Projet TPEA (Blockchain)

Pezos®

1 Modalités

Le projet doit être réalisé par groupe de 3 à 4 personnes. Une fois votre groupe établi, il vous sera attribué une identité cryptographique (clé publique/privée) vous permettant de vous authentifier et de participer au réseau ainsi que l'adresse du serveur Pezos®. Le langage de programmation est libre mais les rendus de projets devront être **facilement** compilables sous Linux. La fonction de hachage utilisée sera `Blake2b` (voir TME1 et TME2) et l'algorithme de signature `Ed25519` (voir TME3).

La date de rendu correspond à la dernière semaine de cours durant laquelle il vous sera demandé d'effectuer une présentation de votre implémentation avec un support visuel (slides). La présentation devra durer 10 minutes et sera accompagnée d'une seconde partie de 10 minutes de questions de cours. La notation sera : 50% rendu de projet et 50% oral.

2 Sujet

Pezos® est une blockchain gouvernée par un *dictateur*. Son algorithme de consensus est une *preuve d'autorité*[1] : seul le dictateur a le droit de produire des blocs. En revanche, le dictateur n'est pas le meilleur développeur et les blocs qu'il produit contiennent *systématiquement* des erreurs. Le dictateur s'en remet à ses fidèles utilisateurs pour déceler les erreurs. En récompense, le dictateur a prévu un mécanisme de gratification. Ainsi, pour chaque erreur trouvée, 1 `pez` (la monnaie de Pezos®) sera créé et versé aux utilisateurs ayant trouvé l'erreur. À chaque type d'erreur, on associe un type de `pez` spécifique : si l'on ne trouve qu'un seul type d'erreur, on ne disposera seulement que de ce type de `pez`.

Le dictateur n'a pas une machine puissante. Ainsi, il a décidé que les blocs ne pouvaient être créés que toutes les 10 minutes. Dans ce laps de temps, les utilisateurs peuvent vérifier la validité du dernier bloc produit et injecter une opération de correction. Une fois cette opération injectée (et valide), le dictateur va produire son prochain bloc en l'incluant et en appliquant la gratification. Chaque bloc produit est final et toute opération de correction reçue ne sera valide que pour le dernier bloc produit : si une opération arrive trop tard, elle sera ignorée par le réseau.

Le but de ce projet est donc d'implanter un client de correction capable de se connecter au serveur Pezos®, écoutant les nouveaux blocs et détectant les erreurs pour proposer des corrections afin d'obtenir le maximum de `pez`.

Les sections suivantes décrivent les différents protocoles et structures de données nécessaires pour intéragir avec le client.

3 Serveur

Le serveur de Pezos® utilise un protocole TCP et dispose d'une phase authentification.

3.1 Protocole d'authentification

Pour s'authentifier et pouvoir intéragir avec le réseau, le serveur définit un protocole d'authentification. Le protocole d'authentification, illustré par la figure 1, se déroule en trois temps. Le serveur commence par envoyer une valeur aléatoire au client qu'il doit signer. Puis, le serveur attend du client qu'il lui envoie sa

1. `https://en.wikipedia.org/wiki/Proof_of_authority`

clé publique. Si la clé publique n'est pas autorisée à se connecter au serveur alors le serveur terminera la connexion. Une fois la clé publique envoyée et vérifiée, le serveur attend que le client lui envoie **la signature du hash** de la valeur aléatoire. À l'issue de ce dernier envoi, si la connexion n'a pas été terminée par le serveur, le client est considéré comme connecté et peut commencer à utiliser la couche de messages applicatifs.

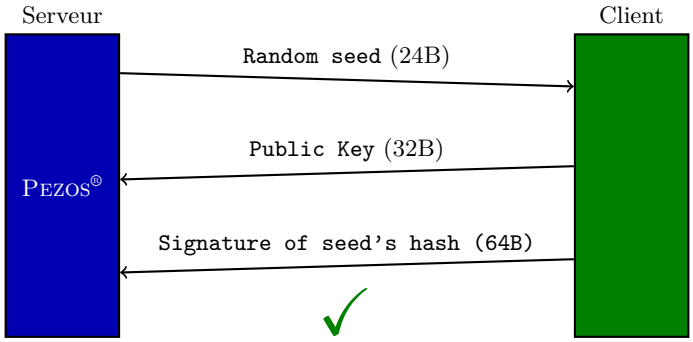

Figure 1 – Protocole d'authentification

⚠Tous les messages échangés sont et doivent être en binaire.
⚠Ils doivent également être préfixés par la taille sur 16 bits du message envoyé. Si ce format n'est pas respecté, le serveur terminera immédiatement la connexion.
⚠Le serveur peut également être amené à vous déconnecter si vous ne terminez pas l'envoi d'un message.

$$\underbrace{0x00202}_{\text{length} = 32}\underbrace{\text{dfd3fb419c78e23806d6be7c2bcc1def9dfcd2d6efcc86260a86e4bd9f0ace6}}_{\text{Ed25519 (32B) public key}}$$

Figure 2 – Exemple de message de clé publique valide (encodé en hexadécimal)

Pour plus de détails, l'ensemble des encodages des messages réseaux, dont les messages d'authentification, sont disponibles dans la section 6.1.

3.2 Protocole applicatif

Une fois authentifié, les utilisateurs peuvent désormais interagir avec la chaîne via la couche applicative. Les messages d'authentifications ne sont plus compris par le serveur. Cette couche applicative comporte 9 types de messages différents :

```
MESSAGES =
  | GET CURRENT HEAD              (TAG = 1)
  | CURRENT HEAD <block>          (TAG = 2)
  | GET BLOCK <level>             (TAG = 3)
  | BLOCK <block>                 (TAG = 4)
  | GET BLOCK OPERATIONS <level>  (TAG = 5)
  | BLOCK OPERATIONS <op list>    (TAG = 6)
  | GET BLOCK STATE <level>       (TAG = 7)
  | BLOCK STATE <state>           (TAG = 8)
  | INJECT OPERATION <op>         (TAG = 9)
```

— Le message `GET CURRENT HEAD` peut être envoyé au serveur pour lui demander la tête courante de la chaîne. Une fois reçu, le serveur répondra à l'aide du message `CURRENT HEAD <block>` accompagné du bloc de tête.
— `GET BLOCK <level>` permet d'obtenir un bloc à un niveau donné, la réponse du serveur sera `BLOCK <block>` si le niveau donné est valide, aucune réponse (ou potentielle déconnexion) sinon.
— `GET BLOCK OPERATIONS <level>` permet d'obtenir les opérations contenues dans un bloc à un niveau donné, la réponse du serveur sera `BLOCK OPERATIONS <op list>`, permettant d'obtenir les opérations contenues dans le bloc donnée, si le niveau donné est valide, aucune réponse (ou potentielle déconnexion).

— `GET BLOCK STATE <level>` permet d'obtenir l'état de la chaîne à un niveau donné, la réponse du serveur sera `BLOCK STATE <state>`, permettant d'obtenir l'état du bloc à ce niveau, si le niveau donné est valide, aucune réponse (ou potentielle déconnexion).
— Finalement, `INJECT OPERATION <op>` permet d'injecter une opération de correction sur le réseau. Pas de réponse du serveur (ou potentielle déconnexion) si l'opération est invalide.

⚠À chaque nouveau bloc (toutes les 10 minutes), le serveur notifie tous ses pairs authentifiés via un message `BLOCK`. Votre client devra donc lire ce nouveau message périodique.

Pour tous ces messages, les formats de données utilisés sont décrits dans la section 6.1.

⚠Comme pour le protocole d'authentification, tous les messages doivent être et seront préfixés de la taille totale du message.

4 Structures de données, erreurs et corrections

Le client peut suivre la boucle d'intéraction illustrée par la figure 3 :

1. Le serveur dispose d'une nouvelle tête qu'il notifie au client ;
2. Le client récolte les informations nécessaires pour vérifier les données annoncées ;
3. Une fois l'erreur repérée, le client injecte une opération de dénonciation puis se met en attente d'un nouveau bloc ;
4. Au bout de 10 minutes, le dictateur va créer un nouveau bloc sur le serveur qui sera ensuite notifié aux client, etc.

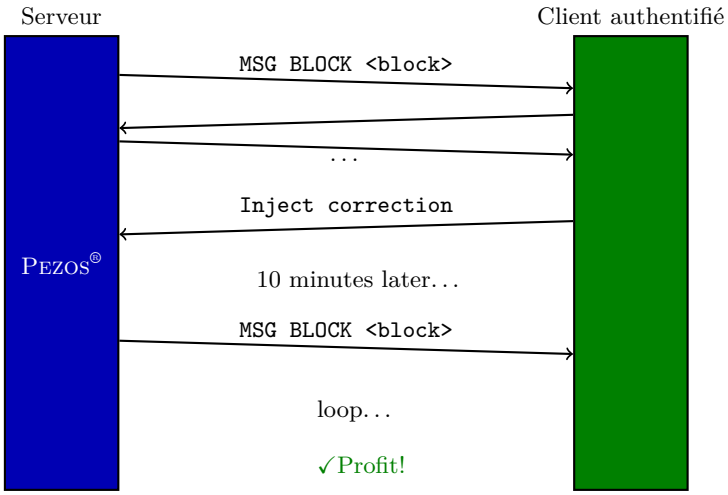

Figure 3 – Boucle d'interaction

Les sections suivantes présentes les différentes structures de données et les erreurs potentielles que l'on peut corriger.

4.1 Blocs

Comme énoncé plus haut, à chaque nouveau bloc, une erreur sera présente dans ce bloc. Les blocs sont composés de 6 champs potentiellement erronées :

— level : le niveau du bloc level n'a pas d'erreur
— predecessor : le hash du bloc précédent
— timestamp : la date du production du bloc
— operations hash : la "hash list" (*c.f.* section 4.1.4) des opérations contenues dans le bloc
— state hash : le hash de l'état du bloc
— signature : la signature du bloc

4.1.1 Level

Le niveau est un entier 32 bits. Cette valeur sera **toujours** correcte : le dictateur ne se trompe jamais sur le niveau du bloc.

4.1.2 Predecessor

Le hash du bloc précédent peut être erronné. Pour vérifier la validité de cette valeur, il est nécessaire d'obtenir le bloc précédent et de hacher sa représentation encodée.

4.1.3 Timestamp

Dans Pezos®, les valeurs temporelles sont encodées sur un entier 64 bits représentant le nombre de secondes écoulées depuis le 1^{er} janvier 1970 (epoch). Cette valeur peut également être erronnée. Pour être valide, le temps du bloc doit être **au moins** espacé de 10 minutes par rapport au bloc précédent comme décrit par le protocole économique.

4.1.4 Operations hash me

Le bloc contient la **hash list** [2] des opérations du bloc. Cette valeur peut-être erronnée. Pour calculer la bonne valeur, il faut récupérer la liste des opérations contenues dans le bloc et procéder au calcul suivant :

$$\texttt{ops\_hash}(OPS) = \begin{cases} \text{32 octets à 0} & \text{si la liste d'opération OPS est vide} \\ \mathcal{H}(op) & \text{si op est le seul élément de la liste} \\ \mathcal{H}(\texttt{ops\_hash}(OPS \setminus \{\texttt{last op}\}), \mathcal{H}(\texttt{last op})) & \text{sinon, avec } \texttt{last op} \text{ la dernière opération} \end{cases}$$

4.1.5 State hash

Le bloc contient le hash de l'état (encodé) résultant de l'application de ce bloc. Cette valeur peut être erronnée. La correction doit être le hash correct de l'état (encodé). Pour plus de détails sur l'encodage, voir la section 6.4.

4.1.6 Signature

Le bloc contient la signature produite par le dictateur. Celle-ci peut également être erronnée. La clé publique du dictateur permettant de précéder à la vérification de cette signature peut être retrouvée dans l'état de la chaîne. Attention : le dictateur signe un bloc mais, évidemment, il ne peut pas signer la signature qu'il n'a pas encore produite. Ainsi, les données signées sont l'ensemble du bloc (encodés) sans le champ signature.

⚠Comme pour l'authentification, nous ne signons pas directement la donnée mais le hash de cette donnée. Vous ferez attention à également hasher le sous-ensemble du bloc avant de procéder à votre vérification de signature.

4.1.7 Exemple d'encodage de bloc

Exemple d'encodage de bloc :

```
level:           44
predecessor:     1c80203a30e5de4d980cc555131d1b4a4750edc82c0c443179d88de1ae4f6cdf
timestamp:       2021-10-10 15:21:09
operations hash: 0000000000000000000000000000000000000000000000000000000000000000
context hash:    22a00d1c8c0fbaefedd71ddb83d455033efd259a8f0adf189b9f850a0d1945f2
signature:       cc3faffc696c86db13d50752fdb7edd0ee1ce19ab350f60899939fc139d58996419c1
                 3b812b7f005fafaf23924d2f1df555036bc61e7b67cb679375e5756b306
```

s'encode en la valeur binaire suivante :

```
0000002c1c80203a30e5de4d980cc555131d1b4a4750edc82c0c443179d88de1ae4f6cdf00000000
616304e5000000000000000000000000000000000000000000000000000000000000000022a00d1c
8c0fbaefedd71ddb83d455033efd259a8f0adf189b9f850a0d1945f2cc3faffc696c86db13d50752
fdb7edd0ee1ce19ab350f60899939fc139d58996419c13b812b7f005fafaf23924d2f1df555036bc
61e7b67cb679375e5756b306
```

L'encodage complet des blocs est disponible en section 6.2

2. https://en.wikipedia.org/wiki/Hash_list

4.2 Opérations

Les opérations possibles sont :

```
Operations =
  | BAD PREDECESSOR <hash>
  | BAD TIMESTAMP <time>
  | BAD OPERATIONS HASH <hash>
  | BAD CONTEXT HASH <hash>
  | BAD SIGNATURE
```

Chacune de ces opérations est utilisée pour proposer une correction à l'erreur contenue dans le bloc de tête. Par exemple, si la signature du bloc est mauvaise alors le client devra injecter l'opération `BAD SIGNATURE` à l'aide du message réseau `INJECT OPERATION`. Pour être valide, en plus du contenu, on doit ajouter à l'opération : la clé publique du manager et la signature. La signature doit être produite à partir de la concaténation encodé du contenu et de la clé publique. sign(publicKey, concat(operation, publickey))

L'ensemble des encodages possibles sont décrits dans la section 6.3.

4.3 État

L'état contient trois composants :
— La clé publique du dictateur ;
— Le temps **correct** du bloc précédent ;
— L'ensemble des comptes utilisateurs : *i.e.* les clés publiques et les `pez` spécifiques.
Il permet notamment de consulter le montant de `pez` dont chaque utilisateur dispose.

5 Conseils

— Munissez-vous, au plus tôt, de fonctions d'affichage/logging propres : cela est extrêmement précieux pour afficher les données et comprendre les traces d'exécution du programme pour chercher les différents bugs.
— Déterminez au plus tôt les types de données des valeurs que vous manipuler. Vous éviterez ainsi les problèmes de type `bytes` vs. `hex`.
— Prototypez votre implémentation puis lorsque vous obtenez une résultat positif, nettoyez et commentez votre code. La suite sera ainsi plus simple. De plus, des points de styles peuvent être enlevés ou ajoutés [3] en fonction de la qualité du code.
— Définissez des scénarios de tests locaux au plus tôt aprè avoir récupérés données du réseau : il n'est pas raisonnable d'attendre 10 minutes l'arrivée d'un nouveau bloc pour tester votre nouveau patch.
— Un REPL peut être utile pour pouvoir intéragir manuellement avec le serveur.
— Pour finir, soyez aimables avec la machine du dictateur. Celle-ci n'est (vraiment) pas puissante et vulnérable au spam afin de faciliter la vie de ses utilisateurs. De plus, bien que le serveur ait été éprouvé, il n'est pas exclus que le serveur tombe en marche. Le cas échéant, n'hésitez pas à contacter le dictateur.

3. Le dictateur a cependant de hauts standards...

6 Annexe – Encodages

6.1 Encodage des messages

MESSAGE
=======

Name	Size	Contents
nb bytes in next field	2 bytes	16-bit integer
message	Determined from data	MSG

6.1.1 Messages d'authentification

MSG: RANDOM SEED
================

Name	Size	Contents
seed	24 bytes	bytes

MSG: PUBLIC KEY
===============

Name	Size	Contents
user public key	32 bytes	bytes

MSG: SIGNATURE
==============

Name	Size	Contents
signature	64 bytes	bytes

6.1.2 Messages applicatifs

MSG: GET CURRENT HEAD (tag 1)
=============================

Name	Size	Contents
Tag	2 bytes	16-bit integer

MSG: CURRENT HEAD (tag 2)
=========================

Name	Size	Contents
Tag	2 bytes	16-bit integer
block	172 bytes	BLOCK

MSG: GET BLOCK (tag 3)
======================

Name	Size	Contents
Tag	2 bytes	16-bit integer
level	4 bytes	32-bit integer

MSG: BLOCK (tag 4)
==================

Name	Size	Contents
Tag	2 bytes	16-bit integer
block	172 bytes	BLOCK

MSG: GET BLOCK OPERATIONS (tag 5)

Name	Size	Contents
Tag	2 bytes	16-bit integer
level	4 bytes	32-bit integer

MSG: GET STATE (tag 7)

Name	Size	Contents
Tag	2 bytes	16-bit integer
level	4 bytes	signed 32-bit integer

MSG: BLOCK OPERATIONS (tag 6)

Name	Size	Contents
Tag	2 bytes	16-bit integer
# bytes in next field	2 bytes	16-bit integer
operations	Variable	sequence of OPERATION

MSG: BLOCK STATE (tag 8)

Name	Size	Contents
Tag	2 bytes	16-bit integer
state	Determined from data	STATE

MSG: INJECT OPERATION (tag 9)

Name	Size	Contents
Tag	2 bytes	16-bit integer
operation	Determined from data	OPERATION

6.2 Encodage d'un bloc

BLOCK
=====

Name	Size	Contents
level	4 bytes	32-bit integer
predecessor	32 bytes	bytes
timestamp	8 bytes	64-bit integer
operations_hash	32 bytes	bytes
state_hash	32 bytes	bytes
signature	64 bytes	bytes

6.3 Encodage des opérations

SIGNED OPERATION
================

Name	Size	Contents
contents	Determined from data	OPERATION
user public key	32 bytes	bytes
signature	64 bytes	bytes

OPERATION: BAD PREDECESSOR (tag 1)
==================================

Name	Size	Contents
Tag	2 bytes	16-bit integer
hash	32 bytes	bytes

OPERATION: BAD TIMESTAMP (tag 2)
================================

Name	Size	Contents
Tag	2 bytes	16-bit integer
time	8 bytes	64-bit integer

OPERATION: BAD OPERATIONS HASH (tag 3)
======================================

Name	Size	Contents
Tag	2 bytes	16-bit integer
hash	32 bytes	bytes

OPERATION: BAD CONTEXT HASH (tag 4)
===================================

Name	Size	Contents
Tag	2 bytes	16-bit integer
hash	32 bytes	bytes

OPERATION: BAD SIGNATURE (tag 5)

Name	Size	Contents
Tag	2 bytes	16-bit integer

6.4 Encodage de l'état

STATE

Name	Size	Contents
dictator public key	32 bytes	bytes
predecessor_timestamp	8 bytes	64-bit integer
nb bytes in next sequence	4 bytes	32-bit integer
accounts	Variable	sequence of ACCOUNT

ACCOUNT

Name	Size	Contents
user public key	32 bytes	bytes
level pez	4 bytes	32-bit integer
timestamp pez	4 bytes	32-bit integer
operations_hash pez	4 bytes	32-bit integer
context_hash pez	4 bytes	32-bit integer
signature pez	4 bytes	32-bit integer

predecessor pez