Julie Barthet - Maxime Brasley - Alexis Lopes Vaz - Mathieu Vinot

Projet : Distributed Mining

Sommaire :

1. Introduction

a. Présentation du projet

Ce projet a pour but de développer un système distribué en Java pour la recherche de hashs spécifiques. Pour une chaîne de caractères donnée, nous devons trouver un hash qui commence par un nombre déterminé de zéros. Cette technique, connue sous le nom de "Proof of Work", est largement utilisée dans le monde des crypto-monnaies, notamment le Bitcoin, pour sécuriser et valider les transactions.

2. Répartition du travail

a. Présentation de l'équipe

L'équipe est composée de quatre membres:

- Julie Barthet, sous le nom jbrht sur GitHub
- Maxime Brasley, sous le nom maxime-brsl sur GitHub
- Alexis Lopes Vaz, sous le nom de lopezvaz3u sur GitHub
- Mathieu Vinot, sous le nom de ElFamosoMathieu sur GitHub

b. Répartition du travail

Maxime a commencé à mettre en place le projet, en créant les différentes classes nécessaires, à établir une première connexion entre un worker et le serveur. A la suite de cela, il a essayé de mettre en place le protocole de connexion entre le serveur et les workers mais a rencontré quelques difficultés pour réaliser cette tâche. Il a continué sur la mise en place d'une classe permettant les interactions avec les services webapp fournies, puis a commencé à travailler sur les différentes fonctions pour réaliser les tâches de minage, tout en faisant le lien avec les services de la webapp.

Alexis a continué les recherches de Maxime sur la mise en place du protocole entre le serveur et les workers. Il a essentiellement travaillé sur cette partie pendant toute la durée du projet. Il a aussi contribué à trouver et corriger un certain nombre de bugs dans l'application ainsi que sur l'amélioration et la maintenabilité du code.

Julie s'est concentrée sur l'application du protocole en créant les fonctions qui gèrent les messages envoyés et reçus avec l'aide d'Alexis, puis a adapté une partie du code de Maxime pour qu'il respecte le protocole attendu.

Mathieu a travaillé principalement sur le canal sécurisé et la génération aléatoire du mot de passe. Il a essayé d'aider sur le travail de la conception et a cherché des solutions sur le statut des workers.

3. Architecture du projet

a. Description générale

Le projet est une application distribuée avec trois parties distinctes. En premier, nous avons la webapp, elle nous permet de récupérer un travail à faire mais aussi de vérifier si le hash trouvé est bien correct. En second, nous avons le serveur, c'est lui qui fait le lien avec la webapp et les workers pour leur envoyer des instructions. Et enfin nous avons les workers, qui se connectent au serveur et attendent les instructions, c'est eux qui s'occupent de la logique de minage.

Voici les différentes classes dont notre projet est composé :

```
src/
├── ApiConnect.java
├── LauncherServer.java
├── Messages.java
├── Server.java
├── Solution.java
├── State.java
└── Worker.java
```

Tout d'abord nous avons ***ApiConnect***, c'est la classe qui permet de communiquer avec les webservices. Dans cette classe, on retrouve les informations pour se connecter à la webapp ainsi que les méthodes qui appellent les webservices *generate_work* et *validate_work*.

La classe ***LauncherServer*** permet de gérer les interactions avec l'utilisateur directement. Elle lance le serveur et affiche sa console ce qui permet d'entrer des commandses. Lorsqu'elle reconnaît une commande, elle l'envoie au serveur qui fera les actions nécessaires.

Messages est une classe avec uniquement des constantes correspondant aux messages qui seront envoyés par le serveur et les workers. Nous avons opté pour une classe listant tous ces messages dans le but d'augmenter la scalabilité, principalement parce que les messages sont utilisés dans les deux classes pour être reçus ou bien envoyés.

Le ***Server*** est la classe qui écoute les connexions des workers sur le port 1337, et qui initie le dialogue à la connexion d'un worker. Cette classe utilise les fonctions d'***ApiConnect*** pour toutes les interactions avec la webapp et se charge également d'envoyer les instructions au worker.

Solution est un record qui prend une difficulté, un hash et un nonce, ce qui permet de transporter les données d'une solution plus facilement afin de construire le String qui sera envoyé à la webapp pour vérification

State est une énumération qui correspond à l'état d'un worker, il peut être en WAITING, READY, DISCONNECTED, MINING. L'énumération permet d'utiliser ces états dans l'ensemble du projet plus facilement.

Enfin, la classe ***Worker*** permet de mettre en place le minage et de se connecter au ***Server***. Le ***Worker*** une fois connecté, écoute les instructions du ***Server.*** Si l'utilisateur veut miner, le ***Worker*** reçoit la data, le jump et le nonce de départ. Le jump correspond au nombre de workers connectés. Une fois un hash trouvé, il communique avec le ***Server*** pour faire vérifier les informations.

Pour le canal de communication sécurisé, des clés et certificats sont à généré à la racine du projet.

4. Choix techniques

a. Technologies utilisées

Notre projet est développé uniquement en Java 21, et n'utilise aucune librairie externe à la JDK. Nous utilisons Socket de Java Net pour permettre la connexion entre le serveur et les workers ainsi que la communication bidirectionnelle, ce qui facilite les échanges de données.
Pour la communication avec plusieurs workers, nous utilisons des Thread qui permettent d'exécuter parallèlement plusieurs tâches, ce qui permet d'écouter encore les commandes du serveur pour notamment le statut mais aussi l'annulation du travail. Nous utilisons aussi les Executors et ExecutorService du package Concurrent de Java. Ils permettent une gestion efficace de l'exécution concurrente de multiples tâches de minage (une par worker), en attribuant chaque tâche disponible à un thread distinct du groupe de threads tout en optimisant l'utilisation des ressources.
Pour la connexion sécurisée, nous avons utilisé Secure Socket Layer qui à l'aide du protocole TLS va assurer la confidentialité et l'intégrité des données tout en correspondant aux consignes.

b. Justifications de nos choix

Pour l'algorithme de minage, nous utilisons la méthode de force brute. Les workers vont essayer toutes les possibilités de 0 jusqu'à trouver la solution. Nous avons choisi cette méthode car elle est simple à mettre en place et peut fonctionner avec un seul worker. Évidemment, plus il y a de workers disponibles, plus cela sera efficace. En effet, nous utilisons un pas pour répartir le travail entre les workers. Si nous avons trois workers, le premier commence la recherche avec un nonce à 0, le deuxième à 1 et le troisième à 2. Ensuite, chaque worker incrémente son nonce par le nombre total de workers. Cette méthode permet de multiplier la vitesse de calcul par le nombre de workers (à condition que le nombre de workers ne dépasse pas le nombre de cœurs du PC), tout en couvrant toutes les possibilités.

Chaque worker utilise une méthode handleMessage() pour centraliser le traitement des messages reçus du serveur. Cette approche permet une bonne gestion des différents types de messages, améliorant ainsi la maintenabilité et facilitant l'ajout de nouveaux types de messages si nécessaire.

Nous utilisons des sockets pour la communication entre le serveur et les workers. La classe Worker possède un PrintWriter et un BufferedReader qui permettent de faire fonctionner le protocole en envoyant des messages de manière bidirectionnelle. Les sockets nous offrent également la flexibilité nécessaire pour étendre notre architecture et gérer des communications complexes entre plusieurs composants distribués du système.

Nous avons utilisé SSL car c'était l'évolution du socket que nous utilisions déjà dans un premier temps. Nous avons utilisé le protocole TLS.

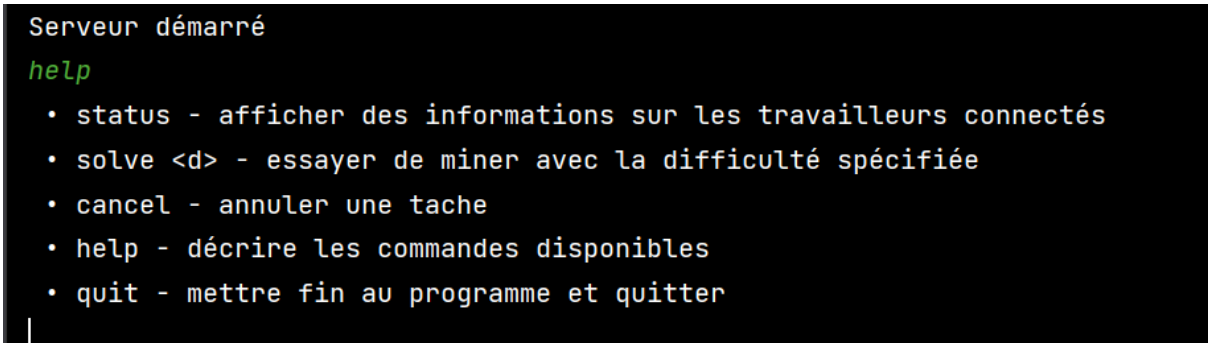
5. Difficultés rencontrées et solutions trouvées

Nous avons rencontré quelques difficultés, notamment sur le type des données sur lesquelles travailler. En effet pour le minage, nous avons initialement travaillé avec des “int”. Integer fonctionnait bien jusqu'à la difficulté 7 car le nonce trouvé était à l'itération 553.369.067 ce qui est inférieur à la max_value de Integer qui est de 2.147.483.647. La difficulté 8 a été trouvée en plus de 22 milliards d'itérations soit 10x plus que la difficulté 7. Nous avons opté pour convertir les données en tableau de bytes, que ce soit le nonce, le hash ou le jump. Nous avons mis en place une fonction pour incrémenter les tableaux, notamment pour augmenter le nonce à chaque itération mais aussi une fonction pour concaténer le nonce au hash afin de vérifier s' il commence par le nombre de 0 souhaité.

Une autre des difficultés a été l'utilisation des webservices, notamment validate_task qui prend un format JSON, qui n'était pas le même que sur la documentation du webservice. Notre programme trouvait le bon hash et le bon nonce mais lors de l'envoi au webservice il y avait une erreur. Pour comprendre le souci, nous avons utilisé postman afin de voir plus en détail l'erreur et c'est là que nous avons compris qu'il fallait refaire reformater le JSON.

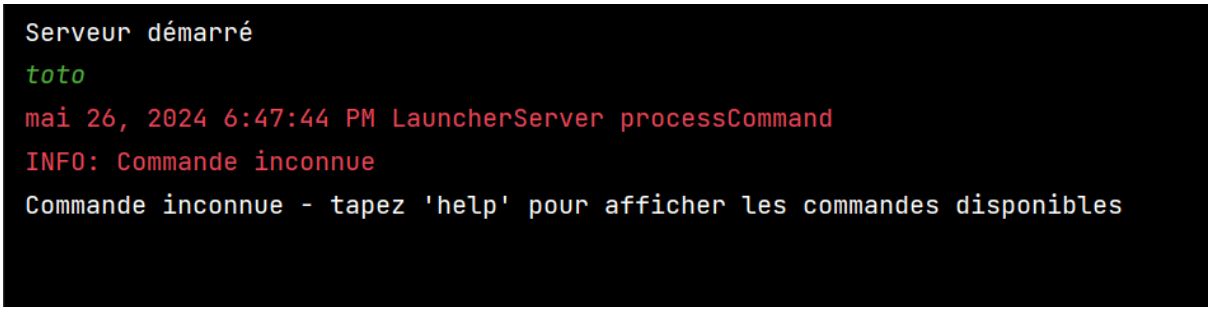
6. Fonctionnalités implémentées

Démarrage du serveur

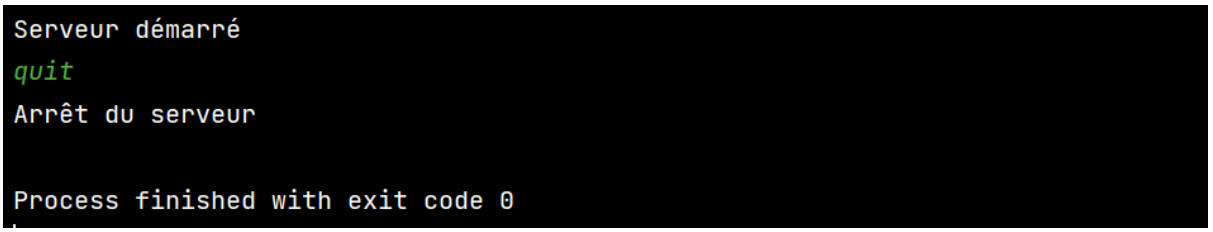


Au lancement de LaucherServeur.java, le serveur se lance et un terminal est disponible pour que l'utilisateur rentre les commandes disponibles.
Les commandes disponibles sont énumérées avec la commande *help* comme montré sur l'image ci-dessus.

Une commande inconnue ou mal formée renvoie ce message à l'utilisateur :



Arrêt du serveur



La commande *'quit'* ferme le serveur et libère le port 1337.

Démarrage d’un worker

Si le serveur n’est pas lancé on a un message d’erreur :

```
mai 26, 2024 7:00:35 PM Worker main
WARNING: Erreur lors de la création du socket: Connection refused: connect

Process finished with exit code 0
```

Sinon le worker se connecte au serveur et le protocole de communication démarre.

Protocole d’authentification

Lorsqu’un worker se connecte au serveur, le serveur teste d’abord si le worker connaît le protocole avec le message WHO_ARE_YOU_?. Si le worker répond ITS_ME,le serveur valide et passe à la suite.
Après ça le serveur demande GIMME_PASSWORD puis le worker répond PASSWD avec le mot de passe.
Sur la branche main, le mot de passe est en dur dans le serveur et le worker.
Sur la branche communication_ssl (que nous n’avons finalement pas implémentée avant la fin du projet) le mot de passe est généré aléatoirement puis transmis au worker via une socket SSL.

Affichage côté worker:

```
Message received : WHO_ARE_YOU_?
Message sent : ITS_ME
Message received : GIMME_PASSWORD
Message sent : PASSWD mdp
```

Affiche côté serveur :

```
Serveur démarré
Nouveau worker connecté: Socket[addr=/127.0.0.1,port=55924,localport=1337]
Message sent : WHO_ARE_YOU_?
Message received : ITS_ME
Message sent : GIMME_PASSWORD
Message received : PASSWD mdp
```

Erreur dans le mot de passe fourni par le worker

Si le mot de passe fourni par le worker ne correspond pas à ce qu'attend le serveur, celui-ci lui renvoie le message DONT_FOOL_ME et ferme la connexion avec le worker.

Affichage côté worker:

```
Message received : GIMME_PASSWORD
Message sent : PASSWD mdp1
Message received : YOU_DONT_FOOL_ME
mai 26, 2024 7:46:22 PM Worker closeConnection
INFO: Connexion fermée.
mai 26, 2024 7:46:22 PM Worker run
WARNING: Erreur lors de la lecture du message: Socket closed

Process finished with exit code 0
```

Affiche côté serveur :

```
Message sent : GIMME_PASSWORD
Message received : PASSWD mdp1
Message sent : YOU_DONT_FOOL_ME
mai 26, 2024 7:46:22 PM Worker closeConnection
INFO: Connexion fermée.
```

Signalement de la disponibilité du worker

Une fois l'authentification passée le serveur prévient le worker qu'il est accepté avec la commande HELLO_YOU puis le worker répond par un READY pour signifier sa disponibilité pour une tâche enfin le serveur répond OK.

Affichage côté worker:

```
Message sent : PASSWD mdp
Message received : HELLO_YOU
Message sent : READY
Message received : OK
```

Affiche côté serveur :

```
Message received : PASSWD mdp
Message sent : HELLO_YOU
Message received : READY
Message sent : OK
```

Lancement d'une tâche de minage

Lorsque l'utilisateur entre solve avec la difficulté qu'il souhaite, le serveur reçoit la commande et va initialiser les paramètres nécessaires au minage. En effet, il récupère la difficulté dans le message pour récupérer le travail, il calcule (en fonction des workers) les paramètres du nonce (start et increment). Il envoie toutes ces données aux workers à travers les messages NONCE, PAYLOAD et SOLVE en suivant le protocole.

Une fois que nous nous sommes assurés que les trois instructions ont été reçues et que les données ont été récupérées, le minage peut donc débuter.

Affichage côté serveur :

```
solve 6
Message sent : NONCE 0 2
Message sent : NONCE 1 2
Message sent : PAYLOAD s3xv226nfl84z6g6yfysrg55tcvdkp6r3yuhmg3ra7tcqo6530t
Message sent : PAYLOAD s3xv226nfl84z6g6yfysrg55tcvdkp6r3yuhmg3ra7tcqo6530t
Message sent : SOLVE 6
Minage en cours...
Message sent : SOLVE 6
Minage en cours...
```

Découverte et transmission de la solution

Une fois la solution trouvée par un worker celui-ci le signale au serveur via la commande FOUND

```
Minage du bloc: 012b7057
Message sent : FOUND 00000021d3966beb3cebd24baf692b613071541b09f42ce107408fdea2f6d808 12b7057
```

Le serveur la reçoit et envoie SOLVED à tous les workers en train de miner :

```
Durée de l'exécution: 00:00:30
Message sent : SOLVED
Message sent : SOLVED
```

Le worker le reçoit et s'arrête :

```
Minage du bloc: 0137d024
Message received : SOLVED
Minage du bloc: 0137d026
Le travail a été annulé
Message sent : READY
```

Avant de se remettre en état READY.

Annulation d'une tâche en cours

L'utilisateur peut écrire dans le terminal du serveur la commande cancel pour annuler la tâche en cours. Tous les workers recevront le message CANCELLED ce qui annulera leur tâche en cours. Les workers disponibles enverront READY une fois de nouveau prêt à recevoir des commandes.

Affichage côté serveur:

```
cancel
Message sent : CANCELLED
mai 26, 2024 8:21:32 PM Server cancelTask
INFO: Toutes les tâches en cours ont été annulées.
```

Affichage côté worker:

```
Message received : CANCELLED
Minage du bloc: 0eb1ca
Le travail a été annulé
Message sent : READY
```

Affichage de l'état des workers

La commande status affiche le statut de minage des workers acceptés par le serveur. Le serveur envoie à chacun des workers la commande PROGRESS ce à quoi ils répondent NOPE ou SOLVE xxx en fonction de s' ils minent ou non.

Pour le cas où les les workers sont en attente (on envoie une commande par worker) :

Affichage côté serveur:

```
status
Message sent : PROGRESS
Message received : READY
Message sent : PROGRESS
Message received : NOPE
```

Affichage côté worker:

```
Message received : PROGRESS
Message sent : NOPE
```

Pour le cas où les workers sont en train de miner :

```
status
Message sent : PROGRESS
Message received : TESTING 30a454
Message sent : PROGRESS
Message received : TESTING 008bc7df
```

Et pour le cas où aucun worker est connecté :

```
Serveur démarré
status
Aucun worker connecté
```

Mot de passe généré aléatoirement et dans un canal sécurisé

Lors de la création du serveur on génère aléatoirement un mot de passe qui est composé de lettres minuscules, majuscules et de numéros.

```
Message received : password's server
Message received : GIMME_PASSWORD
Message received : PASSWD Qj4yVSkm
```

```
Message received : password's server
Message received : GIMME_PASSWORD
Message received : PASSWD BbwAlAYW
```

Après avoir généré les clés privées et les certificats du serveur et du client, le résultat fonctionne correctement.

7.Tests de l'application

Nous avons testé l'application de manière simple. Nous effectuons des scénarios de tests lorsque nous testons une fonctionnalité et avant de l'envoyer sur la branche main.
Le scénario est le suivant :
On lance le serveur et deux workers, on vérifie que le status renvoie bien NOPE.

```
Serveur démarré
Nouveau worker connecté: Socket[addr=/127.0.0.1,port=65029,localport=1337]
Message sent : WHO_ARE_YOU_?
Message received : ITS_ME
Message sent : GIMME_PASSWORD
Message received : PASSWD mdp
Message sent : HELLO_YOU
Message received : READY
Message sent : OK
Nouveau worker connecté: Socket[addr=/127.0.0.1,port=65034,localport=1337]
Message sent : WHO_ARE_YOU_?
Message received : ITS_ME
Message sent : GIMME_PASSWORD
Message received : PASSWD mdp
Message sent : HELLO_YOU
Message received : READY
Message sent : OK
status
Message sent : PROGRESS
Message received : NOPE
Message sent : PROGRESS
Message received : NOPE
```

On lance une tâche de minage (et on vérifie la console des workers) :

```
solve 6
Message sent : NONCE 1 2
Message sent : NONCE 0 2
Message sent : PAYLOAD s3xv226nfl84z6g6yfysrg55tcvdkp6r3yuhmg3ra7tcqo6530t
Message sent : PAYLOAD s3xv226nfl84z6g6yfysrg55tcvdkp6r3yuhmg3ra7tcqo6530t
Message sent : SOLVE 6
Minage en cours...
Message sent : SOLVE 6
Minage en cours...
```

On annule la tâche (et on vérifie la console de chaque worker) :

```
cancel
Message sent : CANCELLED
Message received : READY
Message sent : CANCELLED
Message received : READY
mai 26, 2024 10:39:33 PM Server cancelTask
INFO: Toutes les tâches en cours ont été annulées.
```

Enfin selon le test on refait un status et/ou un solve qui se termine par la résolution du problème.

Nous savons que ce n'est pas la manière la plus optimale de faire. Cependant réaliser des tests unitaires tout en utilisant des threads Java pour recréer le contexte de notre application est quelque chose d'inconnu pour nous et cela nous semblait complexe donc avons choisi de faire nos tests à la main.

Pour la partie sécurisée le résultat est le même, il faut juste bien générer les clés et les certificats en suivant le mode opératoire présent dans le README de la branche.

8.Bilan du projet

Voici un récapitulatif des aspects positifs, négatifs et des leçons tirées de la réalisation de ce projet :

Points positifs

- L'architecture que nous avons mise en place a permis une séparation claire entre la webapp, le serveur et les workers, ce qui a facilité le développement et les tests.
- L'utilisation de threads et de l'ExecutorService s'est avérée efficace pour l'exécution des tâches.
- La méthode de force brute pour le minage est simple, et efficace pour le problème à résoudre avec l'ajout de workers.
- Nous avons réussi à implémenter toutes les fonctionnalités principales, comme le démarrage/arrêt du serveur et des workers ou la gestion des messages ainsi que le minage d'une difficulté donnée.

Points négatifs

- La contrainte des threads a également été complexe à appréhender, car au-delà de la communication entre le worker et le serveur, il fallait, qu'en plein minage, la saisie et l'envoi d'autres commandes soient possibles. De nombreux bugs ont de ce fait été rencontrés et corrigés.
- L'erreur rencontrée liée à la documentation du webservice nous a fait perdre du temps, mais cela nous a permis de développer notre réflexion pour déployer des solutions afin d'identifier la source du problème.
- Le retard prit sur le mot de passe dans un canal sécurisé nous à contraint à laisser la fonctionnalité sur une autre branche par manque de test.

9. Conclusion

Le projet "Distributed Mining" a permis de mettre en place une application pour la recherche de hashs spécifiques. Malgré de nombreuses difficultés, nous avons réussi à construire un système qui répond aux attentes et qui suit le protocole demandé. Pour cela nous avons mis en place un système distribué qui communique via un protocole que nous avons créé. Il est possible de contrôler l'application en ligne de commande comme demandé et nous voyons en direct les messages envoyés par chacune des parties de l'application. Ce projet, bien que complexe selon nous, nous a permis de concevoir et développer un projet inédit dans nos études. Ce qui a été au final un défi difficile mais formateur.