Solana:
高パフォーマンスなブロックチェーンを実現する新しいアーキテクチャ v0.8.13

Anatoly Yakovenko
anatoly@solana.io

免責事項 当資料はトークンの販売を意図した文書ではなく、Solana の仕組みに対する読者からのフィードバックやコメントを集めることを目的としたものである。Solana がトークンセール（SAFT 形式等）を開催する場合には、リスク要因など開示すべき情報をまとめた文書を作成する。その際は併せて本文の最新版が開示される予定だが、大幅な改訂がなされる可能性がある。また米国内に向けて販売がおこなわれる場合は、認定投資家のみを勧誘対象とする予定である。

本文は Solana の事業やトークンの発展性、利用価値、金銭的価値を確約したものではない。このホワイトペーパーに記載される内容は現時点における予定を記したもので、それはプロジェクトの方針により変更されうる。またデータ、暗号通貨の市況やその他の要因、つまり Solana プロジェクトが持つ影響力の範疇を超えたことに起因して、この目論見の成否に影響を与えることも予想される。本文中の未来事象に関する記述は Solana プロジェクトの分析に基づいたものであり、その見込みは確約されたものではない。

概要

本文では Proof of History（PoH）による新たなブロックチェーンのアーキテクチャを提言する。PoH はトラストレスなネットワークにあって、一連のイベント発生タイミングの前後関係を記帳し、それを証明する仕組みである——この台帳の特徴は、情報を追記することは可能だが、記帳済みの情報の書き換えは不可能である。PoH は Proof of Work（PoW）あるいは Proof of Stake（PoS）などの合意形成アルゴリズムと併用することで、Byzantine Fault Tolerant によるステートマシーンの状態共有のオーバーヘッドを低減し、結果的に合意形成にかかる時間を短縮することができる。また PoH による時間管理の性能をより強固にする二つのアルゴリズムを紹介する。ひとつは任意のサイズに分断されたネットワークから復旧可能な PoS アルゴリズム、もうひとつは Proof of Replication（PoRep）を効率的にストリーミングするアルゴリズムである。PoRep と PoH の組み合わせは、時間（順序）とデータ保管を司る台帳の偽造に対する防衛手段となる。現代のハードウェア性能と 1Gbps のネットワーク環境が備えられていることを前提として、このプロトコルによる実装は最大 710,000TPS（transaction per-second）のスループットが実現可能であることを示す。

1　はじめに

ブロックチェーンは耐障害性を備える複製されたステートマシンの実装である。現在パブリックに利用可能なブロックチェーンの生成アルゴリズムは時間に依存しない、あるいはネットワーク参加者が正確な時間を管理できていることに期待しない [4, 5]。ネットワーク上の各ノードはローカル時間のみを参照し、他ノードのローカル時間をまったく意識しない。『信頼できる時間の基準がない』ということは、タイムスタンプをメッセージの承諾・拒否の判断に用いる場合に、すべてのノードがそのメッセージに対する判断を同じように下すとは限らないということになる。ここで紹介する PoH は、ネットワーク上の時間の基準となりえる台帳を提供する設計になっている。ここでいう時間とは、二つのイベントに挟まれた期間やメッセージの順序関係を指す。ネットワーク上のすべてのノードは、そのネットワークがトラストレスであるにも関わらず、PoH が生成した時間の基準に依拠できるようになると期待される。

2　本文の構成

本資料は次のような構成が取られている。セクション 3 ではシステム設計の全体像が描かれている。セクション 4 では Proof of History の詳細が語られている。セクション 5 では今回提案される Proof of Stake による合意形成アルゴリズムが示されている。セクション 6 では高速な Proof of Replication の仕組みが詳細に説明されている。セクション 7 ではシステムアーキテクチャとパフォーマンス上限に関する分析がなされている。セクション 7.5 では GPU 計算向きの高パフォーマンスなスマートコントラクト・エンジンについて詳しい考察がなされている。

3　ネットワークデザイン

図 1 に示されているように、システムノードは常に Leader として Proof of History の生成を担う。これはネットワーク全体で一貫性のある時間軸を提供する。スループットを最大化するため、他ノードが効率よく処理を進められるよう、Leader はユーザメッセージを並べて順序付けする。これは RAM（主記憶装置）に格納された状態に基づいてトランザクションを執行し、実行結果を署名付きで Verifier と呼ばれるノードに引き渡す。Verifier は各自が RAM 上に保持している状態から同じトランザクションを繰り返し、実

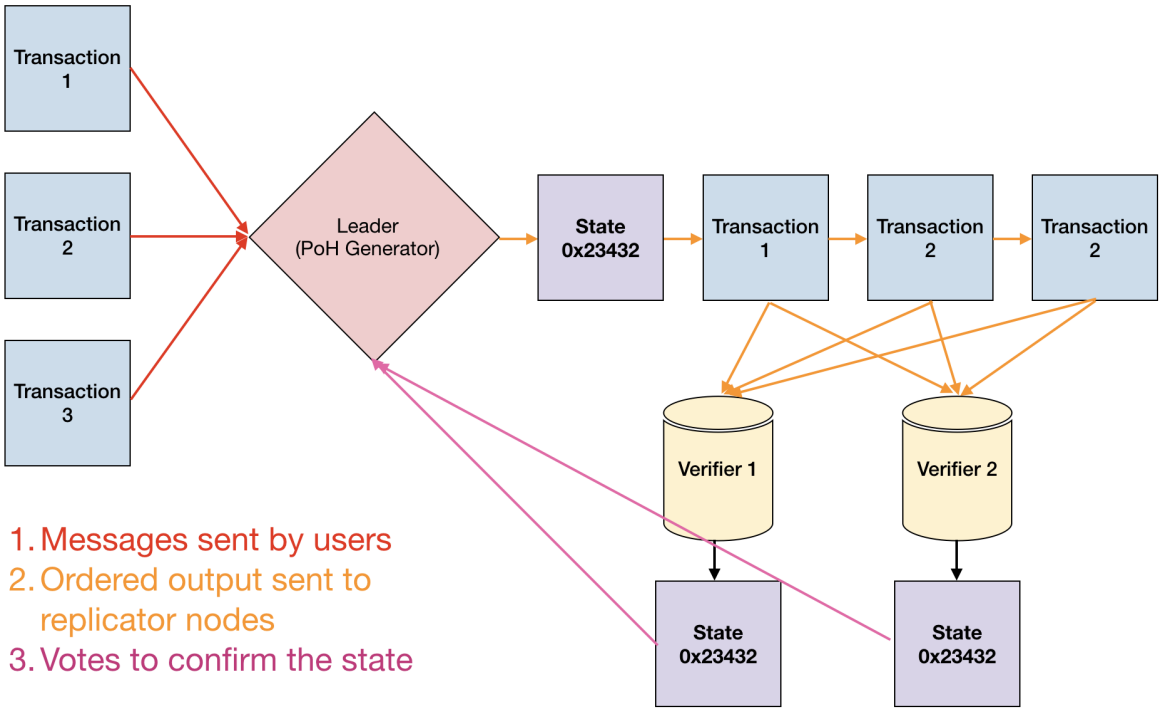


図 1　ネットワーク上のトランザクションフロー

行結果を確認してそれに署名を付与する。この署名付きの計算結果は票となり、ネットワーク上の正史を選び出すための合意形成アルゴリズムに用いられる。

ネットワークが分断されていない状況下では、常に Leader はネットワーク上にひとつ存在する。各 Verifier ノードは Leader と同性能のハードウェアを備えるようにし、PoS に基づいて Leader として選出される可能性が与えられる。ここで挙げられた PoS のアルゴリズムはセクション 5.6 で詳しく説明する。

ネットワークの分断が発生してしまった場合、CAP 定理に基づくと一貫性は可用性より優先される。大規模な分断が発生する状況を想定して、本資料は任意のサイズで分断された状態からネットワークを復旧させるメカニズムを説明する。詳細はセクション 5.12 を参照されたい。

4　Proof of History

Proof of History は任意の二つのイベントに挟まれた時間の経過を暗号学に検証できる方法を提供する。これには入力から出力を予測することが不可能な暗号学に安全な関数で計算することが求められる。一連の計算は単一コア上で行われ、前回計算のアウトプット

は今回計算のインプットとして使われる。関数が呼び出される度、その計算結果と呼び出し回数を記録する。ここまで行われた計算の入出力を分割して、外部マシンの複数コアに並列で再計算させることで、計算結果に対する正否の検証が行われる。データそのもの、あるいはデータのハッシュ値をシーケンスに加えることは、タイムスタンプに相当する情報をそのデータに付与することにあたる。つまりそれは状態、順序、データの記録が、次のハッシュ計算が開始された以前に完了していたことの証明に他ならないからだ。この設計は、複数のシーケンス生成者同士が互いのシーケンスに含まれる情報を提供し、その情報を計算のインプットとして混ぜ込み、同期を取ることができる平行スケーリングをサポートする。この平行スケーリングについてはセクション 4.4 で少し踏み込んだ議論を行う。

4.1 概要説明

システムは次のように設計される。実行前に計算結果を予測できない暗号学的ハッシュ関数 (例：`sha256`、`ripemd` など) を用いる。無作為に選ばれた値を初期入力値としてハッシュ計算を開始、今回の計算結果を次回計算の入力値として使う。今回計算も次回計算も実行するハッシュ関数は同じである。関数の呼び出し回数および計算結果は順次記録される。またここで登場した初期入力値は、当日のニューヨークタイムズ紙の見出しなど、任意の文字列を選択して良い。

PoH シーケンス

順番	計算内容	計算結果
1	sha256(” 無作為に選ばれた初期入力値”)	`hash1`
2	sha256(`hash1`)	`hash2`
3	sha256(`hash2`)	`hash3`

図中の `hashN` は N 番目に行われたハッシュ計算の結果を表している。

ここで要求されるのは都度ハッシュ計算結果と順序番号を発行することに尽きる。

PoH シーケンス

順番	計算内容	計算結果
1	sha256(” 無作為に選ばれた初期入力値”)	`hash1`
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`

ハッシュ関数に衝突耐性が備わっている限り、この一連のハッシュ値はシングルスレッドのみで計算可能である。これは 300 番目のハッシュ計算結果は、当該ハッシュ計算を 300 回繰り返すまで求まらないことから明らかである。この制約から 0 番目から 300 番目までの計算が終わるまでには相応の時間が経過していることが分かる。

図 2 中にあるハッシュ値 `62f51643c1` は 510144806912 番目に求められ、ハッシュ値 `c43d862d88` は 510146904064 番目に求められたことを表している。既に説明した通り、510144806912 番目と 510146904064 番目の間にかかった計算時間の分だけ現実世界の時間も経過することがわかる。

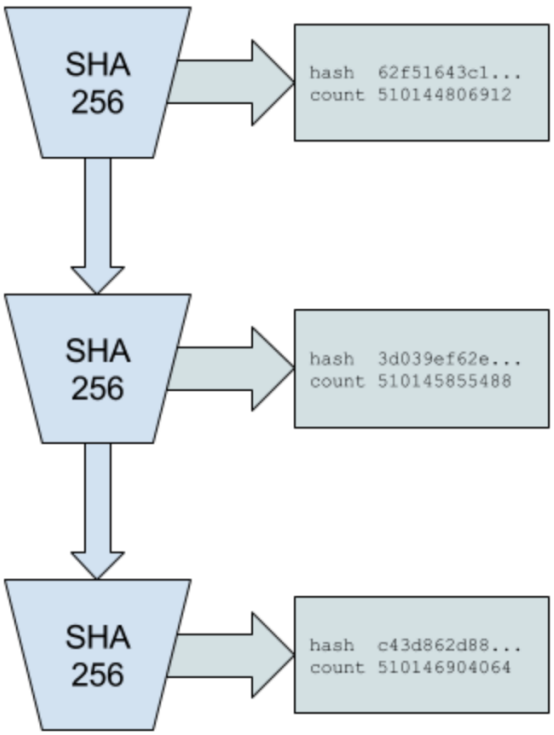


図 2　Proof of History シーケンス

4.2　イベントのタイムスタンプ

上述の通り、このハッシュ値のシーケンスは、あるデータがシーケンスに挿入される以前に、対象のデータが作成されていたことの記録として用いることができる。それはデータと現在の順序に弾き出されて来たハッシュ値に「combine」関数を適用することで、順序と値が組み合わされた計算結果が記録として残るためである。このとき任意のイベントデータが暗号学的にユニークなハッシュ値に変換される。combine 関数は常に衝突耐性を備え、単純に計算結果を積み重ねていくことができる。結果的に得られたハッシュ値はそのデータにとってのタイムスタンプとなる。それは以前のハッシュ値を基にして生成されたハッシュ値であるため、以前に記録されたデータよりも後に生成されていることが言えるからである。

PoH シーケンス

順番	計算内容	計算結果
1	sha256(” 無作為に選ばれた初期入力値”)	`hash1`
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`

例えば写真撮影など、なにか電子データが作成されるようなイベントが発生したとき、

データを含む PoH のシーケンス

順番	計算内容	計算結果
1	sha256(” 無作為に選ばれた初期入力値”)	`hash1`
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`
336	sha256(append(`hash335`, 撮影データの sha256 値))	`hash336`

`Hash336` は `hash335` と撮影データの `sha256` 値を入力値として計算されている。こうして撮影データの `sha256` 値とその順番はシーケンスに記録される。つまり入力値さえ特定できていれば、誰でもこのシーケンスに対する変更内容を再現して検証することができるのだ。シーケンス上の各部分は並列で検証できることになるが、詳しくはセクション 4.3 で説明する。

POH シーケンス

順番	計算内容	計算結果
1	sha256(” 無作為に選ばれた初期入力値”)	`hash1`
200	sha256(hash199)	`hash200`
300	sha256(hash299)	`hash300`
336	sha256(append(hash335, 撮影データ 1 の sha256 値))	`hash336`
400	sha256(hash399)	`hash400`
500	sha256(hash499)	`hash500`
600	sha256(append(hash599, 撮影データ 2 の sha256 値))	`hash600`
700	sha256(hash699)	`hash700`

表 1　2 つのイベントを挿入した PoH シーケンス

一連の計算はひとつずつ順番に行わざるを得ない仕組みから、ある情報がシーケンスに挿入されたタイミングは、それ以降に計算されたハッシュ計算が完了するより以前であることを明らかにできる。

表 1 のシーケンスから**撮影データ 2** が `hash600` の計算完了より以前に、**撮影データ 1** が `hash336` の計算完了より以前に作成されたことが分かる。ハッシュ値をシーケンスに書き込むことは、結果的に後続すべての計算結果に影響を与えることになる。シーケンスへ挿入したいデータに適用されるハッシュ関数が衝突耐性を備える限り、今後どのようなデータが追加されるか分かっていたとしても、未来のシーケンスを事前に予測することは不可能である。

シーケンスに挿入されるデータは無加工でも、メタデータ付きハッシュ値でも良い。

図 3 では入力値 `cfd40df8…`が Proof of History のシーケンスへ挿入されている。挿入された順番は 510145855488 で、その時点の状態は `3d039eef3` である。将来生成されるすべてのハッシュ値は、今回のシーケンスに対するデータ挿入により影響を受ける。その影響範囲を図中では色付きで表現している。

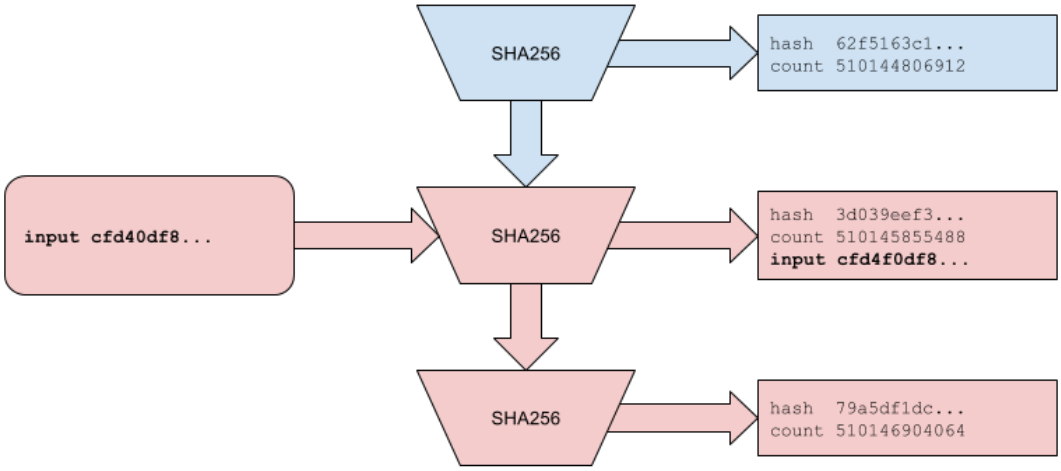


図 3　Proof of History へのデータの挿入

シーケンスを観察しているノードは、すべてのイベントが挿入された順序と、任意のふたつの挿入時点の間で経過した時間を見積もることができる。

4.3　検証

検証時の計算はマルチコアで実行可能なので、生成時よりも圧倒的に短い時間で完了することができる。

コア 1

順番	計算内容	計算結果
200	sha256(hash199)	hash200
300	sha256(hash299)	hash300

コア 2

順番	計算内容	計算結果
300	sha256(hash299)	hash300
400	sha256(hash399)	hash400

例えば 4000 コアを備える GPU を使って計算する場合、検証者は生成されたシーケンスを 4000 に分割して、先頭から末尾までのハッシュ計算が正しく行われていたかを並列計算で確認できる。シーケンス生成にかかる計算時間を次のように計算できるとすると、

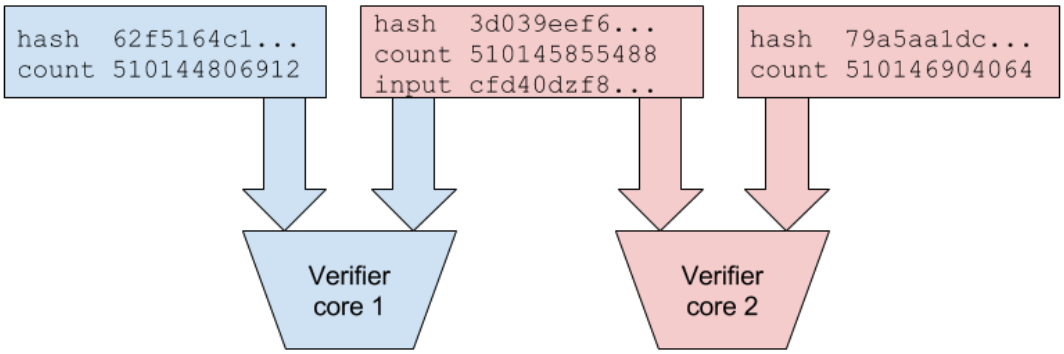


図 4　複数コアによる検証

$$\frac{\text{ハッシュ計算の総回数}}{\text{1 コアで処理される秒間ハッシュ計算回数}}$$

検証にかかる計算時間は、

$$\frac{\text{ハッシュ計算の総回数}}{(\text{1 コアで処理される秒間ハッシュ計算回数} * \text{検証に使えるコア数})}$$

図 4 では分割されたシーケンスを各コアが並列で検証している。すべてのインプットは順番と状態と併せて計算結果に記録され、検証者は同じ計算を繰り返すことでシーケンスを複製していく。赤色に染められたハッシュ値は、シーケンスがデータ挿入により変更されたことを指している。

4.4　平行スケーリング

複数の Proof of History 生成者同士でシーケンスの状態を共有することで、平行スケーリングの実現を可能とする。このスケーリングはシャーディングを使わずに果たせる。2 つの生成者でスケーリングする場合、双方のシーケンスに記録されている全てのイベントの順序を分かるフルシーケンスを構築するためには、各生成者が導き出した計算結果が必要となる。

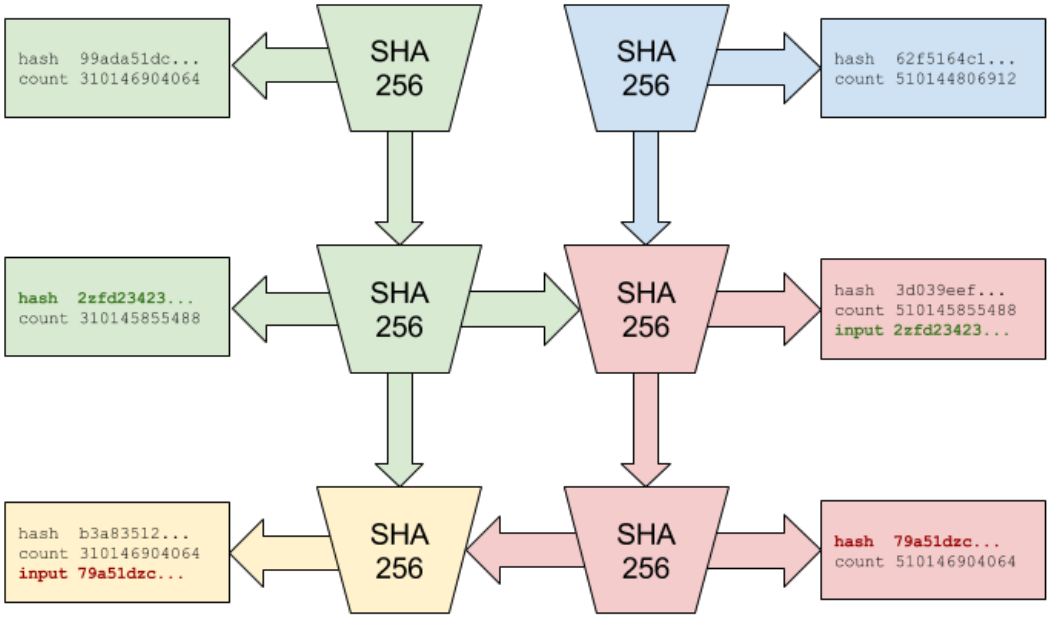


図 5　同期する二つの生成者

PoH 生成者 A		
順番	ハッシュ値	データ
1	hash1a	
2	hash2a	hash1b
3	hash3a	
4	hash4a	

PoH 生成者 B		
順番	ハッシュ値	データ
1	hash1b	
2	hash2b	hash1a
3	hash3b	
4	hash4b	

図中のように生成者 A と B が参加している場合、A は B から直近の状態をデータパケット（hash1b）として受け取る。生成者 A が次に取る状態のハッシュ値は、生成者 B から提供された状態を含めて計算される。よって hash1b は hash3a より以前に計算済みであったことを示すことができる。この仕組みは、例えば 3 つ生成者が存在しているとき、$A \leftrightarrow B \leftrightarrow C$ のように 1 組（A と C）が直接同期を取っていなくても A、B、C 間のフルシーケンスを求めることは可能である。

定期的に同期することで、各生成者は外部のトラフィックを処理することが可能となり、システム全体として対処可能なイベント数を増幅させることができるようになる。ただしネットワークを通した同期であるため、生成者間で生じるレイテンシ分だけ時間の計測精度が犠牲となる。この同期中に発生するグローバルな順序決定の問題は、ハッシュ値順にするなどの事前に取り決めた方法で順序を決定づける。

図 5 は二つの生成者が出力した状態を共有し合い、互いのシーケンスに挿入している様子を表している。色の変化は同期によりシーケンスに影響を受けたことを表現している。同期時に共有されたハッシュ値は色付きの太字で表現されている。

この同期方法により、$A \leftrightarrow B \leftrightarrow C$ のように直接同期していない生成者同士でも、三者間のイベント順序は矛盾なく決定することができる。

このスケーリング方法は回線稼働率が枷になる。10×1 gbps の回線で構成されるネットワーク環境では、回線稼働率が 0.999 のとき、ネットワーク全体の回線稼働率は $0.999^{10} = 0.99$ となる。

4.5 一貫性

生成済みのシーケンス上の最後の計算結果を次回計算の入力値にすることで、一貫性と攻撃耐性が維持されると期待される。

PoH シーケンス A		
順番	データ	計算結果ハッシュ値
10		**hash10a**
20	Event1	**hash20a**
30	Event2	**hash30a**
40	Event3	**hash40a**

PoH 隠れシーケンス B		
順番	データ	計算結果ハッシュ値
10		**hash10b**
20	Event3	**hash20b**
30	Event2	**hash30b**
40	Event1	**hash40b**

悪意を持った PoH 生成者はイベントの発生順序を変えた隠れシーケンスを作る可能性がある。発生するイベントの情報へすぐにアクセスできるなら、正しいシーケンスよりも素早く隠れシーケンスを生成することも可能である。

この攻撃を防ぐために、クライアントがイベントを送信するときは、クライアント自身が正規であると認識するシーケンス上の最新ハッシュ値を送信内容に含めることを求められる。クライアントが Event1 を作成するとき、それに最新ハッシュ値を付与する必要がある。

PoH シーケンス A

順番	データ	計算結果ハッシュ値
10		`hash10a`
20	Event1 = append(event1 data, `hash10a`)	`hash20a`
30	Event2 = append(event2 data, `hash20a`)	`hash30a`
40	Event3 = append(event3 data, `hash30a`)	`hash40a`

シーケンス発行時に Event3 が hash30a を参照しているとき、そのハッシュ値がイベント発生以前に存在しない場合に、シーケンス利用者は参照先のシーケンスに異常を検知する。こうして一部分のイベント順序を書き換える攻撃が有効になるのは、クライアントがイベント送信前に検知したハッシュ値の順番から、そのイベント情報はシーケンスに挿入されるまでの間だけに制限される。つまり極めて短期間の間に発生したイベントの前後関係に影響受けない問題を解決するクライアントソフトウェアに絞って利用することが望ましい。

悪意のある PoH 生成者がクライアント側で付与したハッシュ値を書き換えることを防ぐため、クライアントはイベントデータとハッシュ値に署名を付与した上で送信を行う。

PoH シーケンス A

順番	データ	計算結果ハッシュ値
10		hash10a
20	Event1 = sign(append(event1 data, hash10a), 利用者の秘密鍵)	hash20a
30	Event2 = sign(append(event2 data, hash20a), 利用者の秘密鍵)	hash30a
40	Event3 = sign(append(event3 data, hash30a), 利用者の秘密鍵)	hash40a

このデータの検証を行うためには、署名と付与されたハッシュ値がシーケンス上で保たれていることが必要とされる。

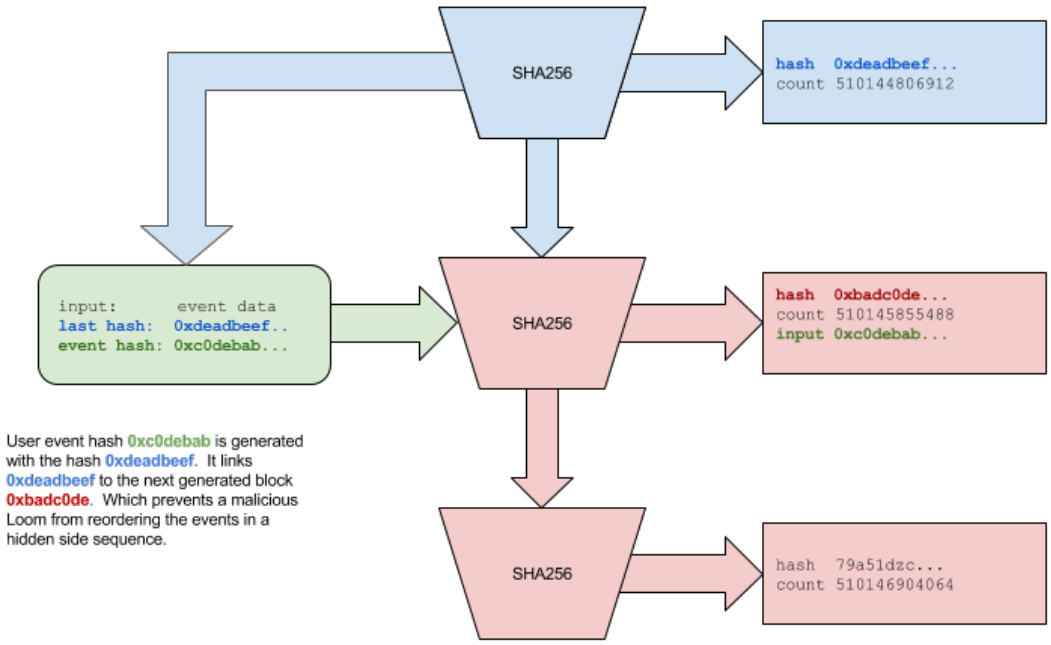


図 6　入力受付時にハッシュの存在性を検証

検証方法：

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

図 6 では、クライアントからのインプットに付与されたハッシュ値 0xdeadbeef…が、シーケンス上の遠くない過去まで遡り、それが存在していることに依存している。図上部の青い部分はクライアントが生成済みのハッシュ値を参照していることを表している。クライアントからのメッセージはハッシュ値 0xdeadbeef…を含む場合においてのみ有効とされる。図中の赤い部分はクライアントからの入力により影響を受けた箇所を指している。

4.6　オーバーヘッド

秒間 4000 回のハッシュ計算が可能な環境では、秒間 160KB のデータが追加で生成され、その検証のためコア数 4000 の GPU を使っておよそ 0.25-0.75 ミリ秒のオーバーヘッ

ドが発生する。

4.7 攻撃

4.7.1 順序の逆転

イベントの順序を逆転させるためには、攻撃者は次のイベントを受け取った後から不正なシーケンスを生成し始めなければならない。この遅れにより、悪意のないノード同士で不正のない情報の交換が可能となる。

4.7.2 速度

複数の生成者が稼働することでより強力な攻撃耐性を得られる。生成者は広帯域な通信を教授でき、他の生成者と共有する状態をシーケンスに混ぜ込む機会を増やせる。また高速低帯域の生成者であっても、周期的に広帯域の生成者とイベントを混ぜ合うことができる。

こうした高速なシーケンスにより、悪意のあるノードが生成する不正なシーケンスは矯正されることになる。

4.7.3 ロングレンジアタック

ロングレンジアタックとは、かつて利用されていた古い秘密鍵を悪用して、不正な台帳を作ることを指す [10]。Proof of History はロングレンジアタックに対する防衛手段となりえる。攻撃に利用できる秘密鍵を入手した悪意のあるノードは、再作成しようとする履歴範囲の開始時点から経過したのと同程度の時間を再現する必要がある。それにはネットワーク上のノードが備える計算速度よりも高速なプロセッサを利用する必要がある。さもなくば攻撃者が作成する不正なシーケンス上で再現される時間は、正規のシーケンスには永遠に追いつくことができない。

さらに時間の基準を統一することで、Proof of Replication の構築をよりシンプルにすることができる（詳細はセクション 6 を参照）。すべてのネットワーク参加者が一箇所のシーケンス上に記録されたイベントに依拠するためである。

PoRep と PoH を一緒に活用することで空間的にも時間的にも堅牢性を高めることができる。

5　Proof of Stake による合意形成

5.1　概要説明

ここで取り上げる Proof of Stake は次に挙げる要件を満たすために設計されている。Proof of History の生成者が生み出したシーケンスを迅速に承認できること、逐次 Proof of History 生成者を選出できること、そして不正を働いている検証者たちへの懲罰ができること。このアルゴリズムはすべてのネットワーク参加ノードがタイムアウト発生前にメッセージを受け取れることを前提としている。

5.2　用語の説明

ボンド　Proof of Work における設備コストに相当する。Proof of Work では、採掘者はハードウェアや電気にお金を支払い、ブロックチェーンのブランチのひとつを支持する。つまりボンドとは、トランザクションを承認する権利を得るために検証者たちが担保として献上する価値のことを指す。

スラッシング　Proof of Stake における nothing at stake 問題への対処法のひとつ [7]。検証者が他のブランチを支持していたことが証明された場合、正規のブランチは検証者がステークしていた資本を処分することができる。つまり検証者の支持するブランチが発散しないよう設計された経済的インセンティブである。

大多数派　全検証者をボンドで重み付けしたときの $\frac{2}{3}$ 以上の割合を指す。大多数派からの支持によりネットワークは合意形成を完了する。つまり少なくとも $\frac{1}{3}$ の支持を不正なブランチに流さなくては、正規ブランチへの合意形成を避けることができない。この攻撃を仕掛けるためには、通貨の時価総額の $\frac{1}{3}$ 近いコストが必要であることが示唆される。

5.3　ボンディング

ボンディングには一定量以上の通貨を必要とし、実行すると所有者に紐づく形でボンディング用の口座へ出金される。ボンディング中の残高は消費できず、所有者が引き出すまでボンディング用の口座に置かれ続ける。ボンディングされた残高は一定時間が経過するまで引き出すことはできない。ボンディング中の参加者たちから大多数の承諾を受けてボンドは有効化される。

5.4　投票

Proof of History 生成者は、予め決められた期間の状態に対し署名を行うことができる。ボンディング中の参加者たちは各自が状態に行った署名によって、生成者の署名内容に対する承認を行わなければならない。つまり単純な Yes 票による投票で、否定票はない。

ボンディング中の参加者から所定時間内に大多数の支持を受けた場合、そのブランチは正式に受け入れられる。

5.5　アンボンディング

N 票を失った通貨は枯れたものとみなされ投票に参加する権利を失う。所有者はアンボンディングの手続きを踏んで資本を取り戻すことができるようになる。

ここに挙げられた N の値は、有効票数に対し枯れた通貨量の比率により動的に変わる。N は枯れた通貨量の増加に合わせて増えていく。これにより規模の大きいネットワークの分断が起きたとき、大きなブランチが小さなブランチよりも速く復旧する仕組みができる。

5.6　選挙

PoH 生成者の選挙は生成者からエラーが検出されたときに開かれる。票をもっとも多く集めた検証者が次の生成者として選出される。同票の場合は公開鍵アドレスの値が大きい方が選択される。

新しいシーケンス開始時にはネットワークの合意が要求されるが、合意形成前にリーダーがエラーを起こした場合は、得票率が次点だった検証者が新たなリーダーとして選択される。このとき改めて合意形成を行わなければならない。

決議を変更するにはより高速な PoH シーケンス上で投票を行う必要がある。また改めて投じる票には変更前の情報を含める必要がある。これを怠るとスラッシングされる恐れがある。シーケンスへの承認が一定以上蓄積されたとき、決議の変更が可能となる。

リーダーが選出されると、その役割を引き継ぐことのできる副系を決めることができる。リーダーに異常が起きたとき副系はその役割を引き継ぎ、代わりを担う。

例外が検出された、あるいは予め定められた時点で、副系はリーダーを代わり、下位の生成者も繰り上がるような設計になる。

5.7　選挙の開催条件

5.7.1　PoH 生成者の分断

PoH 生成者は生成したシーケンスに署名をおこなう。PoH 生成者の鍵が攻撃者に利用できる状況下においてのみ分断は発生しうる。異なるシーケンスに同じ PoH 生成者の署名が与えられることで分断が検知される。

5.7.2　実行系の例外

PoH 生成者にハードウェアの誤動作やバグ、あるいは意図的に起こされたエラーにより、不正な状態や検証者の計算結果が不一致する状態が生成されることがある。これを検知した検証者はゴシップを流し、それが引き金となって新たに選挙が開催される。不正なシーケンス状態を受け入れた検証者は必ずスラッシングを受けてボンドを没収される。

5.7.3　ネットワークのタイムアウト

ネットワークにタイムアウトが発生した場合は選挙が開催される。

5.8　スラッシング

検証者が分断したシーケンス両方に票を投じるとスラッシングにより罰せられる。悪意のある投票が検知されると、ボンドされていた資本は没収されて採掘プールへ送られる。

前回分断を起こしたシーケンスへ投票していた場合でも、それは悪意のある投票とみなされず、ボンドへのスラッシングは執行されない。ただし今回分断を起こしたシーケンスへの票が無効化される。

PoH 生成者が生成した不正なハッシュへ票が投じられた場合にもスラッシングが執行される。生成者は不正なシーケンス状態を無作為に発行するが、これは副系へのフォールバックを発生させる。

5.9　副系の選挙

副系あるいはそれ下位の Proof of History 生成者の選挙も開催される。選挙開催の提案はリーダー生成者のシーケンス上で行われる。タイムアウト前に合意形成がなされると副系は選出され、その役割を担う。リーダーから権限の引き渡しが行われる旨のメッセージがシーケンスに挿入される。不正な状態がシーケンスへ挿入された場合は、副系への

フォールバックが強制される。

副系が選出済みの状態でリーダーが障害を起こした場合、選挙中はその副系が最初のフォールバック先として扱われる。

5.10 可用性

分断状態になりうる CAP に基づいたシステムは、一貫性あるいは可用性のいずれかを優先しなければならない。我々のシステムは可用性を選択したが、時間の計測者という立場から、実時間に基づくタイムアウト以前においては一貫性が保たれる設計を採用する。

Proof of Stake の承認者は一定量の通貨をステーク状態にロックアップすることで、所定の取引に投票する権利を得る。ロックアップの実施も他の取引と同様 PoH のストリームに挿入される取引として扱われる。投票するために PoS 承認者は状態ハッシュ値に署名する必要がある。PoH の台帳で発生したすべての取引完了後の状態を基に、投票が行われるためである。投票の実施も PoH のストリームに挿入される。PoH の台帳をみれば、各投票がどのぐらいの時間間隔で行われていたか推測できる。分断発生時にはどのぐらいの期間、承認者たちが利用不可に陥っていたか推し測ることもできる。

分断の解決にかかる実時間を圧縮するため、利用不可に陥った承認者のステーキングを解除する動的アプローチを検討したい。ステーキングを解除された承認者は合意形成の頭数に入らないので、合意形成にかかる時間が圧縮される仕組みである。$\frac{2}{3}$ を越える承認者が利用可能ならば合意形成に至るまでに必要なハッシュ計算回数は抑えられるため、ステーキング解除はすぐに完了できる。利用可能な承認者の割合が $\frac{2}{3}$ 以下かつ $\frac{1}{2}$ を超えるときは、合意形成までに必要なハッシュ計算回数が増加するため、ステーキング解除にかかる時間が延びてしまう。半数以上の承認者が利用不可になる規模の大きな分断が発生した状況では、ステーキング解除の処理に多大な時間がかかる。そのような状況でも取引や投票は可能だが、$\frac{2}{3}$ の合意形成に至るには相当数のハッシュ計算をこなし、利用不可に陥った承認者のステーキング解除が完了している必要がある。復旧にかかるネットワークの時間差は、ネットワーク利用者たちがどのパーティションを使い続けたいか選択するための猶予にあたる。

5.11 復旧

ここで紹介するシステムはどんな障害が起きても台帳を完全に復旧させることができる。つまり誰でも台帳上の任意の位置にハッシュ値や取引を追記することで分岐を作成し

て良い。もし分岐したシーケンス上において承認者が不在の場合、ボンドの追加がブランチ上で $\frac{2}{3}$ の支持を集めて合意形成に至るには、極めて長い時間が必要とされる。よって検証者不在の状態から完全復旧を果たすためには相当数のハッシュ計算が台帳上で行われなければならない。つまり利用不可になった検証者全てのステーキングが解除されるまで、新たなボンドにより台帳の承認を行うことは不可能となる。

5.12　状態確定

過去に何が起きたか、それがいつ頃に起きたか照合することを PoH は可能にする。PoH 生成者がメッセージストリームを流し出すごとに、承認者はその内容に署名を付けて 500 ミリ秒以内に提出することが求められる。この制限時間はネットワークの状況次第でさらに短縮される。その承認行為自体がストリームへ挿入されるため、投票そのものを直接観測してなかったとしても、それがタイムアウト発生前に完了していたか確認することは誰にでもできる。

5.13　攻撃

5.13.1　コモンズの悲劇

PoH 生成者から流れてきた状態ハッシュ値に対して PoS 承認者が検証を行わず、ひたすら承認を出し続ける状況を指す。経済的インセンティブに起因する問題である。これを回避するため、PoH 生成者は無作為なタイミングで不正なハッシュを生成し、これに投票した承認者はスラッシングされる仕掛けを用意する。不正なハッシュ値が生成されたとき適切な検証と投票が行われることで、ネットワークは副系の PoH 生成者へ即座に切り替えることになる。

検証者は 500 ミリ秒等の短い時間内に回答を返すことが求められる。悪意を持った検証者が他の検証者の投票内容を観測して、それをいち早くストリームへ挿入できてしまう確率を低減するため、タイムアウトは十分短い時間を設定される必要がある。

5.13.2　PoH 生成者との談合

PoH 生成者と談合している承認者は、不正なハッシュが流れてくるタイミングを事前に知ることができる問題を指す。ただしこれは PoH 生成者が承認者の役割も賄える通貨量以上をステーキングしていることと違いがない。結局 PoH 生成者は状態ハッシュ値の計算を行わなくてはならない。

5.13.3　検閲

$\frac{1}{3}$ のボンド保有者が新規ボンド含むシーケンスの受け入れを拒否したとき、検閲が起きているか DoS が発生していると考えられる。ここでは受け入れを拒否している $\frac{1}{3}$ のボンド保有者を”ビザンチン・ボンド保有者”と呼ぶ。この種の攻撃に対しては、ボンドが枯れていくサイクルを動的に調整することでプロトコルは耐性を身につけることができる。DoS 発生時、ビザンチン・ボンド保有者よりも規模の大きい側の保有者たちがビザンチン・ボンド保有者を検閲し、ネットワークを切り離す。これにより規模の大きい側の保有者たちは、ビザンチン・ボンドを時間経過とともに無力化していくことができるので、規模の小さい側のビザンチン・パーティションは長く生き残ることができない。

アルゴリズムは次のように動作する。多数派の合意によりリーダーが選出される。リーダーはビザンチン・ボンド保有者の参加権を取り上げる。Proof of History 生成者はビザンチン・ボンド所有者の数が十分に減り、多数派の割合が大多数派の規模に至るまでの経過時間を記録するためシーケンス生成を継続する。どの程度のボンドが有効化されているかに依って、ボンドが枯れていく速度が動的に決まる。そして分岐した少数派のネットワークは、多数派のネットワークが大多数派に格上げされるまで以上の長い時間を待たされることになる。大多数派が成立したとき、ビザンチン・ボンド保有者にスラッシングの罰を下すことができる。

5.13.4　ロングレンジアタック

PoH は本質的にロングレンジアタックに対する耐性を持つ。過去の一時点からロングレンジアタックを試みる攻撃者は、PoH 生成者の計算能力を越える速度で計算しない限り、永遠に正規台帳を差し替えることはできない。

合意形成プロトコルは第二の防壁である。これは如何なる攻撃も全検証者をステーク解除するより長い時間をかける必要があるためである。これは台帳の履歴に刻まれる可用性に”差”を生み出す要因ともなる。つまり二つの台帳を同じハッシュ計算回数時点で比べた時、最大断片数が最小の台帳が正規のものと考えることができる。

5.13.5　ASIC 攻撃

このプロトコルでは ASIC による攻撃のタイミングが二つ存在する——ネットワーク分断時と状態確定時における作為的タイムアウトである。

分断時の ASIC 攻撃については、ステーキング解除速度が非線形であり、大規模な分断を起こしているネットワークでは圧倒的に遅いオーダーを弾き出すため、ASIC を用いる

メリットが極めて低い。

状態確定時の ASIC 攻撃については、ビザンチン側の検証者たちにボンドのステーキング承認待ちを許し、協力者の PoH 生成者を使って票をねじ込めてしまうという弱点である。PoH 生成者は ASIC を使って 500 ミリ秒分のハッシュ計算をそれより短い時間で完了し、計算結果を協力者ノードに伝えることができてしまう。だが、そもそも PoH 生成者がビザンチン側に属していれば、不正なハッシュ値を挿入するタイミングを協力者ノードへ事前に知らせておかない理由がない。この状況はちょうど PoH 生成者と協力者が各ボンドのステークを組み合わせて同じ身元を共有し、さらに同じハードウェア一式を使っている状況と何ら変わりはない。

6 Proof of Replication のストリーミング

6.1 概要説明

Filecoin proposed a version of Proof of Replication [6]. The goal of this version is to have fast and streaming verifications of Proof of Replication, which are enabled by keeping track of time in Proof of History generated sequence. Replication is not used as a consensus algorithm, but is a useful tool to account for the cost of storing the blockchain history or state at a high availability.

6.2 Algorithm

As shown in Figure 7 CBC encryption encrypts each block of data in sequence, using the previously encrypted block to XOR the input data.

Each replication identity generates a key by signing a hash that has been generated Proof of History sequence. This ties the key to a replicator' s identity, and to a specific Proof of History sequence. Only specific hashes can be selected. (See Section 6.5 on Hash Selection)

The data set is fully encrypted block by block. Then to generate a proof, the key is used to seed a pseudorandom number generator that selects a random 32 bytes from each block.

A merkle hash is computed with the selected PoH hash prepended to the each slice.

The root is published, along with the key, and the selected hash that was generated. The replication node is required to publish another proof in N hashes as they are

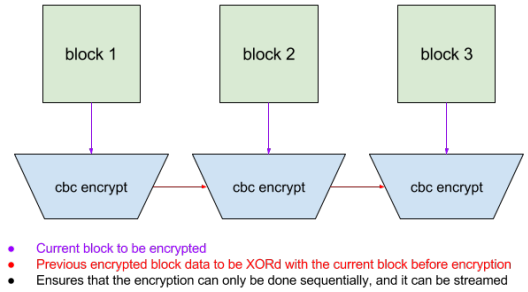


図 7　Sequential CBC encryption

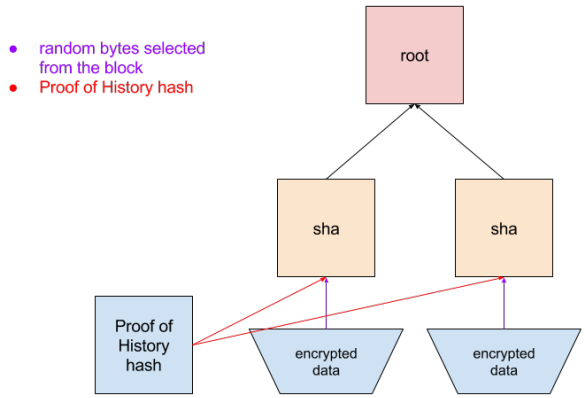


図 8　Fast Proof of Replication

generated by Proof of History generator, where N is approximately $\frac{1}{2}$ the time it takes to encrypt the data. The Proof of History generator will publish specific hashes for Proof of Replication at a predefined periods. The replicator node must select the next published hash for generating the proof. Again, the hash is signed, and random slices are selected from the blocks to create the merkle root.

After a period of N proofs, the data is re-encrypted with a new CBC key.

6.3 Verification

With N cores, each core can stream encryption for each identity. Total space required is $2blocks * Ncores$, since the previous encrypted block is necessary to generate the next one. Each core can then be used to generate all the proofs that derived from the current encrypted block.

Total time to verify proofs is expected to be equal to the time it takes to encrypt. The proofs themselves consume few random bytes from the block, so the amount of data to hash is significantly lower then the encrypted block size. The number of replication identities that can be verified at the same time is equal to the number of available cores. Modern GPUs have 3500+ cores available to them, albeit at $\frac{1}{2}$-$\frac{1}{3}$ the clock speed of a CPU.

6.4 Key Rotation

Without key rotation the same encrypted replication can generate cheap proofs for multiple Proof of History sequences. Keys are rotated periodically and each replication is re-encrypted with a new key that is tied to a unique Proof of History sequence.

Rotation needs to be slow enough that it' s practical to verify replication proofs on GPU hardware, which is slower per core than CPUs.

6.5 Hash Selection

Proof of History generator publishes a hash to be used by the entire network for encrypting Proofs of Replication, and for using as the pseudorandom number generator for byte selection in fast proofs.

Hash is published at a periodic counter that is roughly equal to $\frac{1}{2}$ the time it takes to encrypt the data set. Each replication identity must use the same hash, and use the signed result of the hash as the seed for byte selection, or the encryption key.

The period that each replicator must provide a proof must be smaller than the encryption time. Otherwise the replicator can stream the encryption and delete it for each proof.

A malicious generator could inject data into the sequence prior to this hash to

generate a specific hash. This attack is discussed more in 5.13.2.

6.6 Proof Validation

The Proof of History node is not expected to validate the submitted Proof of Replication proofs. It is expected to keep track of number of pending and verified proofs submitted by the replicator' s identity. A proof is expected to be verified when the replicator is able to sign the proof by a super majority of the validators in the network.

The verifications are collected by the replicator via p2p gossip network, and submitted as one packet that contains a super majority of the validators in the network. This packet verifies all the proofs prior to a specific hash generated by the Proof of History sequence, and can contain multiple replicator identities at once.

6.7 Attacks

6.7.1 Spam

A malicious user could create many replicator identities and spam the network with bad proofs. To facilitate faster verification, nodes are required to provide the encrypted data and the entire merkle tree to the rest of the network when they request verification.

The Proof of Replication that is designed in this paper allows for cheap verification of any additional proofs, as they take no additional space. But each identity would consume 1 core of encryption time. The replication target should be set to a maximum size of readily available cores. Modern GPUs ship with 3500+ cores.

6.7.2 Partial Erasure

A replicator node could attempt to partially erase some of the data to avoid storing the entire state. The number of proofs and the randomness of the seed should make this attack difficult.

For example, a user storing 1 terabyte of data erases a single byte from each 1 megabyte block. A single proof that samples 1 byte out of every megabyte would have a likelihood of collision with any erased byte $1 - (1 - 1/1,000,0000)^{1,000,000} = 0.63$. After 5 proofs the likelihood is 0.99.

6.7.3 Collusion with PoH generator

The signed hash is expected to be used to seed the sample. If a replicator could select a specific hash in advance then the replicator could erase all bytes that are not going to be sampled.

A replicator identity that is colluding with the Proof of History generator could inject a specific transaction at the end of the sequence before the predefined hash for random byte selection is generated. With enough cores, an attacker could generate a hash that is preferable to the replicator' s identity.

This attack could only benefit a single replicator identity. Since all the identities have to use the same exact hash that is cryptographically signed with ECDSA (or equivalent), the resulting signature is unique for each replicator identity, and collision resistant. A single replicator identity would only have marginal gains.

6.7.4 Denial of Service

The cost of adding an additional replicator identity is expected to be equal to the cost of storage. The cost of adding extra computational capacity to verify all the replicator identities is expected to be equal to the cost of a CPU or GPU core per replication identity.

This creates an opportunity for a denial of service attack on the network by creating a large number of valid replicator identities.

To limit this attack, the consensus protocol chosen for the network can select a replication target, and award the replication proofs that meet the desired characteristics, like availability on the network, bandwidth, geolocation etc...

6.7.5 Tragedy of Commons

The PoS verifiers could simply confirm PoRep without doing any work. The economic incentives should be lined up with the PoS verifiers to do work, like by splitting the mining payout between the PoS verifiers and the PoRep replication nodes.

To further avoid this scenario, the PoRep verifiers can submit false proofs a small percentage of the time. They can prove the proof is false by providing the function that generated the false data. Any PoS verifier that confirmed a false proof would be slashed.

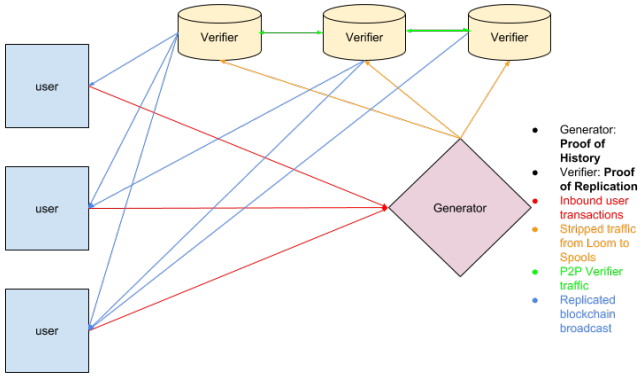


図 9　System Architecture

7　System Architecture

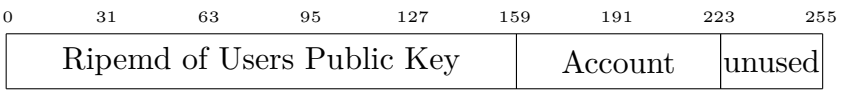
7.1　Components

7.1.1　Leader, Proof of History generator

The Leader is an elected Proof of History generator. It consumes arbitrary user transactions and outputs a Proof of History sequence of all the transactions that guarantees a unique global order in the system. After each batch of transactions the Leader outputs a signature of the state that is the result of running the transactions in that order. This signature is signed with the identity of the Leader.

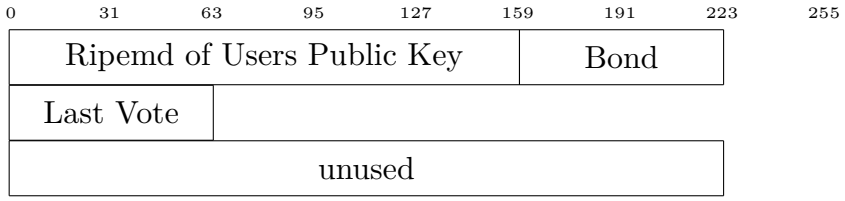
7.1.2　State

A naive hash table indexed by the user' s address. Each cell contains the full user' s address and the memory required for this computation. For example
Transaction table contains:



For a total of 32 bytes.

Proof of Stake bond' s table contains:



For a total of 64 bytes.

7.1.3 Verifier, State Replication

The Verifier nodes replicate the blockchain state and provide high availability of the blockchain state. The replication target is selected by the consensus algorithm, and the validators in the consensus algorithm select and vote the Proof of Replication nodes they approve of based on off-chain defined criteria.

The network could be configured with a minimum Proof of Stake bond size, and a requirement for a single replicator identity per bond.

7.1.4 Validators

These nodes are consuming bandwidth from Verifiers. They are virtual nodes, and can run on the same machines as the Verifiers or the Leader, or on separate machines that are specific to the consensus algorithm configured for this network.

7.2 Network Limits

Leader is expected to be able to take incoming user packets, orders them the most efficient way possible, and sequences them into a Proof of History sequence that is published to downstream Verifiers. Efficiency is based on memory access patterns of the transactions, so the transactions are ordered to minimize faults and to maximize prefetching.

Incoming packet format:

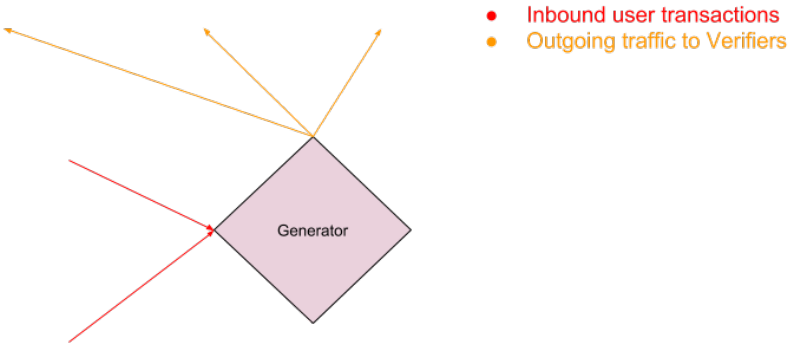
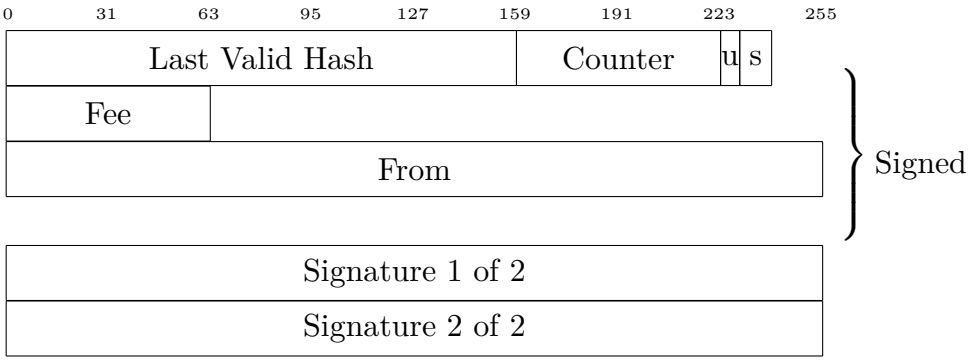
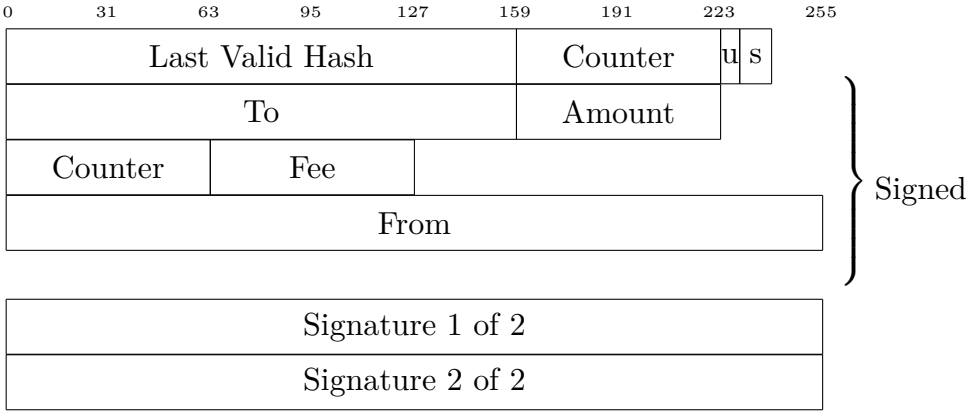


図 10　Generator network limits



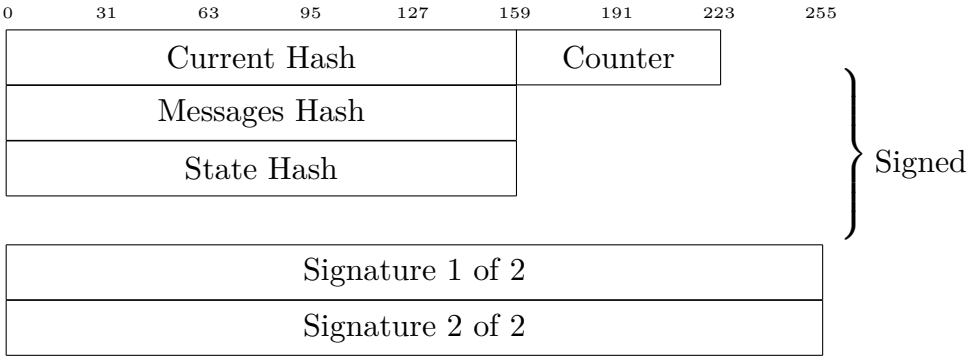
Size $20 + 8 + 16 + 8 + 32 + 3232 = 148$ bytes.

The minimal payload that can be supported would be 1 destination account. With payload:



With payload the minimum size: 176 bytes

The Proof of History sequence packet contains the current hash, counter, and the hash of all the new messages added to the PoH sequence and the state signature after processing all the messages. This packet is sent once every N messages are broadcast. Proof of History packet:



Minimum size of the output packet is: 132 bytes

On a 1gbps network connection the maximum number of transactions possible is 1 gigabit per second / 176 bytes = 710k tps max. Some loss $1 - 4\%$ is expected due to Ethernet framing. The spare capacity over the target amount for the network can be used to increase availability by coding the output with Reed-Solomon codes and striping it to the available downstream Verifiers.

7.3 Computational Limits

Each transaction requires a digest verification. This operation does not use any memory outside of the transaction message itself and can be parallelized independently. Thus throughput is expected to be limited by the number of cores available on the system.

GPU based ECDSA verification servers have had experimental results of 900k operations per second [9].

7.4 Memory Limits

A naive implementation of the state as a 50% full hashtable with 32 byte entries for each account, would theoretically fit 10 billion accounts into 640GB. Steady state random access to this table is measured at $1.1 * 10^7$ writes or reads per second. Based on 2 reads and two writes per transaction, memory throughput can handle 2.75m transactions per second. This was measured on an Amazon Web Services 1TB x1.16xlarge instance.

7.5 High Performance Smart Contracts

Smart contracts are a generalized form of transactions. These are programs that run on each node and modify the state. This design leverages extended Berkeley Packet Filter bytecode as fast and easy to analyze and JIT bytecode as the smart contracts language.

One of its main advantages is a zero cost Foreign Function Interface. Intrinsics, or functions that are implemented on the platform directly, are callable by programs. Calling the intrinsics suspends that program and schedules the intrinsic on a high performance server. Intrinsics are batched together to execute in parallel on the GPU.

In the above example, two different user programs call the same intrinsic. Each program is suspended until the batch execution of the intrinsics is complete. An example intrinsic is ECDSA verification. Batching these calls to execute on the GPU can increase throughput by thousands of times.

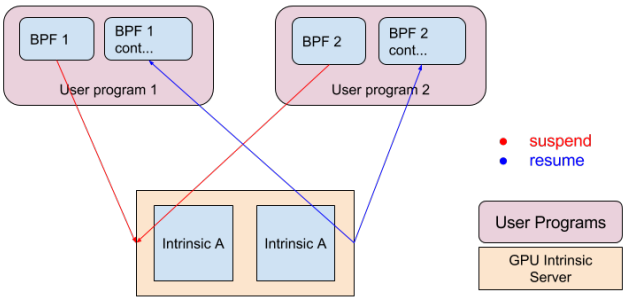


図 11　Executing　BPF programs.

This trampoline requires no native operating system thread context switches, since the BPF bytecode has a well defined context for all the memory that it is using.

eBPF backend has been included in LLVM since 2015, so any LLVM frontend language can be used to write smart contracts. It’ s been in the Linux kernel since 2015, and the first iterations of the bytecode have been around since 1992. A single pass can check eBPF for correctness, ascertain its runtime and memory requirements and convert it to x86 instructions.

参考文献

[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-algorithm/

[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