses, dont l'implémentation de la structure d'une blockchain.

Nous avons donc commencé par définir deux structures : BLOCK et TRANSACTION.

Une structure **TRANSACTION** représente une transaction dans la blockchain. Elle possède quatre variables : sender, receiver, amount et time.

- "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.
- "time" est un entier, correspondant à l'horodatage ("timestamp") de la création de la transaction.

Dans un premier temps, la taille des chaînes de caractères du "sender" et du "receiver" avait été fixée à 20 octets. Finalement, ces chaînes de caractères n'ont plus de tailles fixes

mais des tailles dynamiques, vu que les variables “sender” et “receiver” correspondent maintenant à des clés publiques RSA.
Le montant maximum d’une transaction a été fixé à la taille maximum d’un entier positif sur 32 bits, c’est-à-dire $2^{32} - 1$ tokens.

Une structure **BLOCK** qui représente un bloc d’une blockchain. Elle possède cinq variables : previusHash, transactions, blockHash, proof et time.

- “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.
- proof est un entier, qui correspond à la preuve de travail du bloc (cf. Noeud de minage)
- “time” est un entier, correspondant à l’horodatage (“timestamp”) de la création du bloc.

Nous souhaitions utiliser la structure BLOCK afin de générer les blocs de notre blockchain.
Dans un premier temps, nous avions fixé le nombre de transactions par bloc à dix. Dorénavant, chaque bloc peut posséder autant de transactions que souhaité, ce qui nous permet, plutôt que d’envoyer un bloc à la partie de minage toutes les *x* transactions, de l'envoyer toutes les *x* secondes (s’il y a au moins une transaction).

Par la suite, nous avons travaillé sur les fonctions de hachage, qui sont essentielles à la création d’une blockchain. En effet, les blocs d’une blockchain sont liés par des valeurs de hachage. Pour implémenter ces différentes fonctions il nous a fallu décider de la méthode de hachage que nous souhaitions utiliser. Nous avons opté pour la plus connue : le sha256, qui avait en plus l’avantage d’être sécurisé.
Nous avons récupéré une librairie créée par Brad Conte sur internet, à l'aide de laquelle nous avons 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 avons 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 possède deux variables :

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

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 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 sont 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 calculer la valeur de hachage d'un bloc nous faisons appel à *getHash*, qui réalisait, dans un premier temps, une opération sur le *merkleHash* du bloc actuel, concaténé au hash du bloc précédent. Maintenant, cette fonction *getHash* 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).

Par la suite, nous avons entamé la création du programme principal du nœud de gestion. Celui-ci fonctionne de la manière suivante :
- Tout d'abord, il s'occupe d'initialiser deux structures: une blockchain vide et une liste de transactions non validées vide.
- Ensuite, vient le moment de lancer les tâches (threads) de minage et d'API, selon les demandes du propriétaire du nœud.
- Après cela, le nœud de gestion lance une dernière tâche (thread) qui va s'occuper, toutes les 10 secondes environ, d'envoyer au réseau la version de la blockchain et des transactions non validées que possède le programme.

Après ces trois phases, le nœud de gestion entre dans une nouvelle phase, dans laquelle il va vérifier constamment de un à trois points. Ces trois points sont les suivants :
- Le nœud a-t-il reçu une donnée du réseau de nœud ?

- Le nœud a-t-il reçu une transaction depuis l'API du programme (dans le cas où l'API est active) ?
- Le nœud a-t-il reçu un nouveau bloc de la partie de minage (dans le cas où le minage est actif) ?

Si l'une des trois questions précédentes est vérifiée, alors les données reçues sont acceptées. Si la donnée reçue est une blockchain, l'actuelle blockchain est remplacée par la nouvelle. Si la donnée reçue est une transaction, celle-ci est ajoutée dans les transactions non validées. Pour finir, si la donnée reçue est un bloc, celui-ci est ajouté à la suite de la blockchain actuelle.

Noeud de minage

L'objectif du nœud de minage est de créer les nouveaux blocs à partir des transactions en attente. Celui-ci va donc créer les nouveaux éléments nécessaires à la génération du nouveau bloc, à savoir son hash et la preuve associée.

Le nœud de minage manipule donc 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.

Cette première version du minage plus théorique a été modifiée afin de convenir à notre implémentation du réseau et à nos nœuds de gestion.

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.

Dans un premier temps 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.

Enfin il a fallu lier ce nœud de minage avec le reste du réseau.
Le nœud de minage a donc été mis en lien avec les nœuds de gestion car ce sont eux qui vont avoir besoin du minage pour générer les nouveaux blocs à partir des transactions en attente. Pour activer un nœud de minage il faut initialiser un nœud de gestion avec l'option "-m", le nœud de gestion va alors en plus de son utilisation habituelle lancer sur un nouveau thread le nœud de minage. Le nœud de minage va ensuite récupérer les informations à hacher sur une file commune comprenant les listes de transactions qui attendent d'être hachées. les hacher en générant la preuve puis le bloc trouvé va être renvoyé sur une file de blocs commune.
Le nœud de minage tourne donc en permanence à la recherche de nouvelle transaction pour pouvoir les hacher puis les renvoyer au nœud de gestion.

En conclusion le nœud de minage a donc été lié sans trop de problème au reste de la blockchain. Le fait d'avoir décidé à l'avance comment le nœud de réseau et de minage allaient interagir a permis de faciliter la tâche. Nous n'avons pas encore trouvé de bugs sur le projet causées par le minage ou par l'interaction entre le minage et le nœud de gestion.

Ce qui aurait pu être amélioré mais n'as pas été fait par manque de temps, est la gestion automatique de la difficultée du hachage.
Dans le projet, la difficulté est fixée à trois. Afin de pouvoir réaliser des tests, nous pouvons la régler manuellement en la passant en argument, néanmoins cela ne devrait pas être possible dans une version de production. La difficultée doit normalement s'auto réguler, c'est-à-dire que si les mineurs sont trop rapides on l'augmente, et s'ils sont trop lents on la baisse. Ce contrôle est censé être automatique, mais n'est pas effectué dans notre projet.

API

Afin de pouvoir créer une communication entre notre site web et notre réseau nous avons dû créer une API (ou Application Programming Interface), que l'on pourrait traduire par interface de programmation d'application.

Il y a deux objectifs principaux à notre API. Le premier est de recevoir des requêtes de transactions qui seront ensuite envoyées au nœud de gestion. et le dernier est d'envoyer des informations sur réseau, sur la blockchain et sur les transactions non validées.

L'API peut être lancée à l'initialisation de n'importe quel nœud de gestion, avec l'extension "-a". Celle-ci va être lancée sur un nouveau thread, va écouter en permanence sur son port, et répondra aux requêtes envoyées. (cf. Noeud de gestion)

Les réponses que renvoie l'API aux clients sont toutes renvoyées en format json. En effet la difficulté à laquelle nous avons été confrontés a été "d'envoyer" nos structures à notre site internet, puisque envoyer les structures en elles-mêmes serait inefficace, car elles seraient illisibles par le site. Il nous a donc fallu trouver un moyen de mettre en forme ces structures.
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 quatres fonctions relatives à la conversion de ces structures:

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.

Puis, ***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.

Pour finir, ***ServerListToJSON*** convertit la liste des nœuds connus en une liste au format JSON, devant cette liste le nombre de nœuds connus est indiqué.

Les différentes requêtes possibles sont envoyées sous le format :

http:/host:2048/ressource.

Si la requête est valide on répond :

"HTTP/1.1200OK\nAccess-Control-Allow-Origin:\r\n\r\n".*

Ce qui permet à n'importe qui d'accéder à l'API ensuite on envoie les informations demandées par la requête ou la validation de l'opération.

Pour connaître la nature de la requête nous traitons la ressource obtenue.

- Si la ressource est "transactions/get" alors on renvoie la liste des transactions non validées.
- Si la ressource est "transactions/post" alors l'API va envoyer la nouvelle transaction au noeud de gestion (cf. Noeud de gestion).
- Si la ressource est "server/count" alors on renvoie le nombre de serveurs et la liste de tous les serveurs avec pour chacun leur hostname leur port et leur statut c'est à dire si ce sont des mineurs et des API en plus d'être des noeuds de gestion.
- Si la ressource est "blockchain" alors on renvoie la blockchain.

Si la requête n'est pas valide, on répond :

*"HTTP/1.1 404 Not Found\nAccess-Control-Allow-Origin: *\r\n\r\n"*

Réseau

Notre réseau est un réseau en pair à pair, son rôle est de permettre au nœud de gestion de communiquer avec tous les autres nœuds du réseau. Il est chargé de connaître les autres membres du réseau et partager sa connaissance du réseau avec les autres. Cette séparation permet de simplifier et compartimenter les différentes parties du nœud de gestion.

Dans cette partie, nous allons donc expliquer comment nous l'avons pensé, 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. Néanmoins, le réseau est capable de transmettre n'importe quel type de données. Le principe théorique peut s'avérer plutôt simple mais l'implémentation est beaucoup plus complexe.

Première version du réseau

Dans cette première partie, la première version du réseau est détaillée. Néanmoins au vu de sa complexité et de son manque de stabilité, nous avons décidé de repartir de zéro. Seules les idées d'une liste chaînée contenant tous les serveurs connus et la structure binaire des messages ont été gardées.

Le réseau utilise et travaille sur une liste de nœuds connus. Il peut la modifier au cours du temps pour ajouter des nouveaux nœuds, et retirer ceux qui ne répondent plus. 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. Cette dernière permet de connaître à tout moment le statut et diverses informations sur les clients.

Après 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.

Deuxième version du réseau

Dans cette partie, nous allons décrire la deuxième version du réseau qui a été totalement revue pour drastiquement simplifier le fonctionnement et augmenter la stabilité.

Pour simplifier la communication entre le réseau et la partie gestion, nous avons décidé que celle-ci devait se faire au moyen de deux files : l'une pour l'envoi de message, et l'autre pour la réception. La partie réseau vérifie en permanence s'il n'y a pas un message dans la file d'envoi. Si c'est le cas, alors elle le retire et l'envoie. A l'inverse, lorsqu'un message arrive, il est envoyé dans la seconde file que la partie gestion se charge de vider.
A l'origine nous utilisions des files descriptors, mais nous avons rencontré trop de problèmes au moment de mettre en place ce mode de fonctionnement.

Séparer le nœud de gestion et la partie réseau nous a permis de réduire la complexité. En effet, ces deux parties sont totalement indépendantes. Le réseau est capable d'envoyer n'importe quel message, il peut être utilisé par un tout autre logiciel si besoin est. 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 s'occuper d'agrandir le réseau connu en partageant ses connexions avec les autres serveurs et inversement très régulièrement.

La connaissance des autres nœuds du réseau se fait au travers d'une liste chaînée. Chaque élément est un nœud connu, qui contient son adresse ip ou son nom de domaine, son port, et l'information disant si sa partie minage et/ou api est activée. Toutes ces informations sont stockées dans une structure ClientInfo (cf. Annexe réseau 2 et 3). Cette liste commence par une sentinelle et contient également le nœud sur lequel il est lancé.

La gestion du réseau est séparée en deux grands threads totalement distincts : la réception est l'envoi de messages.

La réception est un serveur acceptant toutes les connexions. Chaque nouvelle connexion entrante engendre la création d'un thread temporaire, cela permet d'améliorer grandement les performances et surtout de pas avoir un serveur limité à une seule connexion simultanée. Si le message reçu comporte la moindre erreur, celui est abandonné pour être sûr qu'il ne fasse pas planter le programme. Si le message entrant est correct, alors une structure MESSAGE est créée, et celle-ci est envoyée dans la file de réception si ce n'est pas un message destinée au réseau. Si c'est un message destiné au réseau, alors la fonction *AddServerFromMessage*() est appelée (celle-ci est détaillée plus loin).

L'envoi de message se base sur la liste des nœuds connus pour envoyer le message. A nouveau, pour une question de performance, un thread temporaire est créé pour pouvoir envoyer simultanément le message à tous les nœuds en même temps. Si un des nœuds n'est pas accessible, il est automatiquement retiré de la liste.

L'ajout de serveur se fait par une fonction qui parcourt les données en binaires du message et les convertit en structures. Si le nœud est déjà présent dans la liste, seul l'information sur le présence d'une API ou d'une partie minage est mise à jour, sinon le nœud est ajouté à la liste. Avant d'ajouter un élément, le programme tente de se connecter pour vérifier que le nœud en question est accessible et en ligne. Si la moindre erreur intervient, le nœud n'est pas ajouté.

Deux autres threads sont également lancés au démarrage du réseau. Le premier se charge d'envoyer toutes les deux secondes l'ensemble des nœuds connus à tous les nœuds connus. Le second est uniquement chargé de vider la file d'envoi de message et de les envoyer.

Pour simplifier la création des messages et l'envoi de messages, nous avons implémenté une structure (cf. Annexe réseau 1), contenant trois champs.
Le premier est un entier permettant de connaître le type de message (blockchain, transaction ou réseau). Le second est également un entier, donnant la taille du troisième élément qui est un tableau d'octets : les données brutes. Lors de l'envoi du message, cette structure est convertie en données binaires, puis envoyée à tous les autres nœuds.

Notre implémentation actuelle ne permet pas de savoir d'où provient le message. Elle ne permet pas non plus de spécifier un destinataire.

Toutes les informations liées au réseau sont accessibles depuis une structure server (cf. Annexe réseau 4). Cette structure doit être initialisée avant le lancement du réseau. Elle comprend les deux files permettant l'envoi et la réception de message, une liste chaînée de nœuds connus et son mutex, deux entiers permettant de savoir si l'instance est une api et/ou un nœud de minage. Enfin un entier est réservé au statut du réseau, ce qui

permet de savoir si celui-ci est correctement démarré, et permet aussi de l'arrêter proprement lorsque le statut est passé à "EXITING".

Pour terminer, nous allons voir l'initialisation d'un réseau. Pour cela, il faut appeler la fonction *Network()* qui prend en paramètre plusieurs choses, un pointeur vers une structure server qui doit être initialisée avec la fonction *InitServer()*. Le second paramètre est un pointeur vers un l'adresse ip ou le nom de domaine du nœud qui vient d'être lancé. Ce paramètre doit être valide et non null. Le troisième est le port du nœud. Le quatrième est l'adresse ip ou le nom de domaine d'un nœud du réseau auquel se connecter. Enfin, le dernier paramètre est le port du nœud sur le réseau. Hormis, les deux premiers paramètres, tous peuvent être nuls, pour les ports, celui par défaut sera alors utilisé. Si des paramètres n'est pas valide, le réseau s'arrête alors automatiquement.

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.

Nous avons par la suite 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.

Nous avions en premier lieu implémenté un 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.

Pour la version finale du site, nous avons implémenté un “Espace Client”, depuis lequel un utilisateur peut consulter son compte, réaliser une transaction, et se déconnecter.

- La page “Mon Compte” n’a pas encore été implémentée, mais nous prévoyons de permettre à l’utilisateur, depuis cette dernière, de changer son mot de passe, ainsi que son identifiant et consulter ses identifiants rsa.
- La page des transactions quant à elle est bien opérationnelle, et permet à l’utilisateur de réaliser des transactions. Le choix du montant est libre, et un menu déroulant affiche la liste des autres utilisateurs du site, c’est-à-dire les personnes à qui le virement peut être adressé.

  Depuis cet espace les utilisateurs peuvent également consulter leur solde en TEKs.
- Le dernier bouton permet simplement à l'utilisateur de se déconnecter.

Nous souhaitions, parallèlement, que les différentes transactions ajoutées à la blockchain puissent être visibles directement depuis une page du site. Pour cela nous avons créé une page “La blockchain”, accessible depuis l’accueil.

A gauche de la page, la liste des blocs de la blockchain est affichée. Les transactions sont regroupées selon le bloc dans lequel elles ont été minées, et le numéro de chaque bloc est affiché en haut de celui-ci.

A droite de la page nous avons décidé d’afficher la liste des nœuds du réseau. L’adresse ip du nœud est affichée, suivi de son port, et de deux icônes indiquant s’il s’agit d’un nœud de minage et/ou une d’une API.

La liste des transactions est actualisée toutes les secondes, ce qui permet de savoir si une transaction a été approuvée, ou bien si elle est encore en cours de validation. Dans les deux cas, l’heure à laquelle la requête de transaction a été faite est affichée.

Nous avons également créé une rubrique “Téléchargements” accessible depuis l’accueil du site, qui propose de télécharger le rapport de soutenance, ainsi que le projet en lui-même. Toujours sur la page d’accueil, on retrouve l’abstract de notre projet, ainsi que les mots-clés le définissant.

Pour finir, en termes d'implémentations secondaires nous avons créé un système de thème sombre et thème clair. Celui-ci s'adapte selon le système d'exploitation du client, mais peut être modifié depuis l'accueil du site.
Nous avons une grande préférence pour le thème sombre qui est très abouti et agréable, tandis que le thème clair n'est pas encore au point, les couleurs sont souvent trop vives, et l'assortiment des couleurs est parfois mal choisi.

Conclusion

Lorsque nous avons choisi de recréer une blockchain décentralisée, nous ne connaissions que vaguement les principes techniques utilisés pour la faire fonctionner, et nous savions qu'il nous faudrait nous renseigner et apprendre énormément pendant toute la durée du projet. Nous sommes fiers aujourd'hui d'avoir réussi à mener ce projet à bien, et d'avoir atteint les objectifs que nous nous étions fixés au départ, tout cela grâce à la cohésion d'équipe et la mise en commun maîtrisée des différents composants de notre projet dont nous avons su faire preuve.

Le plus grand défi de ce projet a été de faire en sorte que le tout fonctionne.
Ce sujet nécessite des interactions entre plusieurs machines, c'est pourquoi il nous a fallu imaginer des protocoles de communications. De plus, il faut également que la couche de transport, ici notre réseau en pair à pair, soit stable et performante.

Nous sommes satisfaits de la réalisation de notre projet, bien que nous n'étions pas sûrs d'y arriver au départ. Ce projet nous a permis de tous nous améliorer en langage C, ainsi que, pour certains, de découvrir le HTML et le CSS.

Annexes

```
* 83114f2 - (HEAD -> main, origin/main, origin/HEAD) modif readme et makefile (2 hours ago) <thimot>
* 8a08001 - add readme (3 hours ago) <Thimot>
* fd957cf - (origin/minage) remove prints (32 hours ago) <thimot>
*   feeb65f - Merge remote-tracking branch 'origin/reseau' into main (32 hours ago) <thimot>
|\
| * 8807efe - (origin/reseau, reseau) modif serverToJSON (32 hours ago) <thimot>
| * c23b9be - debug envoie api et mining (32 hours ago) <thimot>
| * 09d2575 - correction bug transmission api et mining 2 (32 hours ago) <thimot>
| * 0971666 - correction bug transmission api et mining 2 (32 hours ago) <thimot>
| * 735a4b8 - correction bug transmission api et mining 2 (32 hours ago) <thimot>
| * dc9d5b6 - correction bug transmission api et mining (32 hours ago) <thimot>
| * 9d11598 - test info api et mining (33 hours ago) <thimot>
* | 99e19a0 - update each x sec (33 hours ago) <root>
|/
* 8b2d339 - plus de probleme 32bits (2 days ago) <thimot>
* 7b48009 - debug 32bits (2 days ago) <thimot>
* fd15b0a - test 32bits (2 days ago) <thimot>
* 4272570 - debug 32bits (2 days ago) <thimot>
* 22b8f99 - debug 32bits (2 days ago) <thimot>
* 8101cc0 - test 32bits (2 days ago) <thimot>
* ea28db6 - test 32bits (2 days ago) <thimot>
* 47d35c8 - test 32bits (2 days ago) <thimot>
* 74d9530 - test 32bits (2 days ago) <thimot>
* 091e181 - test 32bits (2 days ago) <thimot>
* 4319b30 - test 32bits (2 days ago) <thimot>
* 1a8d78f - test 32bits (2 days ago) <thimot>
* c2f4a74 - test 32bits (2 days ago) <thimot>
* 893480e - test 32bits (2 days ago) <thimot>
* a37c245 - test 32bits (2 days ago) <thimot>
* fd11570 - test 32bits (2 days ago) <thimot>
* 84bc359 - test 32bits (2 days ago) <thimot>
* f6e64d5 - test 32bits (2 days ago) <thimot>
* e674917 - test 32bits (2 days ago) <thimot>
* 1cd178f - test 32bits (2 days ago) <thimot>
* e39cd4b - test 32bits (2 days ago) <thimot>
* 6374528 - test 32bits (2 days ago) <thimot>
* f74e491 - test 32bits (2 days ago) <thimot>
* 960dc11 - test 32bits (2 days ago) <thimot>
* 79f4d7e - test 32bits (2 days ago) <thimot>
* 4e98f79 - test 32bits (2 days ago) <thimot>
* 977f310 - test 32bits (2 days ago) <thimot>
* ac6843a - test 32bits (2 days ago) <thimot>
* a4aea43 - test 32bits (2 days ago) <thimot>
* 43d48ad - test 32bits (2 days ago) <thimot>
* 81f37f4 - test 32bits (2 days ago) <thimot>
* 204a1c0 - test 32bits (2 days ago) <thimot>
* 3315c25 - test 32bits (2 days ago) <thimot>
* 7c21dbd - test 32bits (2 days ago) <thimot>
* 2ba519a - test 32bits (2 days ago) <thimot>
* c57530a - test 32bits (2 days ago) <thimot>
* 20de83f - test 32bits (2 days ago) <thimot>
* 0315d3c - test 32bits (2 days ago) <thimot>
* 7667585 - debug multithread sendMessage (2 days ago) <thimot>
* 0e4796b - multithread envoie msg (2 days ago) <thimot>
* 593ed1a - multithread envoie msg (2 days ago) <thimot>
* 0ab7dc1 - test reduction timeout socket 2 (3 days ago) <thimot>
* d8e5ee4 - test reduction timeout socket (3 days ago) <thimot>
* 5bc38a9 - (origin/gestion) ajout gestion arguments (3 days ago) <thimot>
*   8a0ad3c - (gestion) Merge remote-tracking branch 'origin/reseau' into gestion (3 days ago) <root>
|\
| * 2179c8d - ajout info dans ServerToJson (3 days ago) <thimot>
| * 8daf49b - ajout info api et mining dans reseau (3 days ago) <thimot>
* | 0eb7b0d - 32 grosses bites (3 days ago) <Le beaugosse adrien>
* | e2da395 - update debug (3 days ago) <Le beaugosse adrien>
* | 976294f - debug 32bit (4 days ago) <Le beaugosse adrien>
* | d01780f - time 64bit (4 days ago) <Le beaugosse adrien>
* | b537233 - modif (4 days ago) <Le beaugosse adrien>
* | d0b5945 - size_t -> ull_t (4 days ago) <root>
* | 37b2dc4 - char* -> BYTE v2 (4 days ago) <root>
* | f3d2d4c - char* -> BYTE <3 (4 days ago) <root>
|/
* 7c17984 - modif ServerToList (2 weeks ago) <thimot>
* 7ce24e5 - modif ServerToList (2 weeks ago) <thimot>
```

```
* 7ce24e5 - modif ServerToList (2 weeks ago) <thimot>
* f8f4034 - modif port ServerToList (2 weeks ago) <thimot>
* e6e8835 - ajout server list API (2 weeks ago) <thimot>
* 1d376a7 - correction bug amount money (2 weeks ago) <thimot>
* 10b280d - ok (2 weeks ago) <root>
* 1460298 - beaucoup de belles choses :) (2 weeks ago) <root>
* b62a2a5 - bah thimot c'est con ca (2 weeks ago) <root>
* 3fe3587 - correction bug url (2 weeks ago) <thimot>
* 897b58f - de (2 weeks ago) <root>
* 0dd0bfa - debug pointeur (2 weeks ago) <thimot>
* b49a272 - sdgfsdfsdfdsfsdfdsfsdfsdfsdfsdfsdfdsfdsfsdfsdfsdfsdfsdfsdfsdfsdfsdf (2 weeks ago) <root>
*   fd76091 - Merge remote-tracking branch 'origin/main' into gestion (2 weeks ago) <root>
|\
| * 60b215c - (origin/API, API) add complete transaction command (3 weeks ago) <thimot>
* | 423efbb - oui (2 weeks ago) <Pierre Litoux>
* |   6e385fa - Merge remote-tracking branch 'origin/gestion' into minage (3 weeks ago) <Pierre Litoux>
|\ \
| * | 50bbb39 - dq (3 weeks ago) <root>
* | | 5b6cb12 - je ferai un alias un jour (3 weeks ago) <Pierre Litoux>
* | |   f5da57a - Merge remote-tracking branch 'origin/gestion' into minage (3 weeks ago) <Pierre Litoux>
|\ \ \
| |/ /
| * | 8e08690 - midif (3 weeks ago) <root>
* | | f99f8cd - in order to merge (3 weeks ago) <Pierre Litoux>
* | |   b5ab7ff - Merge remote-tracking branch 'origin/gestion' into minage (3 weeks ago) <Pierre Litoux>
|\ \ \
| |/ /
| * | 3f42bef - d (3 weeks ago) <root>
| |/
| *   c974f70 - Merge remote-tracking branch 'origin/main' into gestion (3 weeks ago) <thimot>
| |\
| * | 3aae588 - ajout calcul amount of money and general_info.h (3 weeks ago) <thimot>
| * | 17b9d6f - remove warning (3 weeks ago) <root>
* | | 9c281b3 - in order ot merge (3 weeks ago) <Pierre Litoux>
* | |   35786a2 - Merge remote-tracking branch 'origin/gestion' into minage (3 weeks ago) <Pierre Litoux>
|\ \ \
| |/ /
| * | 8fdb5a1 - tltostring (3 weeks ago) <root>
* | |   3c5a68a - ok (3 weeks ago) <Pierre Litoux>
|\ \ \
| | |/
| |/|
| * | 79cbf8b - remove test reseau (3 weeks ago) <thimot>
| |/
| * e384299 - oubli for de test (3 weeks ago) <thimot>
| * 352568f - debug (3 weeks ago) <thimot>
| * 2e9b9c2 - api debug (3 weeks ago) <thimot>
| * 4070b02 - balek (3 weeks ago) <thimot>
| *   c65bd8d - Merge remote-tracking branch 'origin/main' into API (3 weeks ago) <thimot>
| |\
| | * 6575969 - modif reseau passe valgrind (3 weeks ago) <thimot>
| | * 7f4aac7 - push before merge (3 weeks ago) <thimot>
| | * 643d8ea - correction bug sameIP() (3 weeks ago) <thimot>
| | * 707990a - pas de bug en fait (3 weeks ago) <thimot>
| | * aa98a8f - modif print message and main (3 weeks ago) <Thimot>
| | * 73727c8 - correction bug valgrind (3 weeks ago) <thimot>
| | *   d51b7da - modif api (3 weeks ago) <thimot>
| | |\
| | * | a6e07ef - truc (3 weeks ago) <thimot>
| | * |   018ba98 - Merge remote-tracking branch 'origin/gestion' into reseau (3 weeks ago) <thimot>
| | |\ \
| * | \ \   610de6c - push before merge (3 weeks ago) <thimot>
| |\ \ \ \
| | | |_|/
| | |/| |
| * | | |   c6f587f - Merge remote-tracking branch 'origin/main' into API (3 weeks ago) <thimot>
| |\ \ \ \
| | | |/ /
| | |/| |
| * | | | 812b700 - commit before merge (3 weeks ago) <thimot>
* | | | | 0f1a225 - j'ai battu mega satan (3 weeks ago) <Pierre Litoux>
* | | | |   bf7f3e7 - fetch (3 weeks ago) <Pierre Litoux>
|\ \ \ \ \
```

```
| * | | | 812b700 - commit before merge (3 weeks ago) <thimot>
* | | | | 0f1a225 - j'ai battu mega satan (3 weeks ago) <Pierre Litoux>
* | | | |   bf7f3e7 - fetch (3 weeks ago) <Pierre Litoux>
|\ \ \ \ \
| | |_|/ /
| |/| | |
| * | | | f46493c - add time transactions (3 weeks ago) <root>
| | |_|/
| |/| |
| * | | e985ce7 - api upgrade (3 weeks ago) <root>
| * | |   98cd9a0 - Merge remote-tracking branch 'origin/main' into gestion (3 weeks ago) <root>
| |\ \ \
| | |/ /
| * | | 9e6c3ab - ok (3 weeks ago) <root>
| * | |   68f7d7f - zedtu (3 weeks ago) <root>
| |\ \ \
| * | | | 470b707 - gestion thread begin (3 weeks ago) <root>
* | | | | 9d89383 - aaaaaaaaaaaaaaa (3 weeks ago) <Pierre Litoux>
* | | | |   2483fa7 - Merge remote-tracking branch 'origin/main' into minage (3 weeks ago) <Pierre Litoux>
|\ \ \ \ \
| | |_|_|/
| |/| | |
| * | | | d647b64 - main basique pour network (3 weeks ago) <Thimot>
| * | | | 6839a3d - restore main (3 weeks ago) <Thimot>
| * | | | 1fdcfd6 - ajout fonction server List to json (3 weeks ago) <Thimot>
| * | | | 853f07e - truc (3 weeks ago) <thimot>
| | |_|/
| |/| |
| * | | 50b6230 - makefile (3 weeks ago) <Pierre Litoux>
| * | |   b52d830 - Merge remote-tracking branch 'origin/main' into API (3 weeks ago) <Pierre Litoux>
| |\ \ \
| | * | | 3d60ff0 - oubli dossier tools (3 weeks ago) <thimot>
| * | | |   7615760 - Merge remote-tracking branch 'origin/main' into API (3 weeks ago) <Pierre Litoux>
| |\ \ \ \
| | |/ / /
| | * | | a9ca8d9 - modif queue en void * (3 weeks ago) <thimot>
| | | |/
| | |/|
| * | | 5c12532 - API sans gmodule (3 weeks ago) <Pierre Litoux>
| * | |   ec2418d - Merge remote-tracking branch 'origin/main' into API (3 weeks ago) <Pierre Litoux>
| |\ \ \
| | |/ /
| | * |   120bc27 - Merge remote-tracking branch 'origin/main' into reseau (3 weeks ago) <thimot>
| | |\ \
| | | * | 23d123e - debug reseau (4 weeks ago) <thimot>
| | * | | ae8223b - v avec resolution nom de domaine (3 weeks ago) <thimot>
| | * | | 366e764 - ajout commentaire et modif mais ya un bug (3 weeks ago) <thimot>
| | * | | 6dea20a - petites modifs (3 weeks ago) <thimot>
| | * | | 21f67c9 - erreur if dans le cas d'un hostname invalide (3 weeks ago) <thimot>
| | * | | 6a36780 - correction free in read_thread (3 weeks ago) <thimot>
| | * | | a1a0d1e - correction seg fault when invalid hostname (3 weeks ago) <thimot>
| | * | | 62a6de2 - amelioration lisibilite (3 weeks ago) <thimot>
| | * | | 90cb5d3 - v2 tests finaux (3 weeks ago) <thimot>
| | * | | 7301a85 - test (3 weeks ago) <thimot>
| | * | | da16257 - test (4 weeks ago) <thimot>
| | * | | 529d795 - debug (4 weeks ago) <thimot>
| | * | | 53a2d3f - debug (4 weeks ago) <thimot>
| | * | | 07de262 - debug sendNetwork (4 weeks ago) <thimot>
| | * | | 5d8efd3 - debug (4 weeks ago) <thimot>
| | * | | 1fcc65b - debug resolution nom de domaine (4 weeks ago) <thimot>
| | * | | f024dae - modif (4 weeks ago) <thimot>
| * | | | bef57c7 - test (3 weeks ago) <Pierre Litoux>
| * | | |   b4001c5 - Merge branch 'API' of github.com:TREGS4/BlockchainS4 into API (4 weeks ago) <thimot>
| |\ \ \ \
| | * \ \ \   06fc3e2 - merge reseau (4 weeks ago) <hugo.ruiz-le-mao>
| | |\ \ \ \
| | | |/ / /
| | | * | | 8df5aa9 - save before big changement (4 weeks ago) <thimot>
| | | * | | 2b0a4f0 - debug reseau (4 weeks ago) <thimot>
| | | * | | d719ada - debug problem when error (4 weeks ago) <thimot>
| | | |/ /
| | | * | 345e1fa - reseau v2 test (4 weeks ago) <thimot>
| * | | | 7278c9c - reset merge (4 weeks ago) <thimot>
```

```
| * | | | 7278c9c - reset merge (4 weeks ago) <thimot>
| |/ / /
| * | | 677b9dc - v1 API (4 weeks ago) <hugo.ruiz-le-mao>
| * | |   c72127a - merge gestion/origin (4 weeks ago) <hugo.ruiz-le-mao>
| |\ \ \
| | | |/
| | | |/|
| | * | 5c9942e - json (4 weeks ago) <root>
| * | |   2db57e5 - push (4 weeks ago) <hugo.ruiz-le-mao>
| |\ \ \
| | |/ /
| | * | 35389c3 - rsa openssl (5 weeks ago) <root>
| * | | b2afc78 - deuxieme push (4 weeks ago) <hugo.ruiz-le-mao>
| * | | ea5d9ef -  api premier push (4 weeks ago) <hugo.ruiz-le-mao>
| * | | 9d7a62c - tttserver (4 weeks ago) <hugo.ruiz-le-mao>
| | |/
| |/|
| * |   66ade9e - Merge remote-tracking branch 'origin/main' into reseau (4 weeks ago) <thimot>
| |\ \
| | * | cbb6482 - présentation soutenance 2 (6 weeks ago) <Clac9queur>
| | * | 35809b6 - rapport de soutenance 2 (6 weeks ago) <Clac9queur>
| | * | 945f827 - test finis (6 weeks ago) <Thimot>
| | * | 68a832a - test reseau (6 weeks ago) <Thimot>
| * | | 330a065 - prepare for merge into main (4 weeks ago) <thimot>
| * | | ce8e74a - test envoie blockchain (5 weeks ago) <thimot>
| * | | b499fa1 - tests envoie de message (5 weeks ago) <thimot>
| * | | cb5c70f - fontionnel je crois (5 weeks ago) <thimot>
| * | | b7dc581 - bug boucle infinie (5 weeks ago) <thimot>
| * | | 5d4ef5c - ajout IP lors de l'affichage des messages (5 weeks ago) <thimot>
| * | | a83e9ac - debug addClientFromMessage (5 weeks ago) <thimot>
| * | | ecb9504 - tests (5 weeks ago) <thimot>
| * | | 384102e - correction boucle infinie (5 weeks ago) <thimot>
| * | | f7ed2d7 - v3 debut fonctionnel (5 weeks ago) <thimot>
| * | | 9b73737 - debug (5 weeks ago) <thimot>
| * | | 6feda87 - network v3 (5 weeks ago) <thimot>
| * | | 1cb7bba - tests (5 weeks ago) <thimot>
| * | | 5a91a3c - test envoie de message (5 weeks ago) <thimot>
| * | | 0395742 - tests (5 weeks ago) <thimot>
| * | | feb4fc9 - correction petit probleme (5 weeks ago) <thimot>
| * | | 8a2199b - correction seg fault (5 weeks ago) <thimot>
| * | | b57052a - thread pour remove client (5 weeks ago) <thimot>
| * | | 85bd803 - correction fin de thread (5 weeks ago) <thimot>
| * | | 3039c6e - gestion error read (5 weeks ago) <thimot>
| * | | cf82415 - oubli de unlock un mutext (5 weeks ago) <thimot>
| * | | 18a6b0a - modif pour tests (5 weeks ago) <thimot>
| * | | a5dae32 - amelioration code et debug (5 weeks ago) <thimot>
| * | | 0e631fd - supression client lorsque que probleme (5 weeks ago) <thimot>
| * | | 88100b3 - debug diverse fonctions (5 weeks ago) <thimot>
| * | | 56e6866 - correction bug isInList (5 weeks ago) <thimot>
| * | | 33f17b5 - ajout fonction add client (5 weeks ago) <thimot>
| * | | c631b99 - modif en close direct apres envoie d'un message (5 weeks ago) <thimot>
| * | |   e2ddec6 - Merge branch 'reseau' of github.com:TREGS4/BlockchainS4 into reseau (5 weeks ago) <thimot>
| |\ \ \
| | * | | d5bf6be - debug un bug avec les test fait avec Adrien (5 weeks ago) <Thimot>
| | * | | f9c0534 - correction de 1 ligne (5 weeks ago) <Thimot>
| * | | | eab4a8e - commit for merge (5 weeks ago) <thimot>
| |/ / /
| * | | c37de0c - test5 (5 weeks ago) <thimot>
| * | | 287f318 - tests (5 weeks ago) <thimot>
| * | | 6ee4b14 - test envoie message (5 weeks ago) <thimot>
| * | | 7ffa7a8 - tests (5 weeks ago) <thimot>
| * | | e00ca8f - test send big message (5 weeks ago) <thimot>
| * | |   c377fc1 - Merge remote-tracking branch 'origin/gestion' into reseau (5 weeks ago) <thimot>
| |\ \ \
| | | | |/
| | | |/|
| | * | 4fffe63 - bin to struct (5 weeks ago) <root>
| | * | 7e02ca6 - struct to bin (5 weeks ago) <root>
| * | | f4dd54e - commit before merge (5 weeks ago) <thimot>
| * | |   b2812a3 - to merge gestion (5 weeks ago) <thimot>
| |\ \ \
| | |/ /
| | * | 849d9c2 - changement fonctionnement des transactions (6 weeks ago) <Margaux Cavalie>
```

```
| | * | 849d9c2 - changement fonctionnement des transactions (6 weeks ago) <Margaux Cavalie>
| | * |   1d2250b - verif blockchain (7 weeks ago) <Margaux Cavalie>
| | |\ \
| | * | | 88879ca - verif blockchain (7 weeks ago) <Margaux Cavalie>
| * | | | af1a7cf - premiers validees (5 weeks ago) <thimot>
| * | | | 76425ce - changement port when receiving network (5 weeks ago) <thimot>
| * | | | e003425 - fini modif debut tests (5 weeks ago) <thimot>
| * | | | ad0e8e7 - fini modif debut tests (5 weeks ago) <thimot>
| * | | | 1e01008 - correction bug Send() (5 weeks ago) <thimot>
| * | | | 380588b - amelioration envoie message (5 weeks ago) <thimot>
| * | | | ceb6bf1 - ajout print plus facile (6 weeks ago) <thimot>
| * | | | 57c1c5a - test bug message (6 weeks ago) <thimot>
| * | | | 779bebc - correction bug mutex (6 weeks ago) <thimot>
| * | | | 3ea6305 - test (6 weeks ago) <thimot>
| * | | | 73401fb - amelioration code (6 weeks ago) <thimot>
| * | | |   acef1c1 - Merge branch 'reseau' of github.com:TREGS4/BlockchainS4 into reseau (6 weeks ago) <thimot>
| |\ \ \ \
| | * | | | 7627d40 - test reseau (6 weeks ago) <Thimot>
| | * | | | 7e5933e - test reseau (6 weeks ago) <Thimot>
| * | | | | ddd718f - amelioration lisibilite du code (6 weeks ago) <thimot>
| |/ / / /
* | | | | dfdf8f0 - before a merge (3 weeks ago) <Pierre Litoux>
| |_|_|/
|/| | |
* | | |   99deb55 - (minage) Merge remote-tracking branch 'origin/main' into minage (7 weeks ago) <Pierre Litoux>
|\ \ \ \
| |/ / /
| * | |   2b2095e - Merge remote-tracking branch 'origin/main' into reseau (7 weeks ago) <thimot>
| |\ \ \
| | * \ \   319b124 - Merge branch 'main' of https://github.com/TREGS4/BlockchainS4 into main (3 months ago) <Clac9queur>
| | |\ \ \
| | * | | | 29e6575 - rapport de soutenance (3 months ago) <Clac9queur>
| * | | | | 1c81282 - ajout header size (7 weeks ago) <thimot>
| * | | | | d3ac0d2 - remove print (7 weeks ago) <thimot>
| * | | | | ead6612 - modif main (7 weeks ago) <thimot>
| * | | | | 076932d - correction connection client (7 weeks ago) <thimot>
| * | | | | 5d4fc37 - un autre test (7 weeks ago) <thimot>
| * | | | | 20b3c2e - test message (7 weeks ago) <thimot>
| * | | | | 4cb2e19 - amelioration header (7 weeks ago) <thimot>
| * | | | | 7248192 - tests (7 weeks ago) <thimot>
| * | | | | 328cffc - corrections bugs (7 weeks ago) <thimot>
| * | | | | c4a559b - corrections divers bugs (7 weeks ago) <thimot>
| * | | | | 5fad5d0 - bug avec comp IP (7 weeks ago) <thimot>
| * | | | | 9c56d2f - correction IP (7 weeks ago) <thimot>
| * | | | | 627313f - correction main (7 weeks ago) <thimot>
| * | | | | 49ab185 - modif affichage (7 weeks ago) <thimot>
| * | | | | 6b43a78 - correction timeout (7 weeks ago) <thimot>
| * | | | | 1b08540 - tests9 (7 weeks ago) <thimot>
| * | | | | 500e155 - tests8 (7 weeks ago) <thimot>
| * | | | | dd4ebab - tests7 (7 weeks ago) <thimot>
| * | | | | 64c1a1c - tests6 (7 weeks ago) <thimot>
| * | | | | cd1c5ed - tests6 (7 weeks ago) <thimot>
| * | | | | fef2098 - tests5 (7 weeks ago) <thimot>
| * | | | | ed6db7b - tests5 (7 weeks ago) <thimot>
| * | | | | 903d669 - tests4 (7 weeks ago) <thimot>
| * | | | | 1452240 - tests (7 weeks ago) <thimot>
| * | | | | e5c2865 - tests (7 weeks ago) <thimot>
| * | | | | d98ea21 - tests (7 weeks ago) <thimot>
| * | | | | d3c4e68 - tests (7 weeks ago) <thimot>
| * | | | | 333c8ac - premier tests (7 weeks ago) <thimot>
| * | | | | f42303b - reseau quasi fini manque juste l'envoie de message (8 weeks ago) <thimot>
| * | | | | ffa5174 - reception message fonctionnel mais pas de traitement (8 weeks ago) <thimot>
| * | | | | 7d0d5a5 - debut integration parsing message (2 months ago) <Thimot>
* | | | | | 1671893 - minage multi thread (7 weeks ago) <Pierre Litoux>
* | | | | |   90d69f9 - Merge remote-tracking branch 'origin/reseau' into minage (2 months ago) <Pierre Litoux>
|\ \ \ \ \ \
| |/ / / / /
| * | | | | 229f0f5 - compile sans warning (2 months ago) <Thimot>
* | | | | |   4a32757 - Merge remote-tracking branch 'origin/reseau' into minage (2 months ago) <Pierre Litoux>
|\ \ \ \ \ \
| |/ / / / /
| * | | | | 270a5b2 - v1 fonctionnel mais sans peer to peer (2 months ago) <Thimot>
| * | | | | f32e082 - v1 server fonctionne, peut recevoir et envoyer info (2 months ago) <Thimot>
```

```
| * | | | | f32e082 - v1 server fonctionne, peut recevoir et envoyer info (2 months ago) <Thimot>
| * | | | | 3833fca - debut changement vers liste chainee (2 months ago) <Thimot>
* | | | | |   6114687 - commit for merge 2 (2 months ago) <Pierre Litoux>
|\ \ \ \ \ \
| |/ / / / /
| * | | | | 65f95b7 - v1 presque fini reste plus qu'a kill a thread (2 months ago) <Thimot>
| * | | | | 7a380bb - coorection beaucoup de bugs, debut kill propre cote client et serveur (2 months ago) <Thimot>
| * | | | | 857c12e - debug de la liste de server, bug avec realloc (3 months ago) <Thimot>
| * | | | | 74564cf - debug en cours (3 months ago) <Thimot>
| * | | | | 511900c - client se connect au serveur (3 months ago) <Thimot>
| |/ / / /
* | | | | 177abc8 - commit before merge (2 months ago) <Pierre Litoux>
| |_|_|/
|/| | |
* | | | 3b2a132 - Delete blockchain (3 months ago) <TheSnowyxGIT>
* | | | 1eca874 - ajout system zebi de json (3 months ago) <Adrien Pingard>
| |_|/
|/| |
* | | a688104 - creation branch bis (3 months ago) <Margaux Cavalie>
| |/
|/|
* | e323f86 - creation de branch (3 months ago) <Margaux Cavalie>
* |   7c69867 - seqdgqsdgsrfhd (3 months ago) <Adrien Pingard>
|\ \
| * | 9558b0a - rm a file (3 months ago) <Pierre Litoux>
| * | 4c046a5 - minage v1 (3 months ago) <Pierre Litoux>
| * | c974e54 - Pour adrien (3 months ago) <Pierre Litoux>
| * | 090d398 - prototype minage (3 months ago) <Pierre Litoux>
| * |   2b650cd - pouet (3 months ago) <Pierre Litoux>
| |\ \
| | |/
| * | f50fc28 - voila (3 months ago) <Pierre Litoux>
| * | 48fb626 - ok (3 months ago) <Pierre Litoux>
| * |   f0b5566 - Merge remote-tracking branch 'origin/main' into minage (3 months ago) <Pierre Litoux>
| |\ \
| * | | e1ce9f6 - push (3 months ago) <Pierre Litoux>
| * | | f75144f - v1 avec commentaires (3 months ago) <Pierre Litoux>
* | | | ac4ec33 - envoie v1 noeud gestion (3 months ago) <Adrien Pingard>
| |_|/
|/| |
* | | 3cf071e - tous les .c qui passe pas avec -Werror ont ete renomes en .marchepas (3 months ago) <Thimot>
* | | 7f9242d - correction flags -Wall -Wextra on Makefile (3 months ago) <Thimot>
* | | 4c31a21 - correction problem with the Makefile (3 months ago) <Thimot>
| |/
|/|
* | 879a715 - correction problem with the Makefile (3 months ago) <Thimot>
* | b088f79 - garbage (3 months ago) <Thimot>
* | 2a531b4 - delete waste (3 months ago) <Thimot>
* | 32a7346 - fork while client and thread for read and write (3 months ago) <Thimot>
* | 69c2f2f - pb avec lors de la compilation (3 months ago) <Thimot>
* | 0f7dcb7 - debut (3 months ago) <Thimot>
* |   0a32c1b - Merge branch 'main' into reseau (3 months ago) <Thimot>
|\ \
| |/
| * 6543a12 - modif Makefile (3 months ago) <Thimot>
* | 46c6bb2 - truc (3 months ago) <Thimot>
* |   212903b - Merge branch 'main' into reseau (3 months ago) <Thimot>
|\ \
| |/
| * a92afcd - makefile fonctionnel (3 months ago) <Thimot>
* | 76d3e6d - premier push (3 months ago) <Thimot>
|/
* 04cc70d - ajout d'un main (3 months ago) <Thimot>
* 8e9aaf9 - reverse (4 months ago) <TheSnowyxGIT>
*   7ecb62d - Merge branch 'main' of github.com:TREGS4/BlockchainS4 into main (4 months ago) <TheSnowyxGIT>
|\
| * 79e1339 - structure (4 months ago) <Thimot>
| *   d728c0e - Merge branch 'main' of github.com:TREGS4/BlockchainS4 into main (4 months ago) <Thimot>
| |\
| * | 430bac9 - branche web (4 months ago) <Thimot>
* | | f6549d5 - fe (4 months ago) <TheSnowyxGIT>
| |/
|/|
```

```
|\
| * 79e1339 - structure (4 months ago) <Thimot>
| *   d728c0e - Merge branch 'main' of github.com:TREGS4/BlockchainS4 into main (4 months ago) <Thimot>
| |\
| * | 430bac9 - branche web (4 months ago) <Thimot>
* | | f6549d5 - fe (4 months ago) <TheSnowyxGIT>
| |/
|/|
* | 1f3f032 - zebi (4 months ago) <Adrien Pingard>
|/
* 6bf9f63 - structure (4 months ago) <Clac9queur>
* 6e14f1a - Ajout d'une librairie de hashage sha256 (4 months ago) <Adrien Pingard>
* e26c5ea - bonjour (4 months ago) <Hugo Ruiz Le Mao>
* fa839f5 - Initial commit (4 months ago) <Clac9queur>
```

Historique des commits sur notre répertoire de travail

```c
typedef struct
{
    int type;
    unsigned long long sizeData;
    BYTE *data;
} MESSAGE;
```

Réseau 1

```c
//contains the hostname and the port of a client
struct address
{
    //hostname of the client, null terminated
    char *hostname;

    //port of the client
    char port[PORT_SIZE + 1];
};
```

Réseau 2

```
struct clientInfo
{
    struct address address;
    int api;
    int mining;

    //TRUE is the it's the sentinel, FALSE otherwise
    short isSentinel;

    //Pointer to the next, previous and sentinel element
    //prev of sentinel is always NULL
    //next of the last element is always NULL
    struct clientInfo *next;
    struct clientInfo *prev;
    struct clientInfo *sentinel;
};
```

Réseau 3

```
//One structure is initialised when starting, it contains all the informations for each part of the network code
struct server
{
    //Status of the server
    int status;

    //Boolean saying if the server is also api and/or a miner
    int api;
    int mining;

    struct address address;

    shared_queue *OutgoingMessages;
    shared_queue *IncomingMessages;

    //The sentinel to the list of known server
    struct clientInfo *KnownServers;

    //lock when using/modifing the list of known servers
    pthread_mutex_t lockKnownServers;

    //lock when using/modifing the status of the server
    pthread_mutex_t lockStatus;
};
```

Réseau 4