Exercices sur Sync HotStuff

Sync HotStuff : Simple and Practical Synchronous State Machine Replication

Article disponible à l'adresse suivante `https://eprint.iacr.org/2019/270.pdf`

Questions sur cet article

- Qu'est-ce que le problème SMR ?
- Quelles sont les propriétés d'un service SMR ? Donner des exemples.

> Le *State Machine Replication* est une implémentation d'un service pour *valider* (on dit aussi *commit* en anglais) et ordonner les requêtes de clients.
>
> L'implémentation est trivialement réalisée par un serveur centralisé (comme c'est le cas pour un serveur Web). Cependant, une telle implémentation n'est pas très résistante aux pannes, ni à la censure (le service Web peut être censuré par un état).
>
> Afin de palier ce problème, on constuit une architecture tolérante aux pannes constituée de N noeuds (appelés également réplicas ou noeuds) dont f peuvent tomber en panne (le nombre maximal f de noeuds en panne dépend de plusieurs hypothèses).
>
> Chaque noeud joue le rôle d'un serveur qui assure (une partie de) ce service : il accepte les requêtes que les clients lui envoie, et il doit avoir un vision d'ensemble de toutes les requêtes envoyées (à lui ou aux autres noeurs) afin de partager, avec les autres noeuds, un *ordre total* sur les requêtes.
>
> Ce service distribué doit garantir les deux propriétés suivantes :
> (1) *Safety* : les noeuds qui ne sont pas en panne doivent valider les mêmes requêtes et dans le même ordre.
>
> (2) *Liveness* : chaque requête d'un client est *inévitablement* (*eventually* en anglais) valider et ordonner par le service (global).
>
> La principale difficulté du problème SMT est de construire l'ordre total (on dit aussi l'*état global*) sur les requêtes. La solution prend la forme d'un *protocole de consensus* BFT (*Byzantin Fault Tolerant*).
>
> Exemple : Une blockchain est un exemple typique du problème SMR : les clients (*wallets*) envoient leurs transactions aux noeuds du réseau, et la blockchain doit les ordonner (les transactions sont ordonnées dans des blocs, et les blocs sont ordonnés entre eux).

- Quel est l'objectif de l'algorithme Sync HotStuff ?

> Sync HotStuff est un algorithme BFT qui résoud le problème SMR. Pour cela, il fait des hypothèses sur le fonctionnement du réseau et sur le nombre maximal de Byzantins possibles.

- Quelles sont les hypothèses réseaux/machines faites par les auteurs ?

Le réseau est supposé *synchrone*, c'est-à-dire qu'il existe existe une constante Δ telle que tout message envoyé par un noeud (ou un client) au temps t doit nécessairement arrivé à destination au temps $t+\Delta$. La constante Δ est à définir en fonction des capacités du réseau, mais également des machines à accepter (traiter) les requêtes.

Sync HotStuff suppose également que tous les noeuds peuvent communiquer *les uns avec les autres*, à travers des canaux de communication *authentifiés* à l'aide de signatures cryptographiques (implémentées par des primitives cryptographiques permettant de manipuler des paires de clés publiques/privées – architecture PKI).

Le protocole suppose également que les horloges des noeuds sont parfaitement synchronisées et qu'il n'y a pas de décalage dans le temps.

- En quoi Sync HotStuff est-il *practical* ?

Les algorithmes *synchrones* ne sont habituellement pas très "pratiques" car ils demandent un grand nombre de "rounds" pour converger. C'est le cas par exemple de l'algorithme de Dolev-Strong qui nécessiterait exactement $f+1$ rounds pour chaque bloc. Contrairement à Dolev-Strong, Sync HotStuff permet de réaliser le consensus en 1 round quand tout se passe bien. Combien de rounds dans le pire des cas ?

Par ailleurs, les algorithmes synchrones nécessitent une exécution en mode *lock-step* où tous les noeuds doivent commencer et terminer chaque round au même moment. En quoi cela est-il difficile à implémenter ? Sync HotStuff ne nécessite pas d'exécuter les noeuds de manière lock-step.

- Quels sont les deux modes de fonctionnement de l'algorithme ?

— *Steady-state* protocol : phase stable du protocole, le (même) leader envoie des blocs de manière continue, et ces blocs sont votés (certifiés) par une majorité de noeuds et ils sont donc commités
— *View-change* protocol : permet de changer de vue (donc de leader) quand un problème est détecté (lenteur dans la production des blocs, blocs en conflits)

- Quels sont les probèmes liés à l'hypothèse de Lock-Step ?

Quelles solutions pour implémenter une hypothèse lock-step ?

- Quelles sont les limites théoriques en nombre de Bizantins selon les hypothèses réseaux/machines pouvant être supportées pour un protocole de consensus ?

$N = 2f + 1$

- Comment pourraient-être choisis les *leaders* (autrement qu'avec un *round-robin* ?

Des idées ? Il s'agit du problème appelé *Leader election*.
"Selecting a unique leader is equivalent to solving the consensus problem"
(J Gray and L. Lamport, *Consensus on Transaction Commit*, 2005)

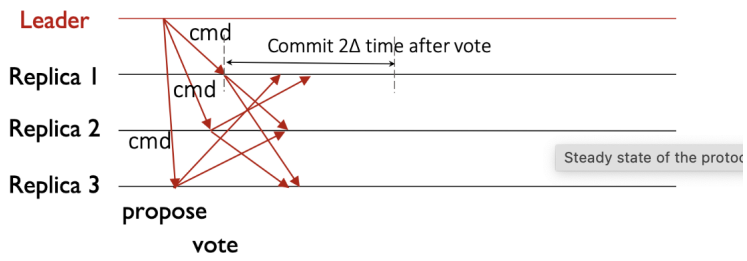
- A quelle vitesse les blocs sont-ils produits ? Commités ?

Si tout se passe bien, les blocs sont commités en 2Δ temps.

- Quelles sont les mécanismes pouvant être utilisés par un attaquant pour empêcher la progression de la blockchain ?

Un leader peut empêcher la blockchain de progresser en ne produisant aucun blocs ou en proposant des blocs en conflit.

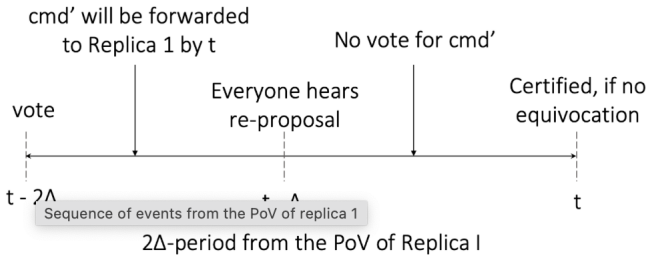
- Donner un exemple d'exécution du protocole *steady state*



- Donner un exemple d'exécution impliquant le protocole *view-change*
- Pourquoi attendre un temps de 2Δ est suffisant pour *commiter* sur un bloc (pourquoi cela assure la *safety*) ?

The 2Δ waiting time suffices for two invariants to be satisfied if the committing replica receives no conflicting command: (i) *cmd* will be certified, i.e., it will be voted for by all honest replicas, and (ii) no conflicting command will be certified.

Suppose replica 1 is committing *cmd* at time t. Let us understand the sequence of events from its point-of-view, pictorially represented in the picture below.



Since replica 1 committed at time t, it must have voted at time $t - 2\Delta$. This vote also acts as a *re-proposal*, and hence all honest replicas receive the proposal by time $t - \Delta$. If no honest replica has received a conflicting command at $t - \Delta$, then the honest replica will vote for *cmd*, and *cmd* will be certified. If an honest replica receives a conflicting command *cmd'* after $t - \Delta$, then *cmd'* will not be voted for (a replica only votes for the first valid proposal). If it received *cmd'* before $t - \Delta$, it would indeed vote for *cmd'*. But this vote will arrive at replica 1 before time t, causing replica 1 not to commit. Thus, a 2Δ wait after a vote suffices to commit.