Rapport projet Scrabblos

Bruno PINTO, Claire BARDOUX, Cyann DONNOT

1 Réponses

Question 0

Nous assurons qu'un auteur n'injecte pas plusieurs lettres dans un bloc. En effet, à chaque fois qu'un utilisateur reçoit un mot, il l'ajoute dans son *word_pool* avec la méthode $WordStore.add(self, item)$, qui se trouve dans le fichier `src/store.py` et qui vérifie chacune des propriétés suivantes afin de prouver leur validité :

- Chaque auteur possède au plus une lettre dans ce mot.
- Toutes les lettres sont de la même période que le mot
- L'auteur du mot en est bien le créateur légitime ($Word.check\_signature(self)$).

Question 1

Nous avons choisi de partir de zéro pour coder le projet en Python. Suite à une lecture un peu trop rapide du sujet, nous avons commencé directement par la version "roue libre". Il n'existe donc pas à ce jour d'auteur ou de politicien en version "tour par tour".

Question 2

Ci-dessous la description des fonctions utiles au consensus :

- `def scrab_score(c):` Attribuer un score à chaque lettre. Nous avons basé notre score selon les règles française du *Scrabble*. Pour plus d'informations voir `https://fr.wikipedia.org/wiki/Lettres_du_Scrabble#Fran\unhbox\voidb@x\setbox\z@\hbox{c}\accent24cais`.
- `def str_score(string):` Attribuer un score à un mot en additionnant le score de chaque lettre.
- `def bestWord(wordS):` Renvoi le mot ayant le score le plus élevée parmi un ensemble de mots.

Consensus: Pour lancer le consensus nous avons choisi d'avoir une horloge par client ConsensusCall qui permet d'envoyer une requête de consensus à intervalles réguliers (10 sec de base). Tous les clients l'ignorent à l'exception du leader (auteur du dernier bloc de la chaîne) qui envoie le mot qu'il pense être le meilleur (jugement effectué grâce à la méthode bestWord(wordS)). Chaque client reçoit alors un bloc dont il vérifie la validité et retourne vrai ou faux suivant s'il souhaite l'ajouter comme bloc suivant. Le résultat est vérifié par chaque client, s'il est en majorité positive, le bloc est ajouté sinon un *kick* est envoyé à l'encontre du leader. Il est alors expulsé de la chaîne et un nouveau leader est choisi aléatoirement pour relancer le consensus.

Question 3

Nous n'avons pas écrit la version "tour par tour". Cependant, pour passer d'un consensus "tour par tour" à un "roue libre" dans le cas d'un serveur central, il faudrait écouter en continu les propositions des clients pendant un certain temps et ensuite couper les écoutes de tous les candidats pour faire le consensus et obtenir le meilleur mot à ajouter à la chaîne. Cet arrêt d'écoute est utile pour éviter d'avoir des modifications pendant ce dernier.

Question 4

Notre implémentation est résistante à l'usurpation d'identité, chaque auteur possède une clef privé ed25519 avec laquelle il encode chaque lettre ou mot qu'il génère. Son **id** étant la clef public, tout le monde peut vérifier qu'un mot a bien été crée par son propriétaire.

Nous ne gérons pas l'ajout de lettres dans le sac initial. En effet comme on envoie une simple liste de caractère rien n'oblige les auteurs de choisir une autre lettre. Pour gérer ce cas, il faudrait que le serveur signe chaque lettre par le serveur au moment de l'envoie pour rendre différentes deux lettre ayant le même nom. Il faudrait ensuite partager le sac avec tout le monde au début pour qu'on puisse vérifier qu'aucune lettre n'est envoyé deux fois.

Question 5

Dans notre cas nous avons directement la solution sans serveur central. Nous avons un serveur de connexion qui permet d'instancier le routeur et d'accéder à la liste des autres participants. Le routeur permet d'envoyer des requêtes à tous les participants connus. Chaque routeur ne gère qu'un unique client.

Question 6

Nous nous appuyons sur un système plus PoW vu que le leader est l'auteur du dernier bloc. Pour passer en PoS, il faudrait choisir le leader en fonction de

son nombre de points dans la blockchain. La gestion des critères se fait dans la fonction *client.Client.consensus(self, _)* et il permettent de savoir si un client applique ou ignore cette requête pour initier un consensus.

2 Implantation

Voici le descriptif de notre implantion, pour le consensus, veuillez vous référer à la section Question 2.

2.1 Serveur

Nous avons choisi de créer un serveur de connexion qui accepte en continue les demande de connexions la bloc Chain et instancie un routeur pour chaque nouveau client. Les routeurs ne font aucun calcul, ils sont justes là pour transmettre les protocoles aux clients connus. Pour évaluer une requête, on reçoit les dictionnaires sérialisés dont chaque clef est un identifiant de requête. Chaque identifiant est associé à une fonction portant le même nom ce qui nous permet d'évaluer cet identifiant directement comme du code pour appliquer cette dernière.

2.2 Client

Il existe une classe client de base dont héritent les auteurs et les politiciens, elle permet de factoriser la gestion des requêtes. Chaque client possède une MessageBox qui récupère les requêtes du serveurs, d'une InputBox qui récupère les entrées dans le cas d'un utilisateur humain et d'une ConsensusCall qui sert d'horloge pour le consensus. La gestion des requêtes se fait de la même manière que dans le routeur.

2.3 Politicien

Pour le politicien, nous avons choisi d'implémenter seulement le bot. Il s'enregistre, récupère les messages dans sa MessageBox, les appliques et lance un thread searching dont le run est la fonction de recherche de mot. Il l'arrête tous les certains temps afin d'avoir toujours un mot à proposer à chaque consensus.

2.4 Auteur

Pour l'auteur, nous avons également choisi de se limiter au bot. Chaque auteur s'enregistre, gère ses requêtes et envoie une lettre dans chaque consensus. Les auteurs quittent la blockchain après avoir envoyé tout leur sac de lettres.

2.5 Store

WordStore et LetterStore sont respectivement le pool de mot et le pool de lettre dans lequel chaque client ajoute ce qu'il reçoit. Les stores vérifient la validité des objets avant chaque ajout ce qui permet d'éviter au maximum les problèmes liés à un éventuel utilisateur malveillant.