Solana: Une nouvelle architecture pour une blockchain haute performance v0.8.14

Anatoly Yakovenko
anatoly@solana.io

Abstract

Cet article propose une nouvelle architecture de blockchain basée sur la Proof of History (PoH) - une preuve pour vérifier l'ordre et le temps écoulé entre les événements. PoH est utilisé pour encoder le " temps incertain " dans un registre – constituant seulement un complément de données. Lorsqu'il est utilisé avec un algorithme de consensus comme le Proof of Work (PoW) ou Proof of Stack (PoS), PoH peut réduire la surcharge de messages dans une machine " à tolérance d'erreurs " (répliquée Byzantine), permettant un temps de traitement de moins d'une seconde. Cet article propose également deux algorithmes qui exploitent les propriétés de conservation du temps du registre PoH - un algorithme de PoS qui peut récupérer à partir de partitions de toute taille et une Proof of Replication (PoRep). La combinaison de PoRep et PoH assure une protection contre la falsification du grand livre en ce qui concerne le temps (commande) et le stockage. Le protocole est analysé sur un réseau 1 gbps, et cet article

montre que le débit peut atteindre 710k transactions par seconde avec le matériel actuel.

1 Introduction

Blockchain est l'implémentation d'une machine tolérante aux pannes. Les blockchains disponibles publiquement ne dépendent pas du temps, ou font une faible hypothèse sur les capacités du participant à garder le temps [4, 5]. Chaque nœud du réseau s'appuie généralement sur sa propre horloge locale sans connaître les horloges des autres participants du réseau. L'absence d'une source fiable de temps signifie que lorsqu'un horodatage de message est utilisé pour accepter ou rejeter un message, il n'y a aucune garantie que tous les autres participants du réseau feront exactement le même choix. Le PoH présenté ici est conçu pour créer un registre avec un passage vérifiable du temps, c'est-à-dire la durée entre les événements et l'ordre des messages. Il est prévu que chaque nœud du réseau pourra compter sur le passage du temps enregistré dans le registre sans confiance.

2 Contour

Le reste de cet article est organisé comme suit. La conception globale du système est décrite dans la section 3. La description détaillée de la Proof of History est décrite dans la section 4. La description détaillée de l'algorithme de consensus proposé Proof of Stake est décrite dans la section 5. Une description détaillée de la Proof of Replication proposée se trouve dans la section 6. L'architecture du système et les limites de performance sont analysées dans la section 7. Un moteur GPU de smart-contract intuitif haute perfomance est décrit dans la Section 7.5

3 Conception du réseau

Comme représenté sur l'image 1, un nœud de système est désigné à tout moment comme " Leader " pour générer une séquence de " Proof of History ", garantissant la cohérence de lecture globale du réseau et un passage de temps vérifiable. Le leader séquence les messages d'utilisateur et les classe de manière à ce qu'ils puissent être traités efficacement par d'autres nœuds du système, maximisant ainsi le débit. Il exécute les transactions sur l'état

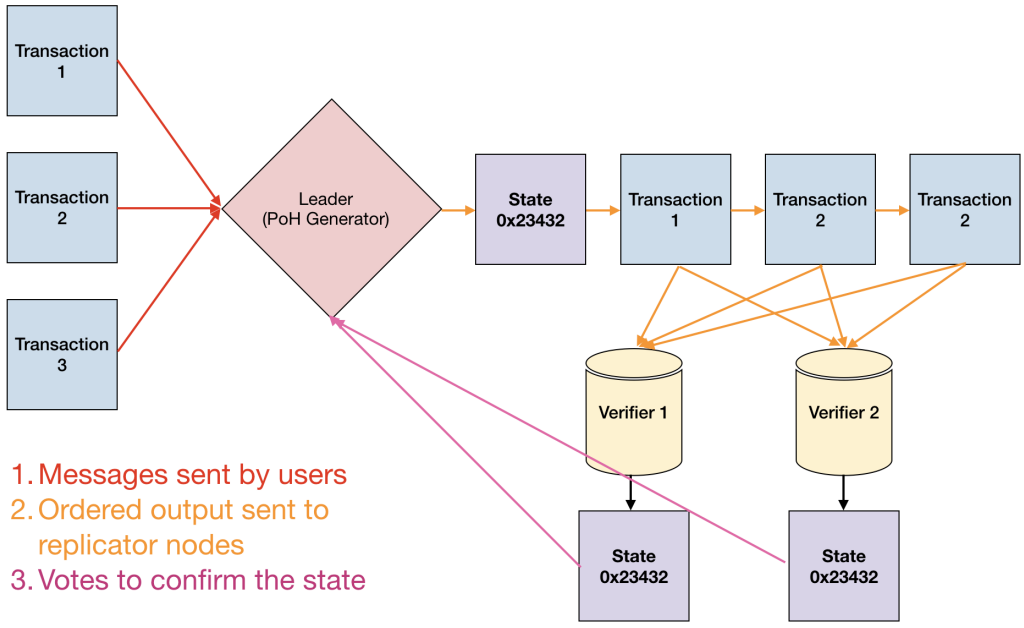


Figure 1: Transaction flow throughout the network.

actuel qui est stocké dans la " RAM " et publie les transactions et une signature de l'état final sur les nœuds de réplication appelés Vérificateurs. Les Vérificateurs exécutent les mêmes transactions sur leurs copies de l'état, et publient leurs signatures enregistrées en tant que confirmations. Les confirmations publiées servent de votes pour l'algorithme de consensus.

Dans un état non classé (partitionné), à un moment donné, il y a un Leader dans le réseau. Chaque noeud vérificateur a les mêmes capacités matérielles qu'un Leader et peut être élu en tant que Leader, ceci se fait via des élections basées sur le PoS. Les élections pour l'algorithme de PoS proposé sont traitées en détail dans la Section 5.6.

En termes de théorème de CAP, la cohérence est presque toujours choisie en fonction de la disponibilité dans un événement d'une partition. Dans le cas d'une grande partition, cet article prévoit un mécanisme pour récupérer le contrôle du réseau à partir d'une partition de n'importe quelle taille. Ceci est expliqué en détail dans la Section 5.12.

4 Proof of History

La " Proof of History " est une séquence de calcul qui peut fournir un moyen de vérifier cryptographiquement le passage du temps entre deux événements. Il utilise une fonction cryptographiquement sécurisée écrite afin que la sortie ne puisse pas être définie par l'entrée, et doit être complètement exécutée pour générer la sortie. La fonction est exécutée dans une séquence sur un seul noyau, sa sortie précédente étant l'entrée actuelle, l'enregistrement périodique de la sortie actuelle et le nombre de fois qu'elle a été appelée. La sortie peut ensuite être recalculée et vérifiée par des ordinateurs externes en parallèle en vérifiant chaque segment de séquence sur un noyau séparé.

Les données peuvent être horodatées dans cette séquence en ajoutant les données (ou un hash de certaines données) dans l'état de la fonction. L'enregistrement de l'état, de l'index et des données tels qu'ils ont été ajoutés dans les séquences fournit un horodatage qui peut garantir que les données ont été créées avant que le hachage suivant ne soit généré dans la séquence. Cette conception prend également en charge la mise à l'échelle horizontale car plusieurs générateurs peuvent se synchroniser entre eux en mélangeant leur état dans les séquences des autres. La mise à l'échelle horizontale est discutée en profondeur dans la Section 4.4

4.1 Description

Le système est conçu pour fonctionner comme suit. Avec une fonction de hash cryptographique, dont la sortie ne peut être prédite sans exécuter la fonction (e.g. `sha256`, `ripemd`, etc.), exécuter la fonction à partir d'une valeur de départ aléatoire et prendre sa sortie comme entrée dans la même fonction à nouveau. Noter le nombre de fois que la fonction a été appelée et la sortie à chaque appel. La valeur aléatoire de départ choisie peut être n'importe quelle chaîne, comme le titre du jour du New York Times.

Par exemple:

PoH Sequence

Index	Operation	Output Hash
1	sha256("any random starting value")	`hash1`
2	sha256(`hash1`)	`hash2`
3	sha256(`hash2`)	`hash3`

Où `hashN` représente le resultat de hash actuel.

Il est seulement nécessaire de publier un sous-ensemble des hashes et des index à un intervalle.

Par exemple:

PoH Sequence

Index	Operation	Output Hash
1	sha256("any random starting value")	`hash1`
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`

Tant que la fonction de hash choisi est résistante aux collisions, cet ensemble de hashes ne pourra être calculé en séquence que par un seul thread. Cela vient du fait qu'il n'est pas possible de prédire quelle sera la valeur du hash à l'index 300 sans lancer l'algorithme depuis le début 300 fois. On peut donc déduire de la structure de données que du temps réel s'est écoulé entre l'index 0 et l'index 300.

Dans l'exemple de la Figure 2, hash `62f51643c1` a été produit au comptage 510144806912 et le hash `c43d862d88` a été produit au comptage 510146904064. Suivant les propriétés précédemment énoncées de l'algorithme PoH, nous pouvons en déduire que le temps réel s'est écoulé entre le compte 510144806912 et le compte 510146904064.

4.2 Horodatage des événements

Cette séquence de hashes peut aussi être utilisée pour enregistrer le fait que certaines données aient été créées avant qu'un certain index hash soit généré. En utilisant une " fonction combinée " pour combiner les données avec le hash actuel à l'index actuel. Les données peuvent simplement être un unique hash d'événements arbitraires. La fonction combinée peut être un simple ajout de données ou n'importe quelle opération qui est résistante aux collisions. Le hash généré par la suite représente un horodatage des données parce que ce hash n'aurait pu être généré seulement après que ces données ne soient insérées.

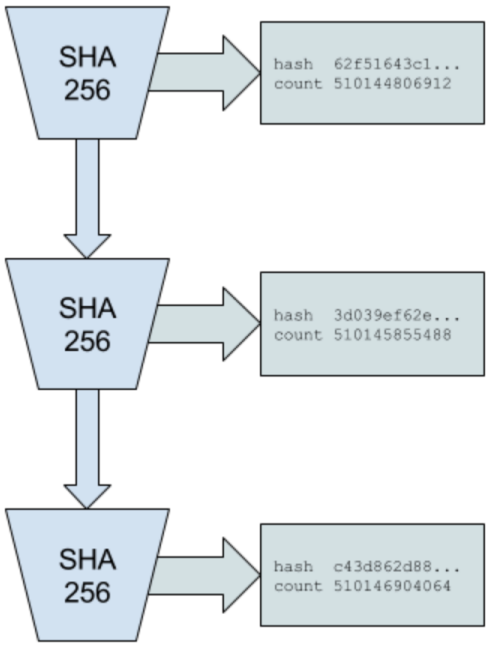


Figure 2: Séquence de Proof of History

Par exemple:

PoH Sequence

Index	Operation	Output Hash
1	sha256("any random starting value")	`hash1`
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`

Si un événement externe survient, comme une photo qui est prise ou la création arbitraire d'une donnée digitale:

POH Sequence

Index	Operation	Output Hash
1	sha256("any random starting value")	`hash1`
200	sha256(hash199)	`hash200`
300	sha256(hash299)	`hash300`
336	sha256(append(hash335, photograph1_sha256))	`hash336`
400	sha256(hash399)	`hash400`
500	sha256(hash499)	`hash500`
600	sha256(append(hash599, photograph2_sha256))	`hash600`
700	sha256(hash699)	`hash700`

Table 1: Séquence PoH avec 2 évènements

PoH Sequence With Data

Index	Operation	Output Hash
1	sha256("any random starting value")	`hash1`
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`
336	sha256(append(`hash335`, photograph_sha256))	`hash336`

`Hash336` est calculé à partir de la donnée binaire ajoutée du `hash335` et du `sha256` de la photo. L'index et le `sha256` de la photo sont enregistrées en tant que partie de la séquence générée. Dès lors, quiconque vérifiant cette séquence peut ensuite recréer ce changement à cette séquence. La vérification peut toujours être effectuée en parallèle et est expliquée dans la Section 4.3

Etant donné que le procédé initial est toujours séquentiel, nous pouvons en déduire que les éléments introduits dans la séquence ont dû se dérouler avant que la prochaine valeur de hash soit calculée.

Dans la séquence reprise dans la Table 1, `photograph2` a été créée avant le `hash600`, et la `photograph1` a été créée avant le `hash336`. Insérer la donnée dans la séquence de hashes produit un changement pour toutes les valeurs dans la séquence. Tant que la fonction de hash utilisée est résistante aux collisions, et que la donnée est ajoutée, il devrait être mathématiquement impossible de pré-calculer une séquence future sur base de connaissance passée relative à quelle donnée sera intégrée dans la séquence.

La donnée qui est mixée à la séquence peut être la donnée brute ou seulement un hash de la donnée accompagnée de métadonnées.

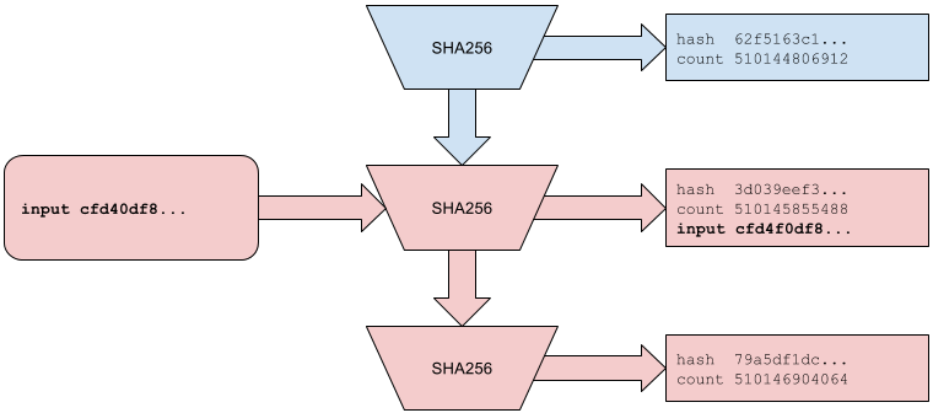


Figure 3: Insertion de données à la Proof of History

Dans l'exemple en Figure 3, l'input `cfd40df8...` a été inséré dans la séquence de Proof of History. Le comptage auquel l'input a été inséré est 510145855488 et l'état auquel il a été inséré est `3d039eef3`. Tous les prochains hashes générés sont impactés par ce changement de séquence. Ce changement est indiqué par le changement de couleur dans la figure.

Chaque node observant la séquence peut déterminer l'ordre dans lequel chaque événement a été inséré et peut déterminer le temps réel entre chaque insertion.

4.3 Vérification

La séquence peut être vérifiée par un ordinateur multi-coeur en moins de temps qu'il ne faut pour la générer.

Par exemple:

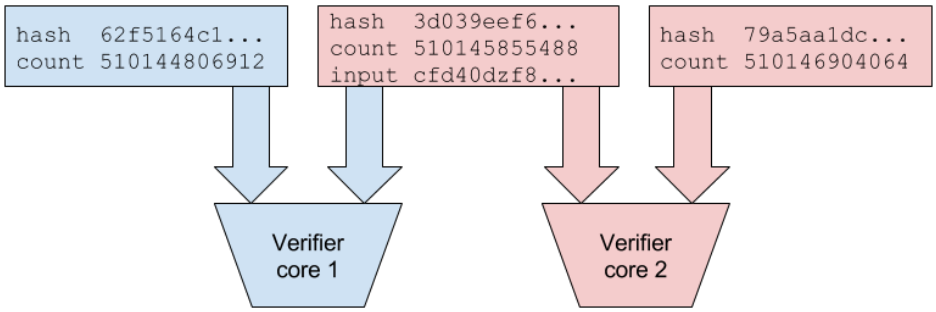


Figure 4: Vérification utilisant plusieurs coeurs

Coeur 1		
Index	Data	Output Hash
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`
Coeur 2		
Index	Data	Output Hash
300	sha256(`hash299`)	`hash300`
400	sha256(`hash399`)	`hash400`

Pour un certain nombre de coeurs donnés, comme un GPU moderne avec 4000 coeurs, le vérificateur peut séparer la séquence de hashes et leurs indexes en 4000 tranches, et en parallèle, assurer que chaque tranche est correcte du premier au dernier hash de la tranche. Si le temps prévu pour produire la séquence est:

$$\frac{\text{Nombre total de Hashes}}{\text{Hashes par seconde pour 1 coeur}}$$

Le temps estimé pour vérifier qu'une séquence soit effectivement correcte est:

$$\frac{\text{Nombre total de Hashes}}{\text{(Hashes par seconde et par coeur * Nombre de coeur disponible à la vérification)}}$$

En exemple, la figure 4, chaque cœur est capable de vérifier individuellement chaque partie de la séquence en parallèle. Puisque toutes les données

en entrée sont enregistrées au niveau de la sortie, avec également le compteur et l'état auquel ils sont attachés, les vérificateurs peuvent répliquer chaque partie en parallèle. Les hashes (en rouge ici plus haut) indiquent que la séquence était modifiée par une insertion de données.

4.4 Mise à l'échelle Horizontale

Il est possible de synchroniser plusieurs " Proof of History " en mélangeant l'état de la séquence de chaque générateur avec un second générateur, et ainsi atteindre la mise à l'échelle horizontale du générateur " Proof of History ". Cette mise à l'échelle est faite sans " sharding ". Le résultat des deux générateurs est nécessaire pour reconstruire l'ordre complet des événements dans le système.

PoH Générateur A		
Index	Hash	Data
1	`hash1a`	
2	`hash2a`	`hash1b`
3	`hash3a`	
4	`hash4a`	

PoH Générateur B		
Index	Hash	Data
1	`hash1b`	
2	`hash2b`	`hash1a`
3	`hash3b`	
4	`hash4b`	

Imaginons les générateurs A et B, A reçoit un paquet de données de B (hash1b), qui contient le dernier état du générateur B, et le dernier état du générateur B observé sur le générateur A. L'état suivant du générateur A dépend alors de l'état de Générateur B, donc nous pouvons déduire que hash1b est arrivé avant hash3a. Cette propriété peut être transitive, donc si trois générateurs sont synchronisés via un seul générateur commun $A \leftrightarrow B \leftrightarrow C$, nous pouvons tracer la dépendance entre A et C même s'ils ne sont pas synchronisés directement.

En synchronisant périodiquement les générateurs, chaque générateur peut alors gérer une partie du trafic externe, ainsi le système global peut gérer un plus grand nombre d'événements à traquer et cela au prix d'une précision en temps réel (dépendent de la latence du réseau entre les générateurs). Un ordre global peut encore être obtenu en choisissant une fonction déterministe qui pourra invoquer tous les événements qui se trouvent dans la fenêtre de synchronisation, comme par exemple la valeur du hash lui-même.

Sur la figure 5, les deux générateurs utilisent la sortie de l'autre et enregistrent l'opération. Le changement de couleur indique que les données de

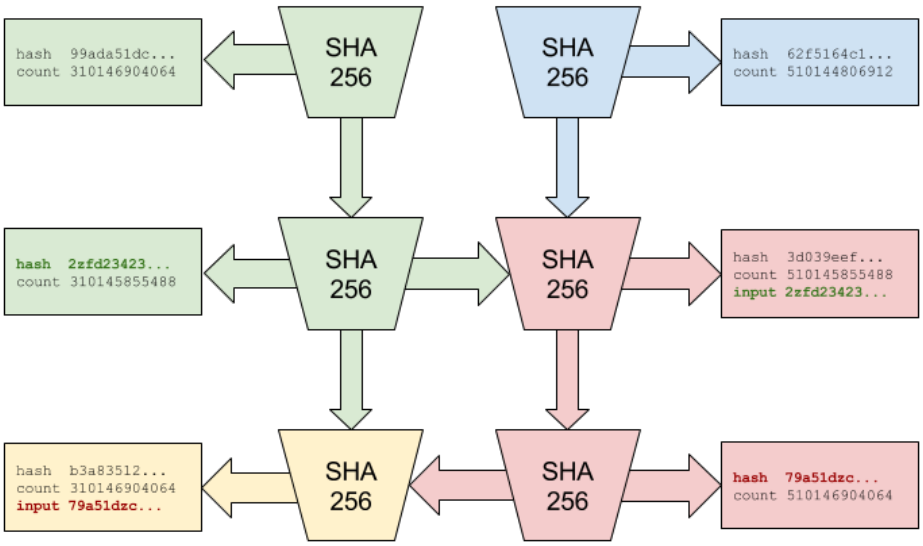


Figure 5: Deux générateurs se synchronisant

l'autre partie ont modifié la séquence. Les hashes générés qui sont mélangés dans chaque flux sont surlignés en gras.

La synchronisation est transitive. $A \leftrightarrow B \leftrightarrow C$ Il existe un ordre d'événements prouvable entre A et C à travers B.

De cette manière, la mise à l'échelle se fait au détriment de la disponibilité. Des connections 10×1 gbps avec une disponibilité de 0.999 auraient une disponibilité de $0.999^{10} = 0.99$.

4.5 Consistance

Les utilisateurs doivent pouvoir renforcer la consistance de la séquence générée et la rendre résistante aux attaques en insérant la dernière sortie observée de la séquence qu'ils considèrent comme valide, dans leur entrée.

PoH Sequence A		
Index	Data	Output Hash
10		`hash10a`
20	Event1	`hash20a`
30	Event2	`hash30a`
40	Event3	`hash40a`

PoH Hidden Sequence B		
Index	Data	Output Hash
10		`hash10b`
20	Event3	`hash20b`
30	Event2	`hash30b`
40	Event1	`hash40b`

Un générateur PoH malveillant peut produire une deuxième séquence cachée avec les événements dans l'ordre inverse et ce, s'il a accès à tous les événements à la fois ou s'il est capable de générer une séquence cachée plus rapide.

Pour éviter cette attaque, chaque événement généré par le client doit contenir le dernier hash observé de ce qu'il considère être une séquence valide. Ainsi, lorsqu'un client crée la donnée "Event1", il doit ajouter le dernier hash observé.

PoH Séquence A

Index	Data	Output Hash
10		`hash10a`
20	Event1 = append(event1 data, `hash10a`)	`hash20a`
30	Event2 = append(event2 data, `hash20a`)	`hash30a`
40	Event3 = append(event3 data, `hash30a`)	`hash40a`

Lorsque la séquence est publiée, Event3 fait référence à hash30a, et si ce n'est pas dans la séquence précédente à cet événement, les consommateurs de la séquence savent qu'il s'agit d'une séquence invalide. L'attaque de " réarrangement partiel " serait alors limitée au nombre de hashes produits durant le laps de temps où le client a observé un événement et ce, depuis son entré. Les clients devraient alors être en mesure d'écrire des softwares qui ne supposent pas que l'ordre soit correct pour la courte période de hashes entre le dernier hash observé et inséré.

Pour empêcher un générateur PoH malveillant de réécrire les hashes des évènements clients, les clients peuvent soumettre une signature des données d'événement ainsi que le dernier hash observé au lieu de simplement soumettre les données.

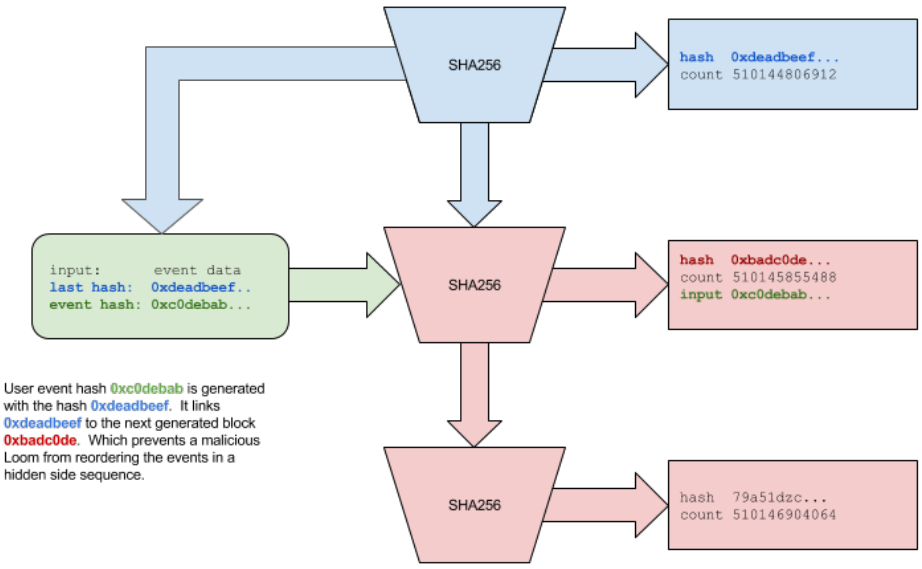


Figure 6: Hash précédent en tant que donnée d'entrée.

PoH Séquence A

Index	Data	Output Hash
10		hash10a
20	Event1 = sign(append(event1 data, hash10a), Client Private Key)	hash20a
30	Event2 = sign(append(event2 data, hash20a), Client Private Key)	hash30a
40	Event3 = sign(append(event3 data, hash30a), Client Private Key)	hash40a

La vérification de ces données nécessite une vérification de signature ainsi qu'une vérification du hash dans la séquence de hachage précédente.
Vérification:

```
(Signature, PublicKey, hash30a, event3 data) = Event3
Verify(Signature, PublicKey, Event3)
Lookup(hash30a, PoHSequence)
```

Dans la figure 6, l'entrée fournie par l'utilisateur dépend du hash `0xdeadbeef...` existant dans la séquence générée avant son insertion. La flèche bleue en haut

à gauche indique que le client fait référence à un hash précédemment produit. Le message du client n'est valide que dans une séquence contenant le hash `0xdeadbeef...`. La couleur rouge de la séquence indique que la séquence a été modifiée par les données du client.

4.6 Overhead

4000 hashes par seconde généreraient 160 kilo-octets de données supplémentaires, et nécessiteraient l'accès à un GPU avec 4000 noeuds et un temps de vérification d'environ 0,25-0,75 millisecondes..

4.7 Attaques

4.7.1 Annulation

La génération d'un ordre inverse nécessiterait qu'un attaquant lance la séquence malveillante après le second événement. Ce délai devrait permettre à tous les nœuds " d'égal à égal " non malveillants de communiquer sur l'ordre original

4.7.2 Vitesse

Avoir plusieurs générateurs peut rendre le déploiement plus résistant aux attaques. Un générateur peut avoir une "bande passante élevée et recevoir de nombreux événements à mélanger dans sa séquence, un autre générateur pouvant être une bande passante à haut débit qui se mélange périodiquement avec le générateur à bande passante élevée.

La séquence à haute vitesse créerait une séquence secondaire de données qu'un attaquant devrait inverser.

4.7.3 Attaques à longue portée

Les attaques à longue portée impliquent l'acquisition d'anciennes clés privées de clients abandonnées et la génération d'un registre falsifié [10]. La preuve historique assure une certaine protection contre les attaques à longue portée. Un utilisateur malveillant qui accède aux anciennes clés privées doit recréer un historique qui demandera autant d'effort de création qu'en a nécessité l'original. Cela nécessiterait l'accès à un processeur plus rapide que celui actuellement utilisé par le réseau, sinon l'attaquant ne rattraperait jamais la longueur de l'historique.

De plus, une seule source de temps permet la construction d'une preuve de réplication plus simple (plus de détails dans la section 6), puisque le réseau est conçu de sorte que tous les participants du réseau s'appuient sur un seul enregistrement historique des événements.

PoRep et PoH ensemble devraient fournir une défense spatiale et temporelle contre la constitution d'un faux registre.

5 Proof of Stake Consensus

5.1 Description

Cette instance spécifique de " Proof of Stake " est conçue pour la confirmation rapide de la séquence actuelle produite par le générateur " Proof of History ", pour le vote et la sélection du générateur de preuve historique suivant, et pour punir les validateurs malveillants. Cet algorithme dépend des messages qui arrivent finalement à tous les nœuds participants dans un certain délai.

5.2 Terminologie

obligations Les obligations sont équivalentes à une dépense en capital dans un système de " Preuve de travail ". Un mineur achète du matériel et de l'électricité et le confie à une seule succursale dans une blockchain " Proof of Work ". Une obligation est une pièce servant de garantie pendant la validation des transactions.

slashing La solution proposée au problème du " rien en jeu " dans les systèmes de preuve de participation [7]. Lorsqu'une preuve de vote pour une autre succursale est publiée, cette succursale peut annuler la caution du validateur. Ceci est une incitation économique conçue pour décourager les validateurs de confirmer plusieurs branches.

super majorité Une super majorité est $\frac{2}{3}$ des validateurs pondérés par leurs liens. Un vote majoritaire indique que le réseau a atteint un consensus, et qu'au moins $\frac{1}{3}$ du réseau aurait dû voter de façon malveillante pour que cette branche soit invalide. Cela mettrait le coût économique d'une attaque à $\frac{1}{3}$ de la capitalisation boursière de la pièce.

5.3 Obligations

Une transaction de liaison implique une quantité de pièces de monnaie spécifiée par un utilisateur qui la transfert vers un compte lié à son identité. Les pièces dans le compte de cautionnement ne peuvent pas être dépensées et doivent rester dans le compte jusqu'à ce que l'utilisateur les supprime. L'utilisateur ne peut supprimer que les pièces périmées qui ont expiré. Les obligations ne sont valables qu'après que la 'super majorité' (grande majorité) des parties prenantes ait confirmé la séquence.

5.4 Vote

Il est prévu que le générateur " Proof of History " puisse publier une signature de l'état à une période prédéfinie. Chaque identité liée doit confirmer cette signature en publiant sa propre signature signée de l'état. Le vote est un simple vote oui, (sans un non). Si la majorité des identités liées a voté dans un délai d'attente déterminé, cette branche sera acceptée comme valable.

5.5 Suppression de vote

Un nombre N de voix manquantes indique que les pièces sont périmées et ne sont plus éligibles pour le vote. L'utilisateur peut émettre une transaction non liée pour les supprimer.

N est une valeur dynamique basée sur le rapport entre les votes périmés et actifs. N augmente à mesure que le nombre de votes périmés augmente. Dans le cas d'une partition réseau importante, cela permet à la branche la plus importante de récupérer plus rapidement que la branche plus petite.

5.6 Elections

L'élection d'un nouveau générateur de PoH se produit lorsque la défaillance du générateur PoH est détectée. Le validateur ayant la plus grande puissance de vote, ou l'adresse de clé publique la plus élevée en cas d'égalité, est choisi comme nouveau générateur de PoH.

Une super majorité de confirmations est requise sur la nouvelle séquence. Si le nouveau leader échoue avant qu'une confirmation de super-majorité ne soit disponible, le validateur le plus élevé suivant est sélectionné, et un nouvel ensemble de confirmations est requis.

Pour changer de vote, un validateur doit voter à un compteur de séquence PoH plus élevé, et le nouveau vote doit contenir les votes qu'il veut changer. Sinon, le deuxième vote pourra être annulé " slashable ". Le changement de vote devrait être conçu de manière à ce qu'il ne puisse avoir lieu qu'à un niveau non majoritaire.

Une fois qu'un générateur de PoH est établi, un générateur suppléant peut être choisi pour prendre en charge les tâches de traitement transactionnel. Si un deuxième existe, il sera considéré comme le prochain leader lors d'une défaillance primaire.

La plate-forme est conçue pour que le suppléant devienne primaire et que les générateurs de rang inférieur soient promus si une anomalie est détectée ou selon un calendrier prédéfini.

5.7 Déclencheurs d'élection

5.7.1 Forked Proof of History generator

Les générateurs de PoH sont conçus avec une identité qui signe la séquence générée. Une " fourchette " ne peut apparaître que si l'identité du générateur PoH a été compromise. Une " fourchette " est détectée car deux enregistrements historiques différents ont été publiés sur la même identité PoH.

5.7.2 Exceptions au Runtime

Un bug, une défaillance ou une erreur intentionelle dans le générateur de PoH pourrait faire en sorte que ce dernier génère un état invalide et publie une signature d'état ne correspondant pas au résultat du validateur local. Les validateurs vont publier la signature correcte *via gossip* et cet évènement déclenchera à son tour un nouveau tour d'éléctions. N'importe quel validateur acceptant un état invalide se verra couper de ses obligations.

5.7.3 Timeouts de Réseaux

Un timeout du réseaux provoquerait une élection.

5.8 Slashing

Le *Slashing* se produit quand un validateur vote deux séquences séparées. Une preuve de vote mal intentionné enlèvera le coin attaché de la circulation

et l'ajoutera à la *mining pool*.

Un vote qui inclus un vote précédant sur une séquence rivale n'est pas éligible en tant que *proof of malicious voting*. Au lieu de couper les liaisons, ce vote supprime le vote en cours sur la séquence de conflit.

Le *Slashing* se produit également si un vote est lancé pour un hash invalide généré par le générateur de PoH. Il est attendu que le générateur génère de manière aléatoire un état invalide, ce qui provoquera un *fallback* au Secondaire.

5.9 Elections Secondaire

Des générateurs de PoH Secondaire et inférieurs peuvent être proposés et acceptés. Une propostion est lancée sur le générateur de séquence primaire. La proposition contient un *timeout*. Si celle-ci est acceptée par une super majorité des votes avant le *timeout* alors le Secondaire est considéré comme élu et vas prendre en charge les fonctions. Le Primaire peut alors faire un trasnfert de fonction au Secondaire en insérant un message dans la séquence générée, indiquant qu'une remise de fonction va se passer. Il peut aussi insérer un état invalide pour forcer le réseau à se replier sur le Secondaire.

Si le Secondaire est élu, il sera alors pris en considération alors que le Primaire se retirera pendant une élection.

5.10 Disponibilité

Les systèmes CAP qui fonctionnent avec des partitions doivent choisir entre cohérence et disponibilité. Notre approche choisi la disponibilité, mais parce que nous avons une mesure du temps subjective, la cohérence est choisie avec un *timeout* humain raisonnable.

Les validateurs de *Proof of Sake* gardent en réserve une certaine somme de token, que l'on appelle *stake*. Cette somme leur permet de voter pour un ensemble de transactions particulier. Le fait de garder en réserve une certaine somme de token est en soit une transaction qui fait partie du fonctionnement du système de PoH. Afin de voter, un validateur PoS doit signer le hash d'un état comme si celui-ci était calculé après avoir traité toutes les transactions d'une position spécifiques dans le registre du PoH. Ce vote est aussi entré comme une transaction dans le flux du système de PoH. En se référant au registre du PoH, nous pouvons donc déduire combien de temps s'est écoulé

entre chaque vote, et si une partition se produit, de déduire pendant combien de temps chaque validateur a été indisponible.

Dans le but de gérer les partitions avec un délai humainemant raisonnable, nous proposons une approche dynamique afin d'*unstake* les validateurs non disponibles. Quand le nombre de validateurs est élevé et au dessus de $\frac{2}{3}$, le processus d'*unstaking* peut être rapide. Le nombre de hash devant être génerés dans le registre est bas tant que le *stake* des valideurs indisponibles n'est pas complètement *unstaked* et qu'ils sont toujours considérés par le consensus. Quand le nombre de validateurs est inférieur à $\frac{2}{3}$ mais au dessus de $\frac{1}{2}$, le processus d' *unstaking* est plus lent, demandant un nombre de hash génerés plus important avant que les validateurs indisponibles soient *unstaked*. Dans une partition large, comme une partition dont il manque $\frac{1}{2}$ ou plus de validateurs, le processus de *unstaking* est très lent. Les transactions peuvent quand même entrer dans le flux, et les validateurs peuvent encore voter, mais les $\frac{2}{3}$ du consensus ne seront pas atteints tant qu'un très grand nombre de hash ne soient génerés et que les validateurs manquants n'aient été *unstaked*. Le délai pour un réseau avant d'être de nouveau opérationnel nous permet en tant que client du réseau de choisir la partition que nous voulons continuer à utiliser.

5.11 Récupération

Dans le système que nous proposons, le registre peut être récupéré après n'importe quelle défaillance. Cela signifie que n'importe qui dans le monde peut chosir un endroit quelconque dans le registre et créer un *fork* valide en ajoutant à la suite de nouveaux hash et transations génerés. Si tous les validateurs sont absents dans ce *fork*, alors cela prendrait un très long moment pour que n'importe quel lien additionel devienne valide, et il serait compliqué pour cette branche de parvenir au consensus d'une super majorité de $\frac{2}{3}$. De ce fait, un rétablissement complet sans validateur disponible exigera un très grand nombre de hash ajoutés au registre, et c'est seulement après l'*unstake* de l'ensemble des valideurs indisponibles que de nouveaux liens pourront être ajoutés au registre.

5.12 Finalité

La PoH permet aux validateurs du réseau d'observer l'historique de ce qu'il s'est passé avec une certaine certitude du moment ou ces évènemments se

sont passés. Au fur et à mesure que le générateur de PoH produit un flux de messages, tous les validateurs sont tenus de soumettre leurs signatures d'état en un délai maximum de 500ms. En fonction de l'état du réseau ce délai peut être plus court. Du fait que chaque vérification est saisie dans le flux d'échange, n'importe qui dans le réseau peut valider que tous les validateurs ont soumis leur vote dans un délai de 500ms sans observer leurs votes de manière directe.

5.13 Attaques

5.13.1 Tragédie des communs

Les validateurs de PoS confirment simplement l'état généré par le générateur de la PoH. Il y'a un intérêt économique pour les valideurs d'approuver chaque état de hash généré sans réellement effectuer de vérification. Pour éviter cette situation, le générateur de la PoH injectera un hash invalide à interval de temps aléatoire. Chaque validateurs ayant voté pour ce hash sera *slashed*. Quand le hash invalide est généré, le réseau devra immédiatement promouvoir le second générateur PoH élu.

Il est nécessaire que chaque validateur réponde dans un délai court - 500ms par exemple. Le *timeout* doit être suffisement bas afin de réduire les chances qu'un validateur malicieux ait le temps d'observer et copier le vote d'un autre validateur.

5.13.2 Collusion avec le générateur de la PoH

Un validateur en collusion avec le générateur de la PoH saurait en avance quand le hash invalide sera produit et ne votera donc pas pour lui. Ce scénario n'est pas différent de celui ou le générateur de la PoH possède un *stake* important. Le générateur PoH doit tout de même effectuer tout le travail afin de produire le hash d'état.

5.13.3 Censure

De la censure ou du déni de service pourraient avoir lieu si $\frac{1}{3}$ des validateurs refusent de valider toute séquence possédant de nouvelles obligations. Le protocole peut empêcher ce genre d'attaque en ajustant dynamiquement la fréquence à laquelle les obligations deviennent obsolètes. Dans le cas d'un déni de service, la plus grande des partitions sera conçue pour bifurquer

et censurer les obligataires Byzantins. Le plus grand réseau se rétablira à mesure que les obligations Byzantines deviennent obsolètes avec le temps. La partition Byzantine la plus petite ne sera pas en mesure d'évoluer pendant une plus longue période.

L'algorithme fonctionnerait comme suit. Une majorité du réseau élirait un nouveau Leader. Le Leader empêcherait ensuite les obligataires Byzantins de participer. Le générateur de la PoH devra continuer la génération de séquence afin de prouver l'écoulement du temps, et cela, jusqu'à ce que suffisamment d'obligations Byzantines soient devenues obsolètes et ainsi permettre au plus grand réseau d'avoir une majorité qualifiée. La fréquence à laquelle les obligations deviennent obsolètes serait basée dynamiquement sur le pourcentage des obligations actives. La branche Byzantine minoritaire du réseau aurait alors besoin d'attendre beaucoup plus longtemps que la branche majoritaire pour récupérer une majorité qualifiée. Une fois qu'une majorité qualifiée a été établie, une compression pourrait être utilisée pour définitivement punir les obligataires Byzantins.

5.13.4 Attaques de longue portée

La PoH fournit une défense naturelle contre les attaques de longue portée. Rétablir le registre à partir de n'importe quel moment dans le passé demanderait à l'attaquant rattraper le registre valide en étant plus rapide que le générateur de la PoH.

Le protocole de consensus fournit une deuxième ligne de défense car toute attaque prendrait plus de temps qu'il n'en faut pour *unstake* l'ensemble des validateurs légitimes. Cela crée un éventuel "écart" dans l'historique du registre. En comparant deux registres de même hauteur, celui avec la plus petite des partitions maximales peut objectivement être considéré comme valide.

5.13.5 Attaques ASIC

Deux opportunités d'attaques ASIC existent dans le protocole – pendant le partitionnement et en trompant le délai d'expiration de l'état de "Finalité".

Pour les attaques ASIC pendant les partitionnements, la vitesse à laquelle les obligations sont *unstake* est non-linéaire, et pour des réseaux avec de grandes partitions cette vitesse devient trop faible pour permettre la réalisation d'une attaque ASIC rentable.

Pour les attaques ASIC pendant l'état de "Finalité", la vulnérabilité permet aux validateurs Byzantins qui ont une *stake* attitrée d'attendre les confirmations d'autres nœuds pour injecter leurs votes en collaboration avec un générateur de PoH. Le générateur de PoH peut alors utiliser son ASIC le plus rapide pour générer des hashes qui nécessiteraient 500ms en moins de temps, et autoriser la communication entre le générateur de PoH et les nœuds collaborateurs. Cependant, si le générateur de PoH est aussi Byzantin, il n'y a pas de raison qu'il n'ait pas communiqué le compteur réel au moment ou une défaillance aurait pu être insérée. Ce scénario n'est pas différent de celui ou un générateur de PoH et tous les collaborateurs partagent la même identité comparé au fait de n'avoir qu'une seule entité de *stake* combinée en n'utilisant qu'un seul ensemble de matériel.

6 Preuve de Réplication diffusées

6.1 Description

Filecoin a proposé une version de Preuve de Réplication [6]. Le but de cette version est d'avoir des vérifications de Preuve de Réplication rapides et en continues, ce qui est rendu possible grace au système de PoH qui permet de dater les des séquences générées. La réplication n'est pas utilisée comme un algorithme de consensus, cependant c'est un outil utile permettant de garantir, contre le coût de stockage de l'historique ou de l'état de la blockchain, une grande disponibilité.

6.2 Algorithme

Comme le montre la figure 7 le chiffrement CBC chiffre chaque bloc de donnée en séquence. Le bloc précédemment chiffré est utilisé afin d'appliquer un XOR à la nouvelle données d'entrée.

Chaque réplication d'identité génère une clé en signant un *hash* qui a été générée par une séquence de PoH. Cela lie la clé à l'identité d'un réplicateur et à une séquence de PoH spécifique. Seuls des *hashs* spécifiques peuvent être sélectionnés. (Voir la section 6.5 dans "Sélection de hash")

Bloc par bloc, l'ensemble de données est totalement chiffré. Puis afin de générer une preuve, un générateur de nombre pseudo-aléatoire permettant de sélectionner 32 octets de chaque bloc est généré à partir de la clé.

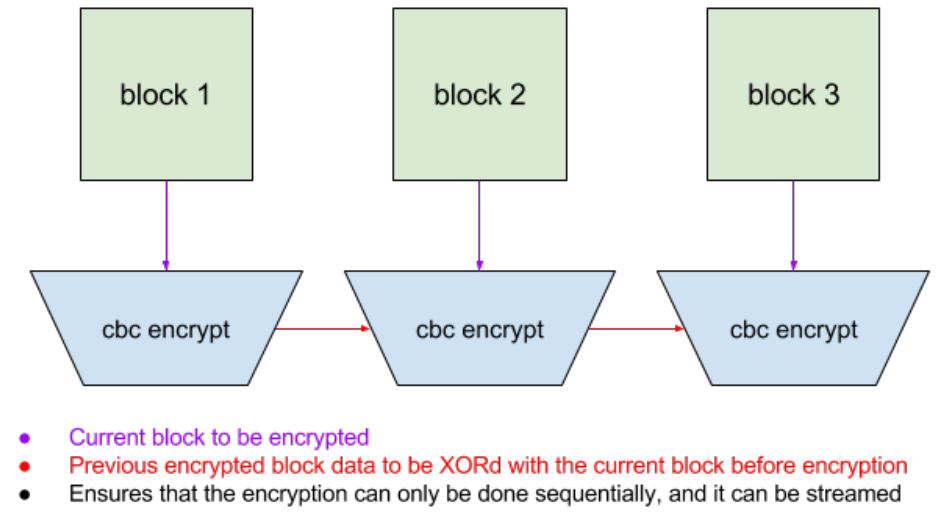


Figure 7: Cryptage CBC séquentiel

Un arbre de Merkle est calculé avec le hash de la PoH sélectionné préfixé à chaque tranche.

La racine, la clé ainsi que le hash généré sont publiés. Le nœud de réplication est exploité pour la publication de nouvelles preuves dans N hashes au fur et à mesures qu'ils sont produits par le générateur de PoH, ou N vaut approximativement $\frac{1}{2}$ du temps nécessaire au chiffrement des données. Le générateur de la PoH publiera des *hashes* spécifiques pour Preuve de Réplication à des périodes prédéfinies. Le nœud de réplication doit sélectionner le prochain *hash* publié pour générer la preuve. A nouveau, le *hash* est signé, et des tranches aléatoires sont sélectionnées des blocs pour créer la racine de l'arbre de Merkle.

Après une période de N preuves, les données sont à nouveau chiffrées avec une nouvelle clé CBC.

6.3 Vérification

Avec un nombre N de cœurs, chaque cœur peut effecture un chiffrement en continu pour chaque identité. L'espace total requis est $2blocs \times Ncoeurs$, car le bloc précédemment chiffré est nécessaire pour générer le suivant. Chaque cœur peut ensuite être utilisé pour générer toutes les preuves qui ont dérivé

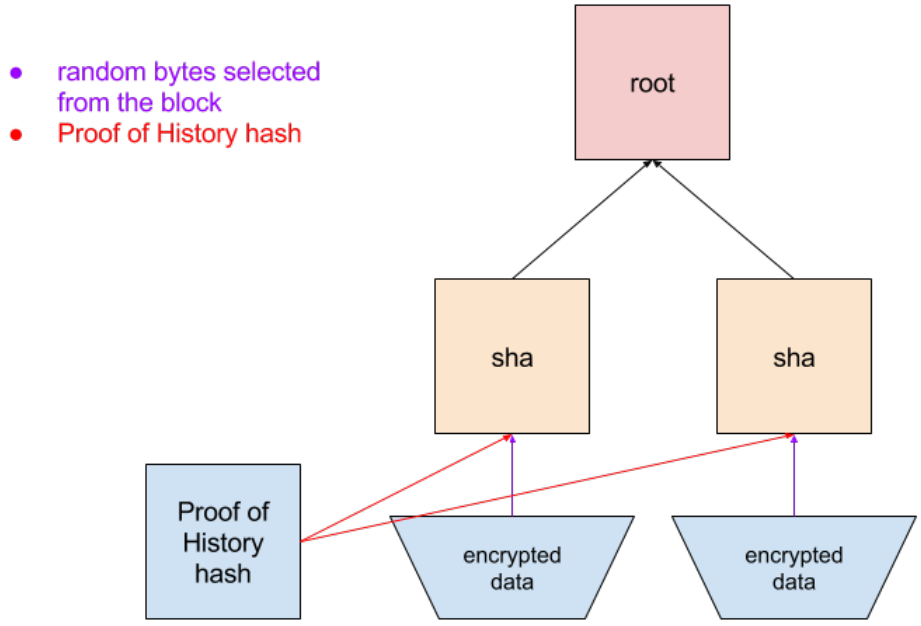


Figure 8: Preuve de Réplication rapide

du bloc chiffré actuel.

Il est attendu que le temps total nécessaire à la vérification des preuves soit égal au temps qu'il a fallu pour les chiffrer. Les preuves elles-mêmes ne consomment que quelques octets aléatoires sur un bloc, la quantité de données nécessaire pour le *hash* est alors significativement plus faible que la taille du bloc chiffré. Le nombre d'identités de réplication pouvant être vérifiées en même temps est égal au nombre de cœurs disponibles. Les GPUs modernes ont 3500+ cœurs disponibles, même s'ils ne fonctionnent qu'à $\frac{1}{2}$ ou $\frac{1}{3}$ de la fréquence d'horloge des CPUs.

6.4 Rotation de Clé

Sans la rotation de clé, la même réplication chifrée peut générer des preuves peu robustes pour de multiples séquences de PoH. Les clés changent de manière périodique et chaque réplication et à nouveau chiffrée avec une nouvelle clé liée à une séquence de PoH spécifique.

Les périodes de rotation des clés doivent être suffisamment longues pour

permettre la vérification des réplications de preuves sur des GPUs, ce qui est plus lent que sur les CPUs.

6.5 Sélection de hash

Le générateur de PoH publie un *hash* visant à être utilisé par l'ensemble du réseau afin de chiffrer les Preuves de Réplication. Ce *hash* et aussi utilisé comme le nombre généré pseudo-aléatoirement pour la sélection d'octets pour les preuves rapides.

Le *hash* est publié à un intervalle de temps régulier qui sont approximativement égaux à $\frac{1}{2}$ du temps nécessaire au chiffrage de l'ensemble des données. Chaque identité de réplication doit utiliser le même *hash* et son résultat signé afin de définir une sélection d'octets ou l'utiliser directement comme clé de chiffrement.

La période à laquelle chaque réplicateur doit fournir une preuve doit être plus petite que le temps de chiffrement. Sinon, le réplicateur est en capacité de diffuser le chiffrement et le supprimer pour chaque preuve.

Un générateur malveillant pourrait injecter des données dans la séquence antérieure à ce hash dans le but de générer un hash spécifique. Cette attaque est plus longuement discutée au sous chapitre 5.13.2.

6.6 Validation de preuve

Il n'est pas attendu du nœud de " proof of history " de valider les preuves de réplication soumises. Il est prévu de garder une trace du nombre de preuves soumises, en attentes et vérifiés, par les réplicateurs. Il est attendu d'une preuve qu'elle soit vérifiée quand le réplicateur est capable de la faire signer par une grande majorité des validateurs du réseau.

Les vérifications sont collectées par le réplicateur via un réseau p2p *gossip* et sont toutes soumises dans un paquet qui contient une super majorité des validateurs du réseau. Ce paquet vérifie l'ensemble des preuves soumises avant un *hash* spécifique généré par la séquence de "Proof of History" et peut contenir plusieurs identités de réplicateurs à la fois.

6.7 Attaques

6.7.1 Spam

Un utilisateur mal intentionné pourrait créer de nombreuses identités de réplicateurs et spammer le réseaux avec des preuves falsifiées. Afin de faciliter une vérification plus rapide, des nœuds sont requis pour fournir les données chiffrées et l'intégralité de l'arbre de Merkle au reste du réseau lorsqu'il demande une vérification.

La preuve de réplication conçue dans ce document permet des vérifications de preuves supplémentaire à bas coût car elles ne nécessitent pas d'espace supplémentaire. Cependant chaque identité consommerait 1 cœur du noyau de chiffrage. La cible de réplication devrait être réglée à une taille maximale de coeurs facilement disponible. Les puces GPU modernes possèdent plus de 3500 coœurs.

6.7.2 Effacement partiel

Un node de réplication pourrait tenter d'effacer partiellement certaines données pour éviter de stocker l'intégralité de l'état. Le nombre de preuves et le caractère aléatoire des *seed* devraient rendre cette attaque difficile.

Par exemple, un utilisateur stockant 1 téraoctet de données efface un seul octet de chaque bloc de 1 mégaoctet. Une seule vérification qui échantillonne 1 octet sur chaque mégaoctet aurait une probabilité de collision un octet effacé de $1 - (1 - 1/1,000,0000)^{1,000,000} = 0.63$. Après 5 vérifications, la probabilité est de 0.99.

6.7.3 Collusion avec le générateur de PoH

Le *hash* signé devrait servir de base à la constitution de l'échantillon. Si un réplicateur avait la possibilité de sélectionner un hash spécifique à l'avance alors il pourrait effacer tous octets qui ne seront pas échantillonnés.

Une identité de réplicateur qui conspire avec le générateur de PoH pourrait injecter une transaction spécifique en fin de séquence avant que le *hash* prédéfinit pour la sélection d'octets aléatoires ne soit généré. Avec suffisamment de cœurs, un attaquant pourrait générer un hash qui sera choisi à la place de l'identité des réplicateurs.

Cette attaque ne peut bénéficier qu'à une seule identité de réplicateur. Du fait que toutes les identités doivent utiliser exactement le même hash

signé par chiffrement ECDSA (ou équivalent), la signature résultante est unique pour chaque identité du réplicateur et résistante à la conspiration. Une identité de réplicateur unique n'aurait que de faible gain.

6.7.4 Déni de service

Le coût d'ajout d'une identité de réplicateur supplémentaire devrait être égal au coût du stockage. Le coût de l'ajout d'une capacité de calcul supplémentaire pour vérifier toutes les identités des réplicateurs est censées être égal au coût d'un cœur de CPU ou GPU par identité de réplication.

Cela crée une vulnérabilité par déni de service sur le réseau exploitable en créant un grand nombre d'identités de réplicateurs valides.

Pour limiter cette attaque, le protocole de consensus choisi pour le réseau peut sélectionner une cible de réplication et lui attribuer les preuves de réplication répondant à des caractéristiques souhaitées, telles que la disponibilité sur le réseau, la bande passante, la géolocalisation, etc...

6.7.5 Tragédie des communs

Les vérificateurs PoS pourraient simplement confirmer la PoRep sans réellement effectuer de vérification. L'intérêt économique devrait être au même niveau qu'avec les vérificateurs du PoS pour la réalisation des vérifications en divisant la récompense de minage entre les vérificateurs du PoS et le nœud de réplication des PoRep.

Pour éviter ce scénario, les vérificateurs PoRep peuvent parfois soumettre de fausses preuves. Ils peuvent prouver que la preuve est fausse en fournissant la fonction qui a généré les fausses données. Tout vérificateur de PoS qui a confirmé une fausse preuve serait détruit.

7 Architecture système

7.1 Composants

7.1.1 Leader, générateur de PoH

Le Leader est un générateur de " proof of history " élu. Il consomme de manière arbitraire des transactions d'utilisateur et affiche une séquence de " proof of history " de toutes les transactions garantissant un ordre global

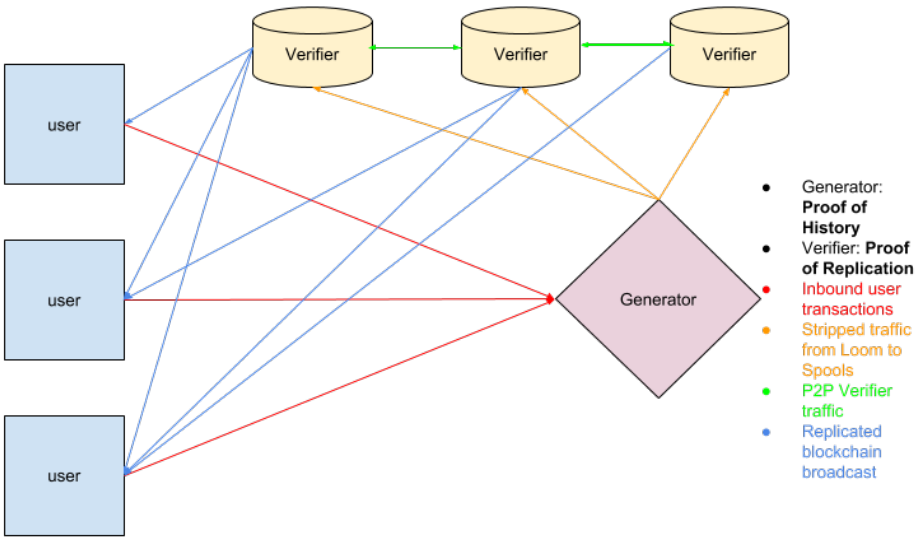


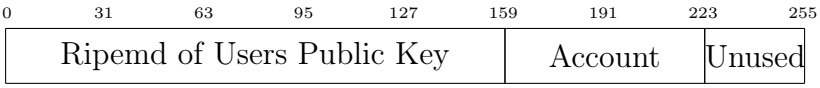
Figure 9: System Architecture

unique dans le système. Après chaque lot de transactions, le Leader émet une signature de l'état qui résulte de l'exécution des transactions dans cet ordre. Cette signature est signée avec l'identité du Leader.

7.1.2 L'état

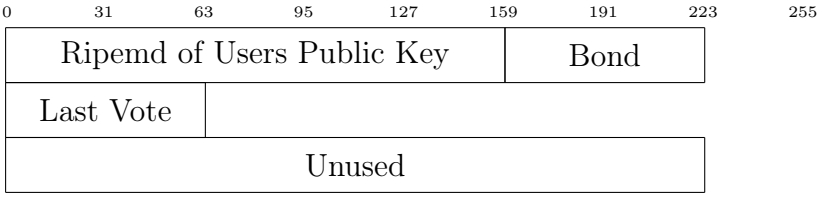
L'état est une simple table de hash indexée par l'adresse des utilisateurs. Chaque cellule contient l'adresse complète de l'utilisateur et la mémoire requise pour ce calcul. Par exemple,

La table de transaction contient :



Pour un total de 32 octets.

La table des obligations de " Proof of Stake " contient:



Pour un total de 64 octets.

7.1.3 Vérificateurs, State Replication

Les nodes vérificateurs répliquent l'état de la blockchain et fournissent une haute disponibilité son état. La cible de réplication est sélectionnée par l'algorithme de consensus et les validateurs de l'algorithme de consensus sélectionnent et votent pour les nœuds de " proof of replication " qu'ils approuvent en fonction de critères définis hors chaîne.

Le réseau peut être configuré avec une taille de liaison " Proof of Stake " minimale et des exigences spécifiques par liaison d'identité de réplicateur.

7.1.4 Validateurs

Ces nœuds consomment de la bande passante des vérificateurs. Ce sont des nœuds virtuels, qui peuvent fonctionner sur les mêmes machines que celles des vérificateurs, du Leader, ou sur des machines distinctes spécifiques pour l'algorithme de consensus configuré pour ce réseau.

7.2 Limites du réseau

Le Leader devrait être en mesure de récupérer les paquets provenants d'utilisateurs, de les ordonner de la manière la plus efficace possible, et de les séquencer dans une séquence de " Proof of History " publiée aux vérificateurs en aval. L'efficacité est basée sur les modèles d'accès à la mémoire des transactions, de ce fait les transactions sont ordonnées pour minimiser les erreurs et maximiser la pré-extraction.

Format des paquets entrants :

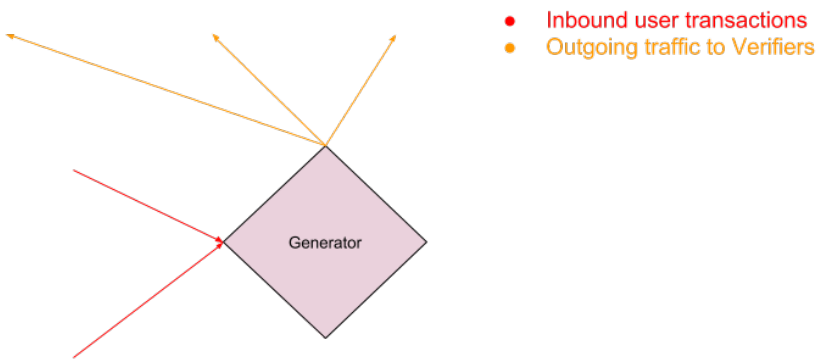
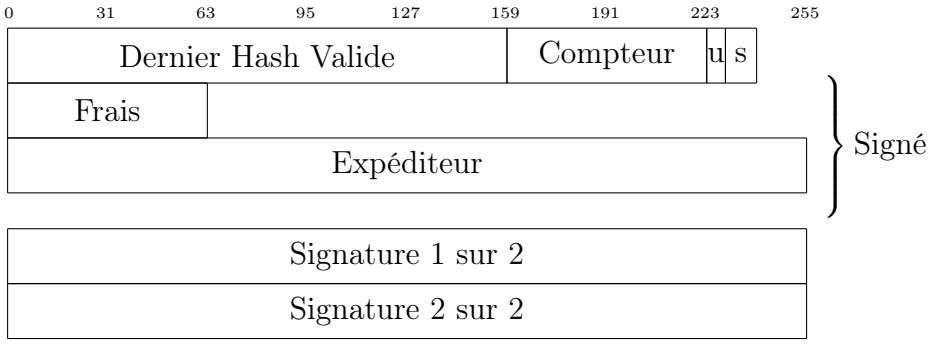


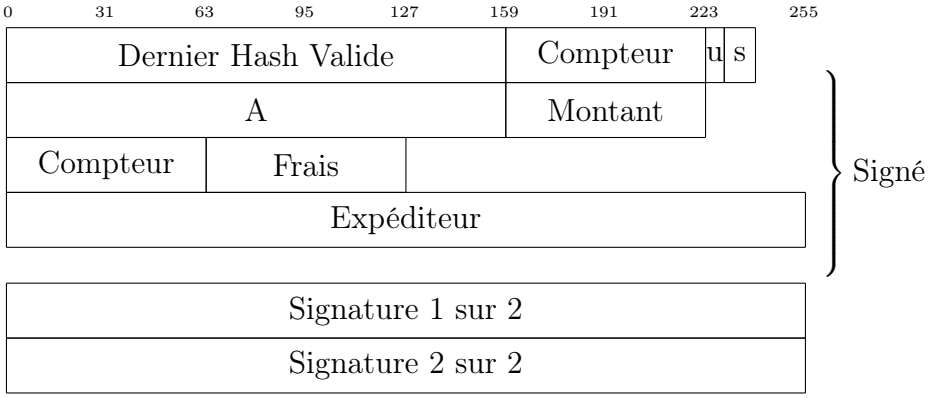
Figure 10: Generator network limits



Taille $20 + 8 + 16 + 8 + 32 + 32 + 32 = 148$ octets.

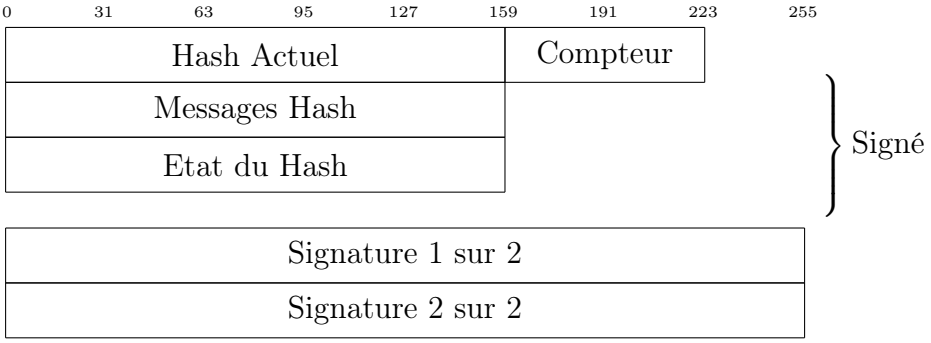
La capacité minimale pouvant être prise en charge serait 1 compte de destination d'une taille minimale de 176 octets.

Avec une capacité:



Avec une capacité minimale: 176 octets

Le paquet de séquence " Proof of History " contient le *hash* actuel, le compteur, le *hash* de tous les nouveaux messages ajoutés à la séquence de PoH et l'état de signature après le traitement de tous les messages. Ce paquet est envoyé une fois tous les N messages diffusés.
Paquet de " Proof of History ":



Taille minimale du paquet de sortie est de : 132 octets

Pour une connexion réseau de 1 gigaoctet par seconde, le nombre maximum de transactions possibles est de 1 gigaoctet par seconde / 176 octets = 710k tps max. Une perte de $(1 - 4\%)$ est prévue en raison de l'encapsulation Ethernet. La capacité inutilisée sur le montant cible pour le réseau peut être utilisée pour augmenter la disponibilité en appliquant le code Reed-Solomon

à la sortie, et en la répartissant aux vérificateurs disponibles en aval.

7.3 Limites de calcul

Chaque transaction nécessite une vérification sommaire. Cette opération n'utilise pas de mémoire en dehors du message de transaction lui-même et peut être parallélisée indépendamment. Le débit devrait donc être limité par le nombre de cœurs disponibles dans le système.

Les GPU basés sur les serveurs de vérification ECDSA ont eu des résultats expérimentaux de 900k opérations par seconde [9].

7.4 Limites de mémoire

Une implémentation naïve de l'état comme une table de hachage complète de 50 pour cent avec des entrées de 32 octets pour chaque compte, pourrait théoriquement contenir 10 milliards de comptes dans 640GB. L'accès régulier aléatoire à cette table est mesuré à $1.1 * 10^7$ écrit ou lit par seconde. Basé sur 2 lectures et deux écritures par transaction, le débit de la mémoire peut gérer 2.75 m de transactions par seconde. Cela a été mesuré sur un Amazon Web Services 1TB x1.16xgrande instance.

7.5 Smart Contracts Haute Performance

Les smarts contracts sont une forme généralisée de transactions. Ce sont des programmes qui s'exécutent sur chaque noeud et en modifient l'état. Cette conception s'appuie sur le Berkeley Packet Filter bytecode, qui est rapide et facile à analyser, et le JIT bytecode comme langage de smart contract.

L'un de ses principaux avantages est le coût nul du Foreign Function Interface. Intrinsèques, ou fonctions qui sont implémentées directement sur la plateforme, sont appelables par les programmes. L'appel des intrinsèques suspend ce programme et planifie l'intrinsèque sur un serveur de haute performance. Les intrinqèques sont regroupées ensemble pour s'exécuter en parallèle sur le GPU.

Dans l'exemple ci-dessus, deux programmes utilisateur différents appellent la même valeur intrinsèque. Chaque programme est suspendu jusqu'à ce que l'exécution par lots de l'intrinsèque soit terminée. Un exemple intrinsèque est la vérification ECDSA. Le traitement de ces appels pour exécuter sur le GPU peut augmenter le débit par des milliers de fois.

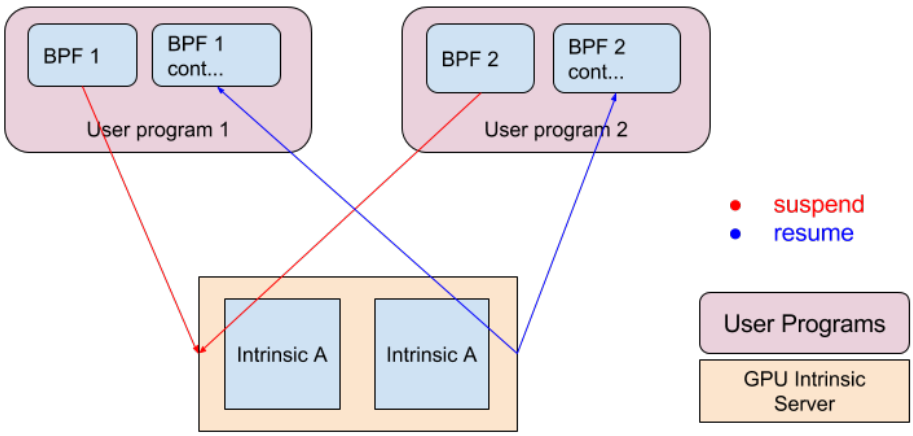


Figure 11: Executing BPF programs.

Ce trampoline ne nécessite pas d'OS natif supportant le threads context switching, puisque le BPF bytecode a un contexte bien défini pour toute la mémoire qu'il utilise.

eBPF backend a été inclus dans LLVM depuis 2015, de sorte que tout langage LLVM frontend peut être utilisé pour écrire des smart contracts. Il est dans le noyau Linux depuis 2015, et les premières itérations du bytecode existent depuis 1992. Un seul passage peut vérifier l'eBPF pour l'exactitude, établir ses exigences d'exécution et de mémoire et le convertir en instructions x86.

References

[1] Liskov, Practical use of Clocks
`http://www.dainf.cefetpr.br/ tacla/SDII/PracticalUseOfClocks.pdf`

[2] Google Spanner TrueTime consistency
`https://cloud.google.com/spanner/docs/true-time-external-consistency`

[3] Solving Agreement with Ordering Oracles
`http://www.inf.usi.ch/faculty/pedone/Paper/2002/2002EDCCb.pdf`

[4] Tendermint: Consensus without Mining
`https://tendermint.com/static/docs/tendermint.pdf`

[5] Hedera: A Governing Council & Public Hashgraph Network
https://s3.amazonaws.com/hedera-hashgraph/hh-whitepaper-v1.0-180313.pdf

[6] Filecoin, proof of replication,
https://filecoin.io/proof-of-replication.pdf

[7] Slasher, A punative Proof of Stake algorithm
https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorith

[8] BitShares Delegated Proof of Stake
https://github.com/BitShares/bitshares/wiki/Delegated-Proof-of-Stake

[9] An Efficient Elliptic Curve Cryptography Signature Server With GPU Acceleration
http://ieeexplore.ieee.org/document/7555336/

[10] Casper the Friendly Finality Gadget
https://arxiv.org/pdf/1710.09437.pdf