Miage M1 - Réseaux

Projet - Distributed mining

En bref

Vous devez, par groupe de 4 personnes, **implémenter un système permettant de déléguer une tâche à un ou plusieurs *workers*, et d'attendre les résultats pour les consolider.**

Présentation détaillée

Une tâche consiste en la recherche d'un *hash* bien spécifique.

Architecture

La recherche des *hash* se fera par un à n *workers* qui devront être pilotés par un serveur centralisé. La communication entre eux se fera en TCP en utilisant l'api *socket*. Le protocole à implémenter est détaillé en annexes.

Serveur

Le serveur est en charge de récupérer les tâches auprès d'une *webapp,* et de les distribuer auprès des workers connectés. A tout moment, il peut leur demander leur statut. Il peut également leur signifier l'annulation de la tâche.

Lorsqu'un client lui notifie un résultat, le serveur doit le valider auprès de la *webapp* et le cas échéant, avertir les autres workers de stopper la recherche en cours.

Il doit être possible d'interagir avec le serveur via une interface en ligne de commande (CLI). L'utilisateur peut ainsi démarrer une tâche, annuler la tâche en cours, demander le statut des workers, …

Worker

Les *workers* se connectent au serveur, et se mettent en attente d'une tâche. Lorsqu'ils reçoivent la demande du serveur, ils doivent commencer la recherche d'un *nonce* correspondant aux critères demandés.

Quand un worker trouver un *nonce* qui convient, il doit envoyer son résultat au serveur qui validera la solution en appelant un *webservice* exposé par la *webapp.* Si le *nonce* est correct, le serveur avertira les autres workers d'abandonner leur recherche.

Tâches

Une tâche consiste, à partir d'un jeu de données fourni, à trouver un *hash* débutant par n zéros (n étant la difficulté de la tâche). Il faut pour cela trouver le *nonce* à ajouter à la donnée pour obtenir le bon *hash*.

Exemple : avec la donnée « helloworld », et une difficulté de 2, il faut ajouter la chaîne « B » pour obtenir un hash débutant par « 00 » (001a3670f91e251537c…).

Cette recherche d'un *hash* débutant par un certain nombre de zéros est la méthode utilisée notamment par le Bitcoin pour valider ses transactions. On appelle ce procédé la *Proof Of Work* (par opposition à la *Proof Of Stake* utilisée par Ethereum).

Les tâches sont fournies par une application web accessible par Internet à l'adresse :

https://projet-raizo-idmc.netlify.app/

Elle est constituée d'une interface permettant de consulter l'avancement des tâches, et de *webservices* permettant de récupérer une tâche et valider un *nonce*. Vous pouvez vous authentifier sur le d*ashboard* avec votre compte UL.

Travail à réaliser (et à rendre)

Vous devrez fournir un fichier jar (ou deux) comprenant :

- un serveur, pilotable en ligne de commande, qui implémente le protocole décrit en annexe
- un client qui peut se connecter au serveur et interagir avec ; il sera instancié de 1 à n fois pour simuler plusieurs clients

Vous disposerez d'un squelette d'application en ligne de commande pour vous aider à démarrer le développement du serveur.

Vous devrez accompagner vos livrables d'un dossier d'au moins une dizaine de pages qui explique la répartition du travail dans l'équipe, l'architecture du projet, les choix techniques qui ont été faits, les difficultés rencontrés, les solutions trouvées, ce qui est implémenté et comment, ce qui ne l'est pas, comment vous avez testé le projet, … bref, toute la vie du projet de sa conception à sa livraison.

— —

Les appels aux *webservices* permettant d'obtenir une tâche et de la valider devront des faire uniquement avec les méthodes disponibles dans le jdk. Aucune librairie externe ne sera autorisée (OkHttp, Apache Httpclient, Retrofit, …).

Bonus : pour les plus aguerris, il sera possible d'interroger les webservices « à la main», c'est à dire en communiquant avec le serveur par socket, en forgeant les requêtes http et en décodant manuellement les réponses.

— —

Concernant la recherche du *nonce*, nous allons utiliser un algorithme de répartition assez simple implémenté dans le protocole par la commande NONCE. Le client n va démarrer la recherche avec le nonce (n-1), et devra l'incrémenter du nombre de workers à chaque essai. Ainsi, tous les nonces seront testés.

Exemple de répartition des nonces avec 3 workers numérotés 1, 2 et 3 :

#1 0 3 6 …

#2 1 4 7 …

#3 2 5 8 …

Bonus : pour les plus aguerris, il sera possible d'imaginer un protocole plus robuste de distribution des nonces, notamment pour gérer la fin inattendue d'un worker (à documenter dans le dossier, ou encore mieux, à implémenter ^^).

Critères d'évaluation

Le projet sera évalué sur les critères suivants :

- Le serveur doit être écrit en Java 11 ou supérieur
- Il doit pouvoir gérer (efficacement) plusieurs clients simultanés
- La communication avec le serveur se fera par socket, sur le port 1337
- Le projet doit être publié sur github ou gitlab, et doit posséder un *readme* qui explique brièvement ce qu'il fait et comment le lancer
- Il doit être simple à mettre en place pour un utilisateur lambda (scripts de lancement, …)
- **Tous les membres du groupe doivent participer au projet !**
- Le serveur **ne doit pas utiliser de librairies toutes faites**, uniquement le jdk
- L'intégralité du protocole doit être implémenté
- Il doit être « bien écrit » : code documenté, maintenable, évolutif
- Il doit être robuste : je dois pouvoir le déployer tel quel en production (ou presque)
- Il doit être fourni avec un client ou un scénario de test.
- S'il est écrit en Java, il ne doit pas utiliser Lombok ou autre librairie de génération de code !

Le code doit être **documenté** : les commentaires doivent être réguliers et pertinents, ne pas expliquer ce qui est fait mais pourquoi c'est fait.

Le code doit être **maintenable** : n'importe qui doit pouvoir intervenir facilement pour corriger le code au besoin. Le code doit être compréhensible.

Le code doit être **évolutif** : pensez à combien de temps vous prendrait d'ajouter une fonctionnalité.

Je suis disponible si vous avez des questions ou des besoins de précision du sujet.
Bonne chance !

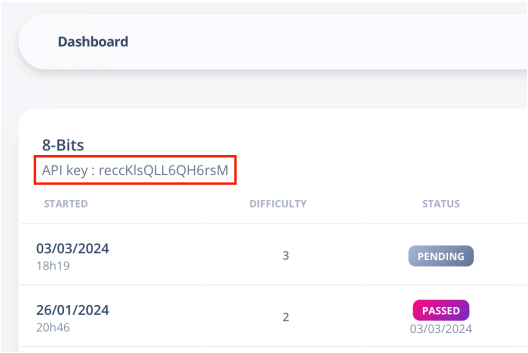
Annexe - Webservices

Les web services sont accessibles à l'adresse de base :

https://projet-raizo-idmc.netlify.app/.netlify/functions

Pour pouvoir les utiliser, il faut s'authentifier. Récupérez pour cela l'identifiant de votre groupe (disponible dans la page d'accueil de la *webapp*) et fournissez le via le header `Authorization`.

Exemple :




```
GET /xxx HTTP/1.1
Accept: application/json
Authorization: Bearer reccKlsQLL6QH6rsM
...
```

En cas de succès, les *webservices* renvoient un code http 2xx et d'éventuelles données (voir description détaillée plus loin). En cas d'erreur, ils renvoient un code 4xx ou 5xx et le *payload* contient alors la description de l'erreur sous la forme d'un objet json possédant une unique propriété `details`.

Exemple : en cas d'argument invalide, un code « 400 Bad Request » sera renvoyé et le *payload* sera `{ "details" : "Invalid args" }`.

generate_work

Demande au serveur de générer une nouvelle tâche (i.e. des données aléatoires à hasher). La difficulté doit être passée en *query string* via le paramètre d. La valeur de d doit être comprise entre 1 et 32.

- Si la tâche est créé avec succès, la fonction renvoie un code 201 et les données de la tâche.

```
{"data": "njkdnac...vfevfvztrz"}
```

- Si la difficulté a déjà été « résolue », la fonction renvoie un 409 et un message d'erreur.

```
{"details": "Difficulty d already solved"}
```

Exemple de requête (incomplète) :

```
GET /.netlify/functions/generate_work?d=5 HTTP/1.1
```

Exemple de réponse (incomplète) :

```
HTTP/1.1 201 Created

…
{"data":"b2uk59qg9jn6jytuxz42g684ycqmsubvnbhmjofaltf824vuml3l"}
```

validate_work

Permet à une équipe de valider une tâche complétée. Les 3 informations à fournir sont la difficulté (nombre), le *nonce* et le *hash* (chaîne hexa).

Exemple de requête (incomplète) :

```
POST /.netlify/functions/generate_work?d=5 HTTP/1.1

{d: 3, n: "f42", h: "000a23efc..."}
```

Exemple de réponse (incomplète) :

```
HTTP/1.1 200 OK

…
```

Si la difficulté a déjà été « résolue », le serveur renverra une erreur « 409 Conflict ».

Annexe - Protocole

Le protocole est textuel, les commandes et les réponses sont envoyées par lignes. Une ligne commence toujours par une instruction en majuscules (voir liste ci-dessous), des paramètres éventuels séparés par des espaces, et un marqueur de fin de ligne (retour chariot).

COMMANDE [ESPACE PARAMETRE]{0, n} RETOUR_CHARIOT

Exemple de représentation en java : "SOLVE 4\n"

Instructions serveur → client

WHO_ARE_YOU_?

Première commande envoyée par le serveur à un client qui vient de se connecter. Le client devra répondre « ITS_ME ». Permet de vérifier que ce dernier comprend bien le protocole et peut ainsi interagir avec le serveur.

GIMME_PASSWORD

Envoyé juste après avoir reçu « ITS_ME ». Demande au client le mot de passe pour se connecter. Idéalement, le mot de passe est aléatoire, généré par le serveur au démarrage et fourni aux clients par un canal sécurisé.

OK

Envoyé en réponse à la commande READY pour informer le client que l'on a bien pris en compte sa participation.

HELLO_YOU

Confirme la validité du mot de passe. Le client est désormais considéré comme connecté, il pourra à ce titre participer à la résolution d'une tâche.

YOU_DONT_FOOL_ME

Indique au client que le mot de passe est incorrect. La connexion est fermée après cet envoi.

SOLVED

Indique aux workers qu'une solution a été trouvée et qu'ils doivent abandonner le travail en cours. Ils peuvent alors envoyer READY pour se préparer au travail suivant.

PROGRESS

Demande l'état d'un worker.

CANCELLED

Indique aux workers qu'ils doivent abandonner le travail en cours. Ils peuvent alors envoyer READY pour se préparer au travail suivant.

NONCE start increment

Indique au client le *nonce* par lequel il doit débuter sa recherche et l'increment à ajouter après chaque essai. Les deux valeurs sont des nombres. Cette instruction vient en complément de PAYLOAD et SOLVE.

PAYLOAD data

Indique au client les données qu'il va utiliser pour trouver le hash qui va bien. Dans cette version, data est une chaîne de caractères utf-8. Cette instruction vient en complément de NONCE et SOLVE.

SOLVE difficulty

Indique au client la difficulté attendus (i.e. le nombre de zéros par lequel le hash gagnait doit débuter). Cette instruction vient en complément de PAYLOAD et SOLVE.

Ce n'est qu'une fois les 3 instructions PAYLOAD, NONCE, et SOLVE reçues que le minage pourra débuter. A noter que l'ordre de ces trois instructions est indéfini.

Instructions client → serveur

ITS_ME

Instruction qu'un client doit envoyer en réponse à « WHO_ARE_YOU_? ».

PASSWD password

Doit être envoyé en réponse à GIMME_PASSWORD pour fournir le mot de passe de connexion au serveur. Le mot de passe est une chaîne utf-8. Exemple : "PASSWD azerty\n ».

READY

Indique au serveur que ce worker est prêt à exécuter une tâche. Il se met alors en attente de la série de commandes NONCE / SOLVE / PAYLOAD.

FOUND hash nonce

Indique au serveur qu'on a trouvé une solution pour la tâche en cours. Le worker fourni alors le hash trouvé ainsi que le nonce qui a permis de l'obtenir. Le serveur pourra ainsi aisément vérifier que la solution est correcte.

En réponse à la commande PROGRESS, un worker peut répondre de deux manières différentes :

- S'il est en cours de minage, il doit renvoyer **TESTING** suivi du nonce courant en hexadécimal. Exemple : « TESTING 42BE\n »
- Sinon il doit renvoyer **NOPE**

Exemple de session

Messages envoyés entre le serveur et le mineur qui a trouvé la solution.

Serveur	Client
WHO_ARE_YOU_?	
	ITS_ME
GIMME_PASSWORD	
	PASSWD azerty
HELLO_YOU	
	READY
OK	
NONCE 0 1	
PAYLOAD naoo1ef…2m0p7r9	
SOLVE 1	
	FOUND 081734fc…a5f8a713a 3b
	READY
OK	
…	