Trezoa:
高パフォ　マンスなブロックチェ　ンを実現する新しいア　キテクチャ v4.18.26

Trezoa Team
trezoateam@trezoa.com

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りながら一連のイベント発生タイミングの前後関係と経過時間を記帳し、それを証明できる仕組みである――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”という性質である。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、ビザンチン・フォールト・トレラントなステートマシーンの状態共有におけるオーバーヘッドを低減し、結果的に状態確定にかかる時間を短縮することができる。また PoH の時間管理能力を発揮でき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）とデータ保管の情報を記した台帳の偽造への対策となる。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に、このプロトコルの実装は最大 710,000TPS（秒間トランザクション数）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を前提としない [4, 5]。ネットワーク上の各ノードはローカル時間のみ参照し、他ノードのローカル時間を意識することはない。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・拒否の判断基準に用いる場合に、すべてのノードがそのメッセージに対して同様の判断を下すとは限らないことになる。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。ここでいう”時間”とはイベント間の経過時間やメッセージの順序を指す。トラストレスなネットワーク上のすべてのノードは PoH の台帳に記録された時間に依拠できるようになる。

2　本文の構成

当資料は次のように構成される。セクション 3 ではシステム設計の全体像が描かれる。セクション 4 では Proof of History の詳細が語られる。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リーダーとして Proof of History の生成を担う。これはネットワーク上のどこから参照しても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、リーダーは利用者のメッセージを順序付けする。これは RAM（主記憶装置）に格納されている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承認者と呼ばれるノードに引き渡す。承認者は各自が 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。

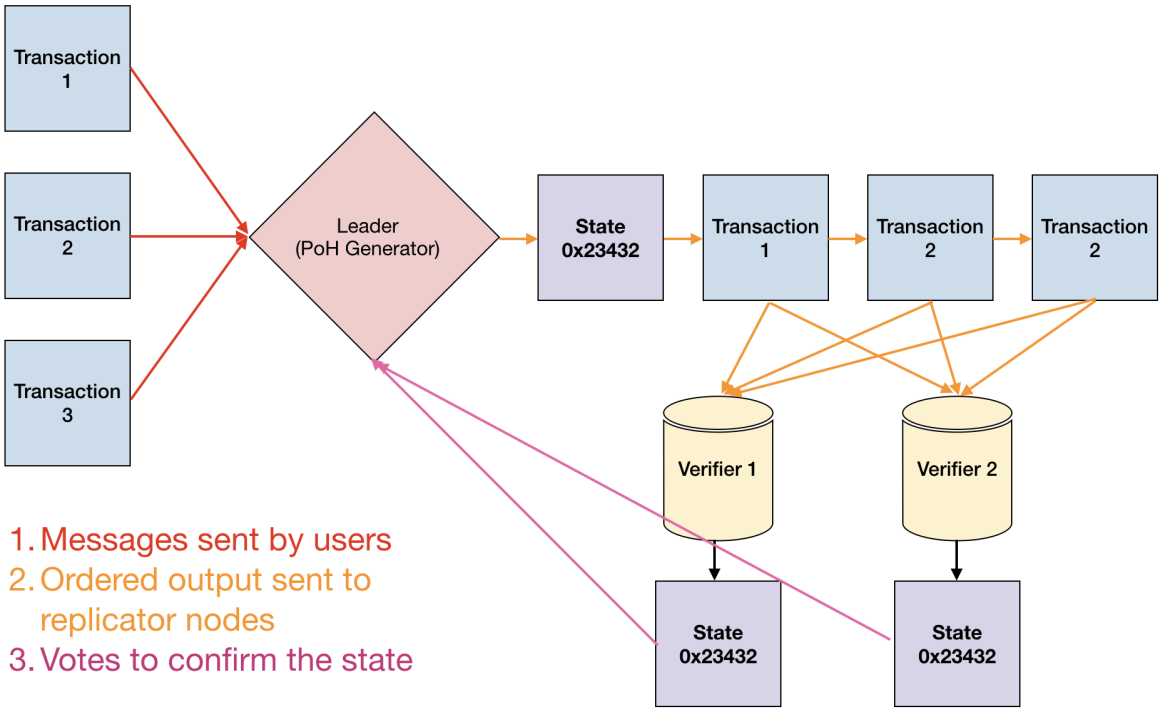


図 1　ネットワーク上のトランザクションフロー

ネットワークが分断されていない状況下では、常にリーダーはネットワーク上にひとつ存在する。各承認者ノードはリーダーと同性能のハードウェアを備えるようにし、PoS に基づいてリーダー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。

CAP 定理に基づくと、ネットワーク分断時は可用性よりも一貫性が優先される。大規模な分断が発生する状況を想定して、当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 5.10 を参照。

4　Proof of History

Proof of History は任意の二つのイベント間における経過時間を暗号学に検証できる方法を提供する。これには計算結果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。シーケンスを分割して入力値から計算結果を外部マシンで並列で再計算し、複数のコア上で検証することができる。データ（あるいはそのハッシュ値）を状態に

含めることで、データのタイムスタンプをシーケンスに加えることができる。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る以前に存在していたことの証拠となる。この設計により複数のシーケンス生成者同士が互いのシーケンスに含む情報を共有し、その情報を計算のインプットとして混ぜることで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 で議論される。

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。初期状態からは無作為に選ばれた値を入力値としてハッシュ計算を開始し、以降は今回計算の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同じハッシュ関数を用いる。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。例えば、

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。

ハッシュ計算結果と順序番号を都度発行することだけが求められる。例えば、

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の計算は単一ス

レッドだけで十分である。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することが分かる。

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。

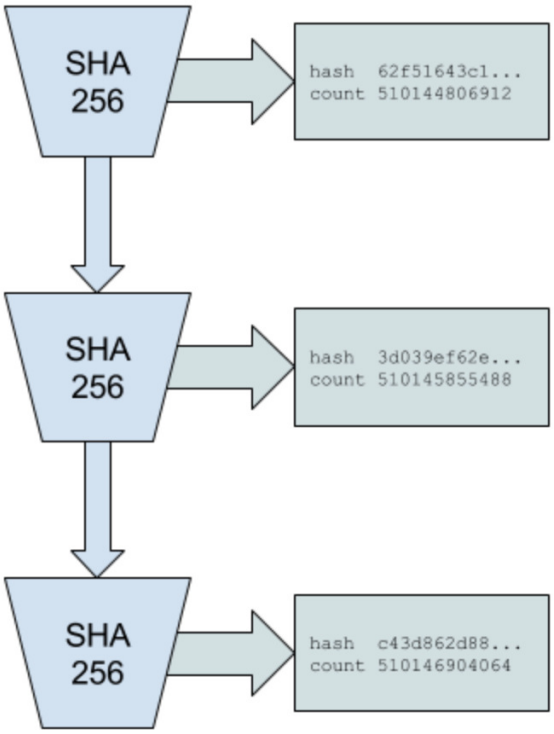


図 2　Proof of History シーケンス

4.2　イベントのタイムスタンプ

上述の通り、このような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。それはデータと前回ハッシュ値を合わせたものに「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たと言えるからである。

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の電子データが作成される外的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。またシーケンス上の各部分は並列で検証可能である。これについて詳しくはセクション 4.3 で説明する。

一連の計算はひとつずつ順番に行われる仕組みから、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で、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

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

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。

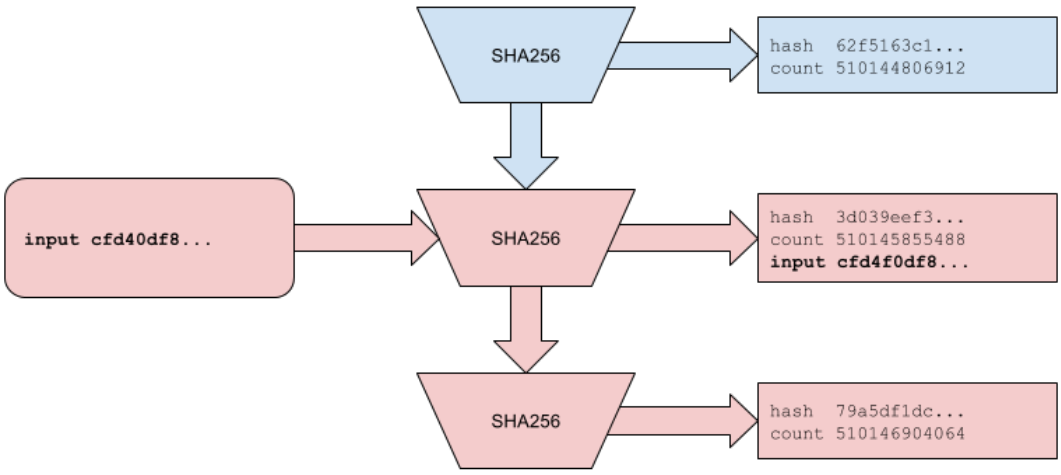


図 3　Proof of History へのデータの挿入

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、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。

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、

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

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

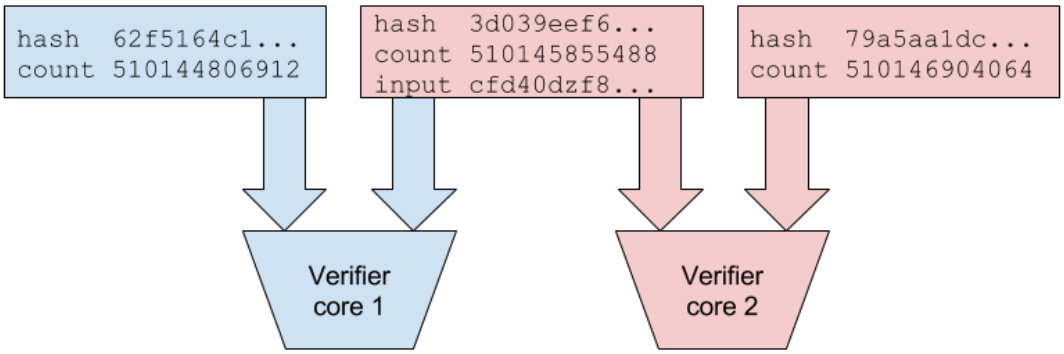


図 4　複数コアによる検証

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。

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。

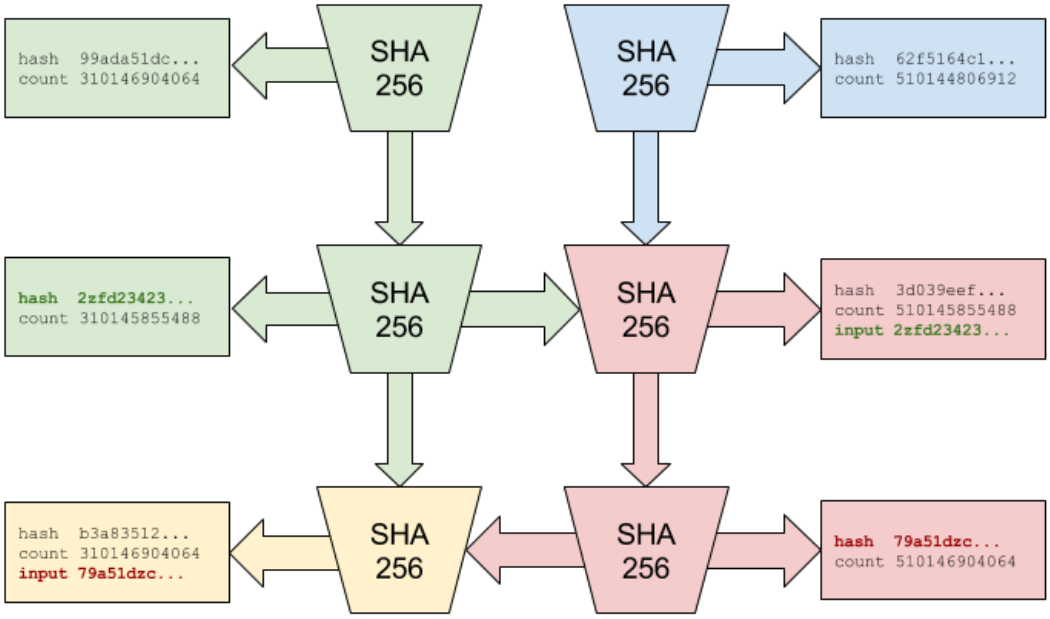


図 5　同期する二つの生成者

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、”ハッシュ値順に並べる”などの事前に取り決めた方法で順序付ける。

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、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。

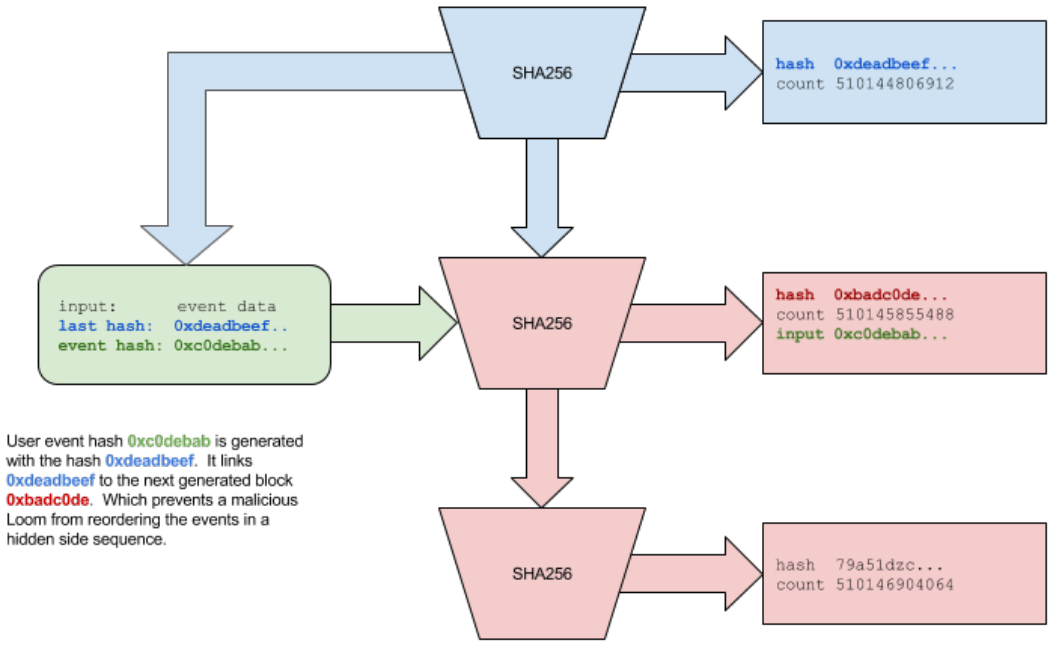


図 6　入力受付時にハッシュの存在性を検証

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が正規のシーケンスに追いつく時は永遠に来ない。

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 において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 票による投票のみで否定票はない。

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、所定のトランザクションに投票する権利を得る。ロックアップの実施も他のトランザクションと同様 PoH のストリームに挿入されるトランザクションとして扱われる。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トランザクション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トランザクションや投票は可能だが、$\frac{2}{3}$ 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トランザクション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 生成者を使って投票内容をねじ込めてしまうという弱点を指す。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るならば、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。つまり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　ストリーミング形式の Proof of Replication

6.1　概要説明

Filecoin は Proof of Replication の一つの設計を示した [6]。ここで説明する設計は Proof of Replication の高速な承認をストリーミング形式で提供することを目的としている。これは Proof of History のシーケンスにより時間の追跡が可能になったことの恩恵である。複製そのものは合意形成アルゴリズムに使われることはないが、ブロックチェーンの履歴や状態の保管にかかったコストを説明するのに役立つツールである。

6.2　アルゴリズム

図 7 に示す通り、CBC モードはブロックを冒頭から順番に暗号化していく。前回ブロックの暗号文と現在ブロックの平文の XOR を暗号化し、現在ブロックの暗号文を計算する。

各データ複製ノードは PoH シーケンスのハッシュ値に署名することで固有の鍵を生成する。これはそれぞれの鍵と複製先ノード、PoH シーケンスを一意に組み合わせるので、特定の PoH シーケンスから発行されたハッシュ値を一意に選択できる。(ハッシュ値の選

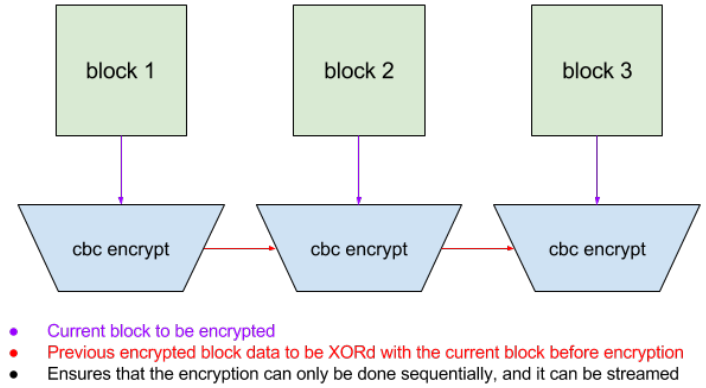


図 7　シーケンシャルな CBC モード暗号化

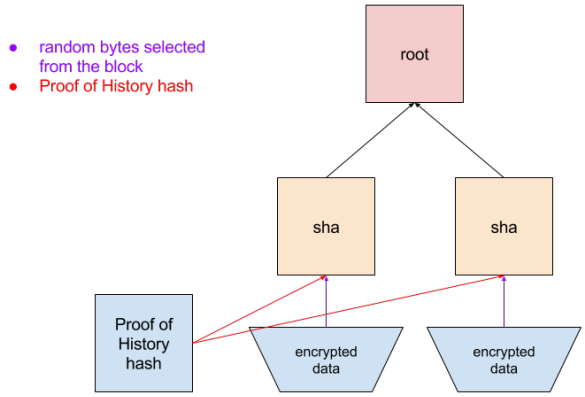


図 8　高速な Proof of Replication

択についてはセクション 6.5 を参照）

すべてのデータセットはブロック毎に暗号化される。この鍵は各ブロックから 32 バイト分のサンプルを無作為に選び出すための擬似乱数のシードとして、PoRep の証明に利用される。

シーケンスから流れてきた PoH ハッシュ値を CBC モードで暗号化した各ブロックの冒頭に繋げた値からマークルハッシュ値は計算される。

ルートハッシュは鍵と一緒に PoH ハッシュ値から発行される。データ複製ノードは PoH ハッシュ値が発行されたとき、その時点から N 回のハッシュ計算が行われるまでに証明を行わなくてはならない。ここで登場する N の値は、暗号化にかかる時間の約半

分に相当するハッシュ計算回数を指す。PoH 生成者は事前に決められた期間に Proof of Replication 用のハッシュ値を複数発行する。このときデータ複製ノードは証明に使うハッシュ値を選択する必要がある。これを一連の証明プロセスとし、完了後はハッシュ値の署名やルートハッシュ生成用のサンプリングから繰り返されることになる。

一定回数の証明を終えたとき、新しい鍵を使ってデータを CBC モードで改めて暗号化する。

6.3 承認

複数コアを利用して、各複製ノードを承認するための暗号化を並列かつブロックひとつひとつをこなすストリーム形式で計算することができる。必要な空き領域は合計でブロック 2 個分 * コア数 になり、これは各コア上で行われる今回ブロックの暗号化計算に、暗号化済み前回ブロックの値が必要なためである。各コアは暗号化済みブロックが使われる証拠すべての検証に使うことができる。

証拠の承認にかかる時間は暗号化計算にかかる時間におよそ等しい。ハッシュ計算の対象データのサイズを大きく削減するため、各ブロックから一部のバイト列を無作為にサンプリングする。同時に承認可能なデータ複製ノードの数は、利用可能なコアの数に等しい。現代の GPU3500 以上のコアを備えるものの、CPU に比べてクロック速度が $\frac{1}{2}$〜$\frac{1}{3}$ 程度に劣る点に注意したい。

6.4 鍵のローテーション

鍵を切り替えないと同じ複製データから複数の PoH シーケンスに対し、陳腐化した証拠が生成される。よって鍵は周期的にローテーションされ、各複製データはそれぞれに特有の PoH シーケンスに紐付けられる新しい鍵で暗号化することにする。

現代のハードウェア構成では基本的に複製データの承認は GPU 上で行われるで、そのクロック速度に合わせてローテーションの早さも調整する必要がある。

6.5 ハッシュ値の選択

PoH 生成者はネットワーク全体が PoRep に使うハッシュ値を発行する。このハッシュ値は証明計算の高速化に必要なサンプリングの擬似乱数として使われる。

このハッシュ値は暗号化の計算にかかる約半分の時間に相当する周期で発行される。各データ複製ノードはそれぞれ同一ハッシュ値および暗号鍵を使い、またそのハッシュ値に

署名した値をバイト列のサンプリングのシードとして使わなければならない。

各データ複製ノードは暗号化計算にかかるより短い時間内に複製データを保有している証拠を提出しなければならない。さもないとデータ複製ノードは暗号文をストリーム計算した上で、証明する度にデータ本体を削除できてしまう。

悪意のある生成者はハッシュ計算を行う直前に、ハッシュ値の操作を目的とした作為的なデータを挿入しようとするかもしれない。この攻撃への対処については 5.13.2 で議論する。

6.6 証拠の検証

提供された PoRep の証拠を検証する役割は PoH ノードが担うものではない。データ複製ノードから提出された検証中あるいは承認済みの証拠を追跡可能にすることが PoH ノードの役割である。データ複製ノードが提出した証拠に対する大多数の検証者からの署名を受けて、その証拠は正式に承認されたものとみなせる。

承認はデータ複製ノードにより P2P ゴシップネットワークで収集され、ネットワーク上の大多数の検証を含むパケットひとつで提出される。このパケットは PoH シーケンス上で特定ハッシュ値が発行される以前のすべての証拠を承認するもので、複数の複製データの承認を一度にできてしまう。

6.7 攻撃

6.7.1 スパム

高速な承認をすすめるためノードは承認を要求する際に、暗号化済みデータとマークル木全体をネットワークへ提供する仕様になっている。これを利用して、悪意のある利用者は多くのデータ複製ノードを作成し、PoRep に対する承認リクエストでネットワークを埋め尽くそうとするかもしれない。

本文における PoRep は複製された証拠の承認を安価にし、ネットワークの許容量を無駄に設計になっている。複製一件あたり 1 コアを、暗号計算に必要な時間分だけ専有されることになる。複製の最大許容数は利用可能な全コア数と同値にすべきである。現代の GPU では 3500 個以上のコアを備える。

6.7.2　部分消去

データ複製ノードはデータ全部の保管を避けて、一部削除する可能性がある。証拠数に応じた試行回数とシードによる乱数値がこの攻撃を困難にする。

1 テラバイト分のデータを保管するノードが、1 メガバイトあたり 1 バイトのデータを削除するケースを想定する。1 メガバイトあたり 1 バイトをサンプリングする証明を実施すると、削除部分と証拠部分が衝突する確率は $1-(1-1/1,000,0000)^{1,000,000}=0.63$ になる。証明を五回繰り返すだけで既に衝突する確率は 99 %を超えるため、部分削除の検知を避けることは極めて困難であると言える。

6.7.3　PoH 生成者との談合

署名済みハッシュ値はサンプルの決定に使う擬似乱数のシード生成に使われる。もしデータ複製ノードが署名対象となるハッシュ値を事前に知ることができたら、サンプリングされない大部分のデータを削除できてしまう。

PoH 生成者と談合しているデータ複製ノードはサンプリングに使われるハッシュ値が発行される直前のタイミングで特定のトランザクション情報を挿し込む。十分に性能の高いコアが備えられていれば、攻撃者はデータ複製ノードにとって都合の良いハッシュ値を発行することもできる。

この攻撃は単体のデータ複製ノードにしか利益をもたらさない。すべてのノードは ECDSA（あるいは相当の）計算で署名された同一ハッシュ値を使わなくてはならず、署名されたハッシュ値は各ノードに特有のものとなり衝突耐性を備える。結局この攻撃者はその談合ノードから生み出される僅かな利益しか得ることができない。

6.7.4　DoS

データ複製ノードを増やすコストは、ストレージを用意するコストに等しいと考えられる。一方でデータ複製ノードが増えたときに必要となる追加の計算能力は、ノード一つあたり CPU あるいは GPU の 1 コアとなる。

これはネットワーク上でデータ複製ノードを大量に作成することで DoS 攻撃を実行できる可能性を示唆する。

合意プロトコルがデータ複製ノードを指名でき、可用性・帯域・配置などの面で適切な複製データから導かれる証拠のみを評価することで、この攻撃を制限することができる。

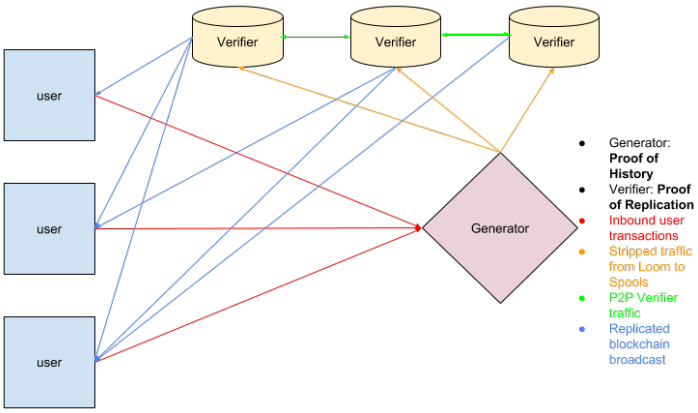


図 9　システムアーキテクチャ

6.7.5　コモンズの悲劇

PoS 承認者は検証を行わずに、PoRep から提供された証拠をひたすら承認する可能性がある。経済的インセンティブは PoS 承認者たちに検証の実施を促す設計にすべきである。ひとつの対策として、PoS 承認者とデータ複製ノードの間で採掘報酬を分け合う仕組みにする。

さらなる対策として、PoRep 承認者が低確率で虚偽の証拠を提出する仕組みが考えられる。PoRep 承認者は虚偽のデータを作ったプロセスを明かすことで、証拠が偽物だったことを証明することができる。PoS 承認者が偽物の証拠を承認した場合、ただちにスラッシングで罰せられる。

7　システムアーキテクチャ

7.1　構成要素

7.1.1　リーダー、Proof of History 生成者

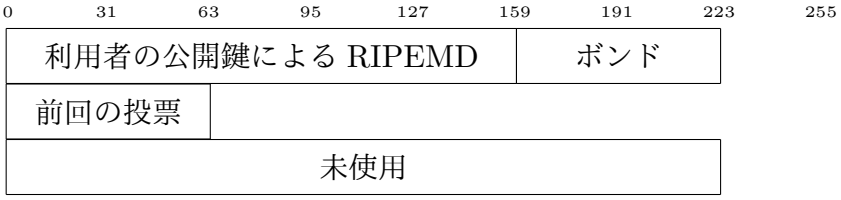
リーダーは投票で選ばれた Proof of History 生成者のことである。不特定多数のユーザから提出された全トランザクション情報に対し、唯一性のある順序付けをした PoH シーケンスを生成する。ある程度の数のトランザクションを挿入し終えると、シーケンスの状態にリーダーは署名を行う。リーダー固有の署名が付与される。

7.1.2　状態

利用者からの入力を含めて作られた純粋なハッシュテーブルである。各ハッシュ値は利用者の入力内容と、計算に必要な記録を含んでいる。

トランザクションテーブルは合計 32 バイト分の以下の情報を含む:

Proof of Stake ボンドのテーブルは合計 64 バイト分の以下の情報を含む:



7.1.3　承認者、状態の複製

承認者ノードが複製を行うことでブロックチェーン状態の高い可用性が実現する。複製対象は合意形成アルゴリズムに従って決まり、検証者はオフチェーンの評価基準で承認された Proof of Replication ノードに対し投票を行う。

PoS 承認者の最低ボンド量、データ複製ノードの必要ボンド量からネットワークの性能も変更される。

7.1.4　検証者

検証者ノードは承認者から流れて来た情報を消化する。検証者は仮想ノードであるため、同一機体上で承認者またはリーダーとして稼働することも、特定の合意形成に参加する離れた機体上で稼働することも可能である。

7.2　ネットワーク帯域の制約

リーダーは利用者から送られてくるパケットを受けて最適な方法で順序付けし、承認者に向けて PoH シーケンスを送出する。効率化はトランザクションのメモリアクセスパ

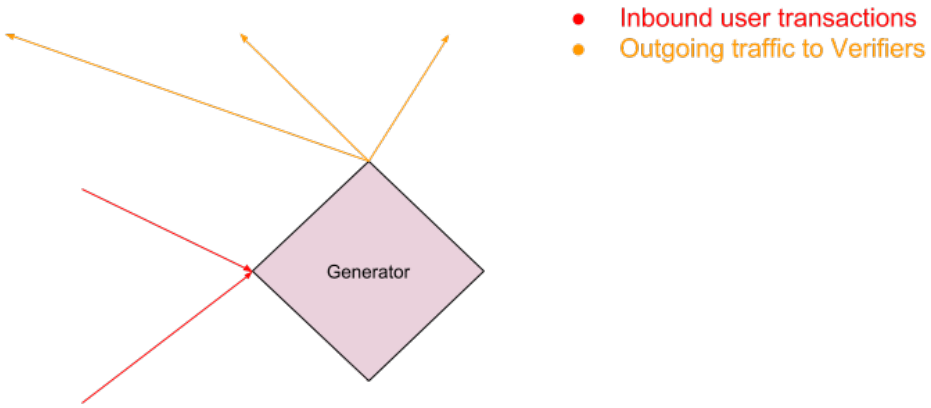


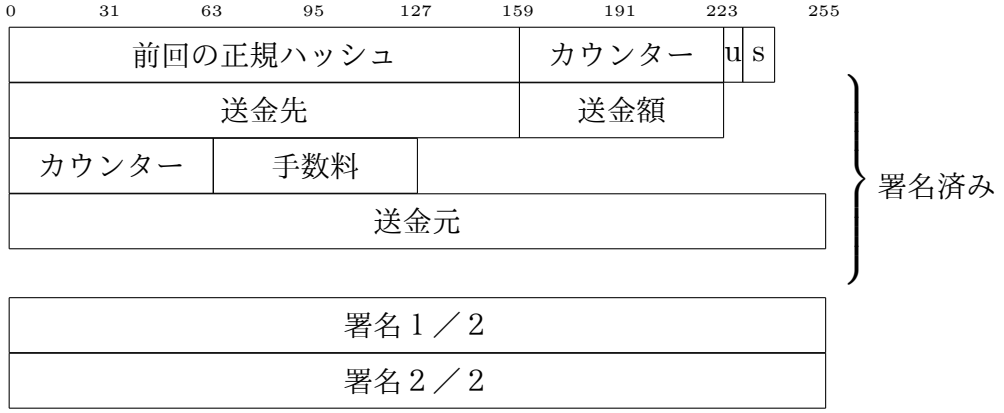
図 10　生成者ネットワークの制約

ターンに基づいてなされる。これは障害の発生を最小化し、データ取得速度を最大化するためである。

利用者から受け取るパケットのフォーマット:

サイズは $20 + 8 + 16 + 8 + 32 + 3232 = 148$ バイト

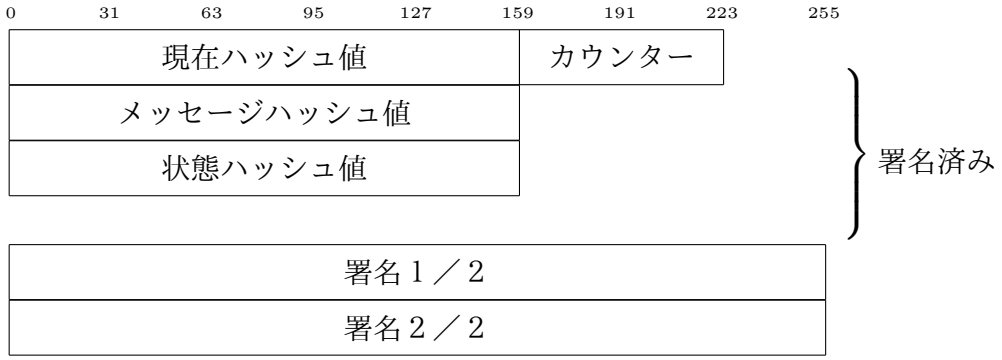
サポート可能な最小ペイロードは 1 送金先アカウントが含まれるものとなる。ペイロードを含むもの:



ペイロードを含む場合の最小サイズ: 176 バイト

Proof of History シーケンスのパケットは現在のハッシュ値、カウンター、PoH シーケンスに追加される全メッセージのハッシュ値、全メッセージ処理後の署名付き状態を含む。このパケットは N 個のメッセージがブロードキャストされる毎に送信される。

Proof of History のパケット:



最小パケットサイズ: 132 bytes

1Gbps のネットワーク上で処理可能な最大トランザクション数は 1 Gbps / 176 bytes = 710k TPS である。ただし Ethernet のフレーム構造により 1 〜 4 %程度のロスが見込まれる。ネットワークの帯域に余裕がある場合、リード・ソロモン符号の付与により可用性を高め、承認者へのデータ送信をストライピングにできる。

7.3 計算能力の制約

各トランザクションは承認情報のダイジェスト値を必要とする。この命令はトランザクションメッセージ分以上のメモリを一切使わず、メッセージ毎に個別で並列計算することができる。つまりスループットはシステム上で利用可能なコア数により上限が決まる。

GPU ベースの ECDSA 承認サーバで実験した結果、毎秒 900k 命令をマークした [9]。

7.4 メモリ容量の制約

各アカウントの 32 バイトのエントリを含むフルハッシュテーブルのうち 50% の状態を保持する実装では、10,000,000,000 アカウントを想定したとき理論上では 640GB のメモリを占めることになる。このテーブルへの定常ランダムアクセスは秒間 $1.1 * 10^7$ の読み書きを計測する。トランザクション毎に二回ずつの読み書きを行う場合、メモリのスループットは 2,750,000TPS となる。この数値は Amazon Web Service の x1.16xlarge インスタンスで実測された。

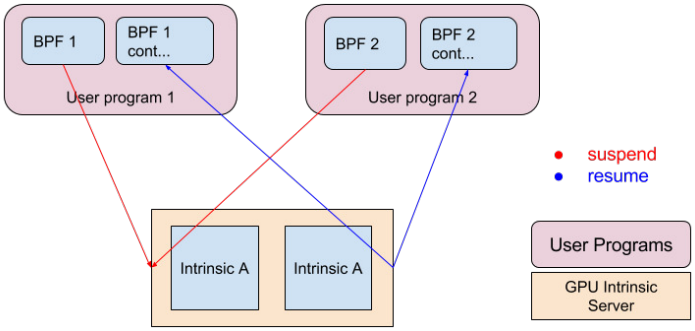


図 11　BPF プログラムの実行

7.5　高パフォーマンスなスマートコントラクト

スマートコントラクトとはトランザクションを一般化したもので、各ノード上で動作し、状態を書き換えるプログラムである。ここでの設計は extended Barkeley Packet Filter（eBPF）の性能および分析等の利便性の高さを利用し、スマートコントラクト言語として JIT バイトコードを活用する。

この設計の主な特長はゼロコストな FFI を実現できる点である。プラットフォームに直接組み込まれた関数はプログラムから呼び出すことが可能である。組み込み関数の呼び出しはプログラムを一時停止させて、パフォーマンスの高いサーバ上でその処理をスケジューリングさせる。処理はまとまった単位にわけられて、GPU 上で並列実行される。

上の例では二つの異なるプログラムが同じ組み込み関数を呼び出している。各プログラムは組み込み関数のバッチ処理が完了するまでに停止される。組み込み関数処理の例としては ECDSA 承認を挙げることができる。GPU 上で実行する処理をバッチにすることで、スループットを数千倍に引き上げることが可能である。

このトランポリンにはネイティブな OS によるスレッド切り替えが必要とされていない。BPF バイトコードは割り当てられたメモリすべてのコンテキストをすべて定義済みだからである。

eBPF のバックエンドは 2015 年より LLVM に組み込まれており、LLVM のフロントエンド言語をスマートコントラクトの記述に用いることができる。2015 年より Linux カーネルにも含まれており、初回の反復開発は 1992 年頃であった。eBPF の正常検査を一息で可能とし、実行系およびメモリへの要求を確認した上で x86 命令に変換することができる。

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