Distributed Mining - Réseaux informatiques

Cyber Sparkle

MIAGE I

2023 - 2024

CASTRO MOUCHERON Nicole
FAYNOT Marine
FRASELLE Nadège
WEISS Lucas

Table des matières

INTRODUCTION..... 3
Présentation du projet..... 3
Organisation du travail..... 4
DÉVELOPPEMENT..... 5
Représentation de l'échange..... 5
Le serveur..... 6
Le worker..... 8
La recherche de hash..... 9
La connexion au webservice..... 10
Scénario de test..... 11
Mining 101..... 11
Miner avec plusieurs machines..... 14
Problèmes rencontrés..... 16
CONCLUSION..... 17
Résumé du projet..... 17
Perspectives d'amélioration..... 18

INTRODUCTION

Présentation du projet

Dans le cadre du projet de *Distributed Mining*, nous avions pour objectif de développer un système de délégation de tâches à un ou plusieurs *workers* et de validation des résultats obtenus.

Cyber Sparkle a pour tâche la recherche d'un *hash* spécifique avec un serveur centralisé qui pilote les *workers* et qui peut communiquer avec eux à tout moment. La communication est faite via *Transmission Control Protocol* (TCP) et les tâches sont récupérées auprès d'une *webapp* (application web) pour être réparties ensuite auprès des workers. Lorsqu'un des *workers* trouve un *nonce* (*number used once*, numéro utilisé une fois), le résultat est envoyé à la *webapp* pour valider le résultat.

Vous trouverez ici le dépôt GitHub du projet ainsi que la documentation utilisateur dans le fichier README. La documentation technique est disponible dans le dossier "javadoc" ci-joint et dans le dépôt GitHub. Pour y accéder ouvrez le fichier `javadoc/apidocs/index.html` dans le navigateur de votre choix.

Organisation du travail

Suite à la lecture du sujet et à l'avoir analysé, nous avons identifié les tâches suivantes :

1) La mise en place
2) L'algorithme de hachage
3) La mise en service du client
4) La mise en service du serveur
5) La gestion des erreurs
6) Les scénarios de test

Ces tâches ont ensuite été découpées en sous catégories, pour que nous puissions avoir un plan d'attaque. Nous avons naturellement travaillé en deux binômes en *peer programming*, car cela nous semblait plus abordable pour déboguer efficacement et partager nos idées. Le binôme Lucas et Marine se sont occupé de la mise en place du projet, c'est-à-dire préparer les énumérations pour les protocoles réseau et HTTP et la DTO nécessaire pour notre solution. Ce binôme a également pris en charge la mise en service du serveur, c'est-à-dire écouter les clients qui se connectent, gérer la connexion de multiples clients, la mise en place de threads pour gérer cette connexion.

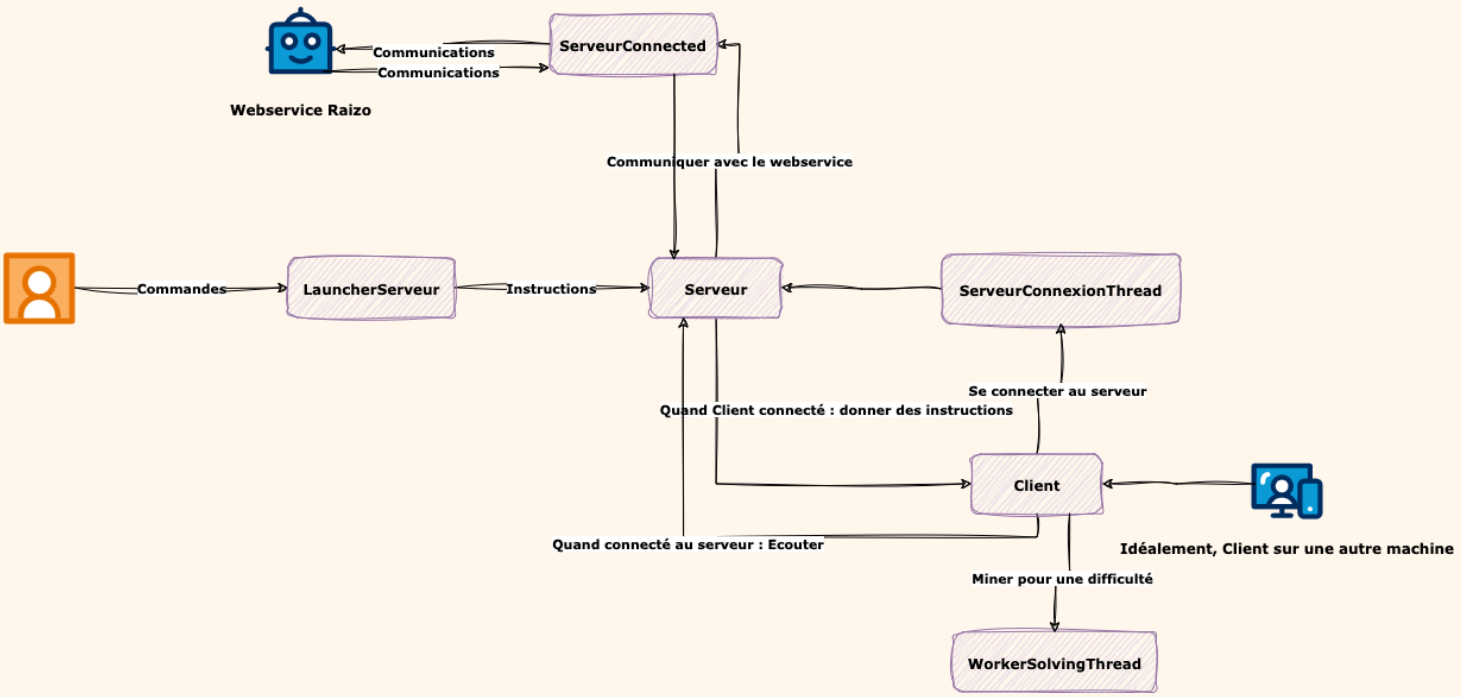
Le binôme Nadège et Nicole ont pris en charge l'algorithme de hachage, la mise en service du client, c'est-à-dire écouter les requêtes du serveur, prendre en compte les instructions y compris pendant qu'un minage est en cours, ou encore mettre en place des threads qui rendent l'écoute du serveur possible lors d'un minage. Ce binôme a également travaillé sur la communication entre le serveur et le webservice Raizo fourni.

Les quatres membres de l'équipe se sont réparti le travail en ce qui concerne la communication entre l'utilisateur et le client, la gestion des éventuelles erreurs (en créant des erreurs customisées notamment), ou encore la rédaction du rapport et des éléments du rendu.

DÉVELOPPEMENT

Représentation de l'échange

Pour introduire notre projet, voici un schéma qui résume les composants principaux et explique comment ils interagissent entre eux.



Ce schéma ne représente pas les DTO, énumérations ou classes utilitaires.

Le serveur

Le serveur fournit les services et ressources nécessaires aux clients. Il écoute d'abord toutes les connexions entrantes de ceux-ci puis génère les tâches à leur donner, les envoie et les valide auprès de l'API raizo. Le serveur utilise un patron Singleton pour lequel il n'y a qu'une seule instance de celui-ci. Il permet de lancer une nouvelle tâche de minage avec une difficulté donnée, de valider une solution proposée par un client et d'afficher l'état actuel des clients connectés au serveur. Si le programme est arrêté proprement, le serveur informe les clients pour qu'ils annulent les tâches en cours. La connexion au serveur est faite par socket sur le port 1337 avec la classe `ServerSocket`.

Il a un *Launcher* qui permet de le lancer et d'écouter et traiter les commandes de l'utilisateur. Le *Launcher* initialise une seule instance de la classe "Serveur" puis reste à l'écoute d'instructions jusqu'à ce que l'utilisateur quitte le programme. Les commandes auquelle le serveur répond sont :

- `status` : affiche les informations sur les clients connectés
- `solve <d>` : essaye de miner avec une difficulté donnée
- `cancel` : annule la tâche en cours
- `help` : affiche le menu de commandes disponibles
- `quit` : termine les tâches en cours et quitte l'application

L'instruction "solve <d>" permet de lancer la résolution d'une tâche, mais avant de résoudre la tâche le programme doit récupérer les données avec la méthode `generate_work` qui fait un appel HTTP au webservice pour obtenir la tâche. Ensuite, le *nonce* est calculé en fonction du nombre de clients disponibles puis les tâches sont distribuées auprès des clients. Une fois que celle-ci est résolue par un des clients, la méthode `validate_work` fait un appel HTTP au *webservice* avec la difficulté, le *nonce* et le *hash*, qui renvoie "vrai" si la solution trouvée est correcte ou "faux" sinon.

Le serveur suit un protocole spécifique et ne répond qu'aux instructions définies, c'est pourquoi nous avons fait une énumération pour éviter des fautes de frappe ou autres erreurs qui pourraient générer des erreurs. Voici les règles du protocole :

- `WHO_ARE_YOU` : première commande envoyée par le serveur à un client
- `GIMME_PASSWORD` : réponse à *ITS_ME* pour demander le mot de passe au client
- `OK` : réponse à *READY* pour informer le client qu'on a pris en compte sa participation
- `HELLO_YOU` : le mot de passe est correct, le client est maintenant connecté
- `YOU_DONT_FOOL_ME` : le mot de passe est incorrect et la connexion est fermée
- `SOLVED` : indique aux workers qu'une solution a été trouvée et qu0il doivent abandonner le travail en cours
- `PROGRESS` : demander l'état d'un worker
- `CANCELLED` : indique aux workers qu'ils doivent abandonner le travail en cours
- `NONCE <start> <increment>` : nonce par lequel le client doit commencer la recherche
- `PAYLOAD <data>` : données a utiliser pour trouver le hash
- `SOLVE <difficulty>` : résoudre pour une difficulté donnée

Le worker

Le *worker* ou client est l'entité qui se connecte au serveur et résout les tâches envoyées par celui-ci. Il se connecte au serveur grâce à son adresse IP et est à l'écoute de messages de sa part grâce à un *InputStreamReader*. Le client va réagir en fonction du message selon un protocole spécifique déterminé par les instructions suivantes :

- `ITS_ME` : réponse à l'instruction *WHO_ARE_YOU*
- `PASSWD <password>` : réponse à *GIMME_PASSWORD*
- `READY` : indique au serveur que le worker est prêt à exécuter une tâche
- `FOUND <hash> <nonce>` : indique au serveur qu'il a trouvé une solution pour la tâche
- `TESTING` : réponse à *PROGRESS*, s'il est en cours de minage il doit renvoyer aussi le *nonce* courant, "NOPE" sinon
- `NOPE`
- `SHUTTING_DOWN` : indique au serveur qu'il met fin au travail en cours

Une exception personnalisée (*InstructionUnknown*) est lancée si l'instruction reçue ne correspond pas au protocole établi.

Pour résoudre les tâches, le *worker* a besoin des données, la difficulté, le *nonce* initial et l'incrément. Le *nonce* initial correspond à *n*-1, avec *n* le nombre de *workers* disponibles. L'incrément correspond au nombre de *workers* disponibles pour ainsi tester tous les *nonces* possibles. Le *hash* est calculé, vérifié localement pour confirmer qu'il commence par le nombre de zéros correspondant à la difficulté donnée, et retourné sous forme de "SolutionDto" (*Data Transfer Object*). Le DTO est utilisé pour simplifier la communication et rendre les couches de l'application plus indépendantes. Il contient un attribut *nonce* et un attribut *hash*. Si le DTO n'est pas nul alors le *worker* répond au serveur "FOUND" avec le *hash* et le *nonce*, sinon le *worker* répond "READY" pour indiquer qu'il est prêt à recevoir une autre tâche.

La recherche de *hash*

Pour calculer le *hash*, le *nonce* initial est transformé en un tableau de bytes grâce à la méthode `toByteArray()`, puis en hexadécimal avec la fonction `formatHex()` qui le renvoie comme une chaîne de caractères. Le *hash* est ensuite calculé grâce à l'algorithme de hachage SHA-256 et renvoyé en format hexadécimal. Le *worker* vérifie si le *hash* trouvé commence par le nombre de zéros du niveau de difficulté, si c'est correct alors la solution a été trouvée et elle sera par la suite envoyée au serveur pour être validée. Si le *hash* n'est pas correct, alors le processus est répété jusqu'à ce que la solution soit trouvée ou qu'on demande au *worker* de s'arrêter, par exemple avec la commande `cancel`.

La connexion au webservice

Afin de récupérer les données liées au projet, et de pouvoir valider la solution trouvée, il était nécessaire d'arriver à se connecter au webservice. La connexion se fait avec une url, avec un ou plusieurs paramètres afin de pouvoir accéder à un retour positif ainsi qu'aux données:

- La génération d'une tâche: c'est une requête GET dans laquelle on indique la complexité. Dans le cas où cette complexité serait déjà résolue, on est informé, sinon, on reçoit la data dont on trouve le hash.
- La validation d'une tâche: c'est une requête POST dont il faut donner la difficulté, le nonce et le hash.

Dans le cas de chaque requête, la clé de l'API qui permet d'identifier notre équipe est transmise. Sans cette clé, la webservice refuse toute requête transmise.

Pour la requête `generate_work`, nous sommes parvenus à interroger le webservice par sockets en forgeant la requête http et en parsant la réponse à la main, comme mentionné dans le bonus du sujet. En ce qui concerne `validate_work`, cependant, nous n'y sommes pas parvenus, notamment à cause de la présence du body.

Scénario de test

Nous avons mis en place Docker pour pouvoir tester le système sur plusieurs machines et accélérer la solution des tâches. Nous avons un fichier Dockerfile avec la configuration nécessaire pour simuler que le ou les clients soient dans le même réseau local que le serveur et qu'ils aient la même IP que celui-ci, comme s'il s'agissait d'une seule machine. Ceci nous permet d'avoir plus de *workers* qui essayent de résoudre la tâche donc la solution est trouvée plus rapidement.

Pour utiliser Cyber Sparkle peut être utilisé sur une ou plusieurs machines, nous expliquerons d'abord l'utilisation basique avec un ordinateur, puis comment en utiliser plusieurs. Vous trouverez les fichiers "cybersparkle1.0-client.jar" et "cybersparkle1.0-serveur.jar" dans ce même dossier, ainsi qu'une image Docker "client". Si vous récupérez le projet depuis le dépôt GitHub vous pouvez régénérer les fichiers vous-même.

Mining 101

Vous aurez besoin d'un ordinateur avec Java 21 et Maven ainsi que d'un terminal ou un invite en ligne de commandes (CLI), nous utilisons zsh mais vous pouvez adapter les commandes à votre terminal.

1. Si vous avez récupéré le projet du dépôt Git vous devez d'abord le *builder* pour générer les fichiers .jar avec les commandes suivantes :

```
mvn clean
mvn install
```

2. Ouvrez deux fenêtres de terminal ou CLI et naviguez dans le dossier où se trouvent les fichiers .jar. Veuillez bien identifier une fenêtre "serveur" et une fenêtre "client" et respecter l'ordre des commandes afin de ne pas avoir des problèmes.

3. Lancez la commande suivante dans la fenêtre serveur :

```
java -jar cybersparkle-1.0-serveur.jar
```

Le serveur va commencer à écouter les connexions au port 1337 de l'ordinateur.

4. Lancez la commande suivante dans la fenêtre client :

```
java -jar cybersparkle-1.0-client.jar
```

Un *three-way handshake* a lieu entre le serveur et le client pour vérifier l'identité du client, voici les messages affichés :

```
$ send :WHO_ARE_YOU_?
Received: ITS_ME
is ITS_ME = true
send :GIMME_PASSWORD
Received: PASSWD 12345
send :HELLO_YOU
Received: READY
send :OK
```

Capture d'écran du terminal "serveur"

```
Connected to server.
Socket[addr=localhost/127.0.0.1,port=1337,localport=56040]
Listening to server...
Waiting for message...
Received: WHO_ARE_YOU_?
WHO_ARE_YOU_?
send :ITS_ME
Listening to server...
Waiting for message...
Received: GIMME_PASSWORD
GIMME_PASSWORD
send :PASSWD 12345
Listening to server...
Waiting for message...
Received: HELLO_YOU
HELLO_YOU
send :READY
Listening to server...
Waiting for message...
Received: OK
OK
Listening to server...
Waiting for message...
```

Capture d'écran du terminal "client"

Le client répond "OK" quand il est prêt à reçevoir des tâches.
Vous pouvez arrêter à tout moment l'exécution du programme avec les touches CTRL + C.

5. Vous pouvez taper `help` dans la fenêtre serveur pour afficher un menu avec les commandes disponibles. Voici un exemple du résultat :

```
• status - display informations about connected workers
• solve <d> - try to mine with given difficulty
• cancel - cancel a task
• help - describe available commands
• quit - terminate pending work and quit
```

Capture d'écran du terminal "serveur" suite à la commande "`help`"

Pour résoudre une tâche avec une difficulté *d*, exécutez la commande suivante en remplaçant *d* par le niveau de difficulté (entier de 1 à 32) :

```
solve <d>
```

La résolution de la tâche commencera et vous verrez sur la fenêtre client que des instructions pour la résolution sont reçue. Si la tâche a déjà été résolue ou lorsque la solution est trouvée, elle s'affiche sur la fenêtre serveur.

6. Vous pouvez demander le statut de la tâche lors de son exécution avec la commande dans la fenêtre serveur :

```
status
```

7. Pour quitter le programme, exécutez la commande suivante dans la fenêtre serveur :

```
status
```

Miner avec plusieurs machines

Lorsque la difficulté des tâches augmente, le temps de résolution augmente aussi, c'est pourquoi il est possible de connecter plusieurs machines pour accélérer le processus (comme de vrais mineurs de Bitcoin !). Pour ce faire, vous aurez besoin de plusieurs ordinateurs (2+), du fichier .jar pour le serveur dans un ordinateur et des images Docker dans chacune des autres machines (qui seront les clients ou workers).

Pour simplifier ce processus, nous avons mis en place Docker pour générer une image qui peut être réutilisée. Si vous avez déjà cette image, vous pouvez avancer à l'étape 3.

Pour générer l'image, vous avez besoin de Docker que vous pouvez installer avec Homebrew, et de Docker Desktop.

1. Lancez Docker Desktop dans la machine serveur.

2. Ouvrez une fenêtre du terminal ou CLI et déplacez vous dans le dossier racine du projet. Exécutez les commandes suivantes pour *builder* le projet, générer une image Docker et l'enregistrer :

```
mvn clean
mvn install
docker image build -t client .
docker image save -o target/client client
```

Un fichier intitulé "client" devrait apparaître maintenant dans le dossier *target* du projet. Envoyez cette image à toutes les machines client (par exemple en utilisant Google Drive car l'image peut dépasser la taille maximale pour les emails) .

3. Vérifiez l'adresse IP locale du serveur, vous pouvez suivre ce guide.

4. Naviguez dans le dossier avec le fichier "S8_projet_reseau-1.0-SNAPSHOT-serveur.jar" et exécutez la commande suivante pour lancer le serveur :

```
java -jar cybersparkle-1.0-serveur.jar
```

5. Démarrez Docker Desktop sur le ou les ordinateurs qui seront les clients pour démarrer automatiquement Docker daemon, vous pouvez le vérifier avec la commande :

```
docker info
```

Si Docker daemon est actif, vous devriez voir des informations en commençant par "Client [...]" ainsi que la version de Docker utilisée.

6. Ouvrez une fenêtre du terminal ou CLI dans le.s ordinateur.s client.s, déplacez vous dans le dossier qui contient le fichier "client" et exécutez les commandes :

```
docker image load -i client
docker run client:latest <adresse_ip_serveur>
```

Vous pouvez vérifier que l'image s'exécute avec la commande suivante depuis une autre fenêtre du terminal :

```
docker ps
```

7. Vous pouvez maintenant interagir avec le serveur comme quand il n'y avait qu'une seule machine. Vous pouvez utiliser `help` pour afficher les options disponibles et `solve <d>` pour résoudre une tâche de difficulté d. Vous verrez dans le terminal des clients que la valeur de l'incrément (`inc`) correspond au nombre de clients (ordinateurs).

Problèmes rencontrés

Un des problèmes rencontrés lors du projet était la connexion avec le webservice, cependant avec plusieurs tentatives et diverses configurations, il était nécessaire de prendre en compte tous les paramètres de connexion tel que l'url bien découpé, le port et la connexion en tant que site sécurisé, étant basé en https.

Un autre des problèmes que nous avons rencontrés est au moment de valider le nonce. Nous avons testé beaucoup d'algorithmes différents pour miner les données, car le webservice refusait toujours certaines difficultés, comme le 2 ou le 4. Le souci venait du fait que nous ne conservions pas un zéro supplémentaire pour les nonce en hexadécimal qui avaient un nombre de chiffres impairs. La traduction par le webservice était donc erronée et le hash calculé ne correspondait pas. Après discussion avec M. Bertrand, nous avons réussi à résoudre le problème, et nous avons pu miner toutes les difficultés jusqu'à 9. La 10 reste trop gourmande en temps et en énergie, et nous ne pouvions pas immobiliser nos machines pendant plus d'une nuit.

Le reste du projet s'est déroulé sans trop d'encombres, nous avons pu avancer tranquillement et fournir un projet fonctionnel.

CONCLUSION

Résumé du projet

Dans le cadre de ce projet, nous avons développé un système de minage distribué qui permet de résoudre des tâches de minage en parallèle avec un ou plusieurs clients qui peuvent se connecter à un serveur. Le serveur unique suit un protocole spécifique pour communiquer avec le ou les clients afin de le faire de manière cohérente, fiable et structurée. Il écoute les connexions entrantes, génère les tâches pour les clients, les envoie et les valide ensuite auprès de l'API raizo. Les clients se connectent au serveur, restent à l'écoute s'ils sont disponibles pour recevoir une tâche puis commencent sa résolution en utilisant l'algorithme de hachage SHA-256. Si le client trouve une réponse, il l'envoie au serveur puis reste disponible pour recevoir d'autres tâches.

Pour faciliter les tests et l'utilisation du système, nous avons mis en place Docker pour utiliser plusieurs machines, ce qui accélère la résolution des tâches, car on peut tester plus de possibilités simultanément. Nous avons également créé des énumérations pour éviter les erreurs de protocole et des exceptions personnalisées pour gérer les commandes inconnues.

Dans l'ensemble, ce projet nous a permis de mettre en pratique les connaissances acquises dans le cours de Réseaux informatiques, de développement en Java et l'utilisation de Docker. Nous avons rencontré quelques problèmes, notamment au moment de valider le nonce, mais nous avons réussi à les résoudre (*merci pour votre aide* ^^) et nous avons acquis de nouvelles connaissances en cours de route.

Perspectives d'amélioration

Nous sommes fiers du projet que nous avons réalisé, mais nous considérons tout de même qu'il y a des perspectives d'amélioration, telles que l'optimisation du code pour réduire le temps de résolution des tâches et la mise en place d'un site (sur GitHub pages par exemple) pour stocker la documentation technique et la rendre plus accessible.

Un aspect qui est essentiel si cette application devrait être déployée en production est de modifier la façon dont la clé pour l'API est stocké. Pour le moment elle se trouve dans la classe "ServerConnected" mais ceci n'est pas sécurisé et permet à d'autres personnes de se connecter à notre compte d'équipe. Une possibilité est d'avoir une variable d'environnement pour la stocker. Il est possible aussi d'avoir un fichier de configuration que chacun des membres de l'équipe a dans le projet, mais qui n'est pas inclus dans le dépôt Git, ainsi qu'une copie *backup* de la clé dans un logiciel de gestion de mots de passe.