Symbol 解体新書

Japanese Translation with Comments

of

Symbol from NEM

Technical Reference

Version 0.10.0.4

岐阜大学大学院医学系研究科

手塚　建一（目指せ北海道）

Ken-ichi Tezuka, Ph.D. (Hokkaido)

Gifu University Graduate School of Medicine

January 2022

目次

はじめに

2021 年 3 月にメインネットがローンチされた Symbol は、まったく新しい NEM ブロックチェーンです。簡素化され使いやすいスマートコントラクトと呼べるアグリゲートトランザクションなどの新機能を搭載し、前バージョンである NEM (NIS1)の高機能化と高速化を同時に実現しています。その中心となるコアプログラマー、Bloody Rookie, gimre, Jaguar0625 の 3 氏による、Symbol Technical Reference（バージョン 0.10.0.4）を日本語化し、簡素ではありますが理解を助けるための解説を加えました。

本書は NEMLOG（https://nemlog.nem.social/）で、「Symbol 白書」として 2019 年 12 月から 2020 年 8 月まで連載されたものを、NEMTUS 勉強会用の参考資料として編集しなおしました。その後 0.10.0.4 で加筆、変更された部分を追記してあります。私がつけ加えたコメント、感想などは赤字で示しています。知識が及ばなかったり、用語が統一されていなかったり、また業界標準の用語と異なっているところ等、多々あるかと思いますが、ご了解の上でご利用ください。

本書を作製するにあたり、サポートをいただきました NEM コミュニティ、NEMLOG の皆様、NEM/Symbol コア開発者の皆様に、深謝いたします。

2022 年 1 月

岐阜大学大学院医学系研究科　手塚　建一（目指せ北海道）

前文

「打たないシュートは 100%はずれる。」

- ウェイン　グレツキー（ホッケー選手）

NEM は、2014 年 1 月に bitcointalk スレッドへの「参加の呼びかけ」としてひっそりと始まりました。暗号通貨世界は 2013 年の終わりに小さなブームを経験したばかりでした（—それは、数年後の狂乱と広がりとは似ても似つかぬ小さなものでした—）が、その狭い空間には大量の熱意が集まっていました。 NXT は最初の PoS ブロックチェーンの 1 つとしてリリースされたばかりで、初期の NEM コミュニティは、NXT コミュニティに触発され、それとつながりを持っていました。これには、現在残っている 3 人のコア開発者全員が含まれます。

何を構築すべきかについて、初期段階での議論がありましたが、すぐに新しいものをゼロから作ることにしました。スクラッチから始めることで、デザイン上の柔軟性が大きく広がり、高いレベルの標準コード形式を採用することができました。それによってブロックチェーンの地平線に、何か新しいものをもたらせる機会が与えられたのです。多くの努力の結果（—主に夜と週末—）、これは 2015 年 3 月の NIS1 メインネットのリリースで最高潮に達しました。私たちは、自分たちが作り上げたものに喜びを感じていましたが、いくつか重要な点を端折ってきたことを知っていたので、その後も改善を続けました。結果的に、NIS1 バージョンに存在した重要部分のパフォーマンスボトルネックを改善し、将来に向けてのイノベーションを高速化するためには、根本的な再構築がふたたび必要になるという認識に至りました。

TechBureau のサポートに感謝します。TechBureau は、まったく新しいチェーンである「Symbol」の構築をサポートしてくれました。これにより、NIS1 に内在していた問題の多くが修正され、将来の改善と強化のための強固な基盤への希望が生まれました。私たちの使命は、決定論的なファイナライゼーション機能を付け加えた、真の高性能「ブロックチェーン」を作ることでした。これについては成功したと思っています。

その過程は私たちにとって長い旅でしたが、これがわれわれにとってスクラッチから作り上げる最後のブロックチェーンであることを願っています。 みなさんが、Symbol を使用して、これから何を作り、どんな斬新なものが構築されるかが本当に楽しみでワクワクしています。貢

献してくれた方々と、私たちにインスピレーションを与えてくれた多くの人々に、改めて感謝したいと思います...

BloodyRookie gimre Jaguar0625

以下の 2 項目は 10.0.3.0 のアップデートで削除されましたが、後で参照できるよう、ここに追記したままにしておきます。

DAG

Directed Acyclic Graph の略で、ブロックチェーンに代わる新しい承認システムとして注目されている技術です。マイニングによってではなく、個々の取引について、その当事者が承認をする点に特徴があり、ブロックチェーンが持っているスケーラビリティー問題を解決すると考えられています。IOTA などが採用しています。

dBFT

delegated Byzantine Fault Tolerance の略。NEO によって採用されている非中央集権的な承認アルゴリズム。NEO の保有者によって選出された Bookkeeper の中から、ランダムにブロック生成権を持つ 1 人が選出され、その他の Bookkeeper によってその妥当性が検証される。66%以上の賛同が得られれば、そのブロックは承認されてチェーンの一部となります。有権者によって選ばれた選挙人が大統領を選出するけど、その政策を拒否する強い権利も持つみたいな感じですね。

Symbol の白書は、他のブロックチェーンのものと比べると、読んでいてとても面白い上に勉強になるので、解説を加えながら日本語バージョンを作る作業も楽しいものになりました。前文の最初から、実に重みを感じる文章です。DAG や dBFT といった最新のコンセンサスアルゴリズムについても勉強できましたし、ついにブロックチェーンの最大の問題であるファイナライゼーション問題にも切り込んでいます。Symbol は、長い時間をかけて熟成されてきたブロックチェーンアーキテクチャーを持つ、最終・最強プラットフォームを目指しているのですね。

Typographical Conventions

Description	Attributes	Example
Project name	bold, colored	**Symbol**
Classes	fixed-width font	`ProcessBootstrapper`
Fields	fixed-width font	`FeeMultiplier`
File paths	fixed-width font	`commit_step.dat`
Configuration settings (prefixed with configuration filename) in text mode	italic	*network:maxDifficultyBlocks*
Configuration settings (without configuration filename) in equations and formulas	regular	maxDifficultyBlocks
Fields in equations and formulas	regular	T::SignerPublicKey
Function names in equations and formulas	regular	VerifiableDataBuffer()

表 1: 本ドキュメントで使用される字体のリスト（日本語版 Symbol 白書ではこれらを反映していませんので、原文を読む際に参照してください）

1. イントロダクション

「火が灰から蘇るように、光が影からおどり出るように。折れた剣から新しい刃が生み出され、王冠を持たぬ者から王が生まれる。」

- J. R. R. トールキン

トラストレス、高性能、層構造、ブロックチェーン様 DLT*プロトコール - これらが、Symbol 開発に影響を与えた基本原理である。DAG や dBFT などいくつかの DLT プロトコールも考慮はされたが、すぐにブロックチェーンプロトコールのほうが、より本物のトラストレス性を持つものとして選択された。ブロックチェーンならば、どのノードもチェーンの完全なコピーをダウンロードすることができ、いつでも独立にそれを検証できるのである。十分なハーベスト性能を持つノードなら、誰の指示も受けることなく、いつでも新しいブロックを作り出すことができる。この選択は、他のプロトコールと比べてスループット性能には劣る。しかし、ビットコイン思想 [Nak09]に、より強固に立脚しているように思えるのである。

Symbol は、確率論的、決定論的ファイナライゼーションの双方をサポートする。確率論的ファイナライゼーションのもとでは、特定のブロックが巻き戻される確率が、より多くのブロックが積み重なる - あるいは繋がれる – のにともなって減少する。この確率は、極めて小さくなるかもしれないが、ゼロになることはない（脚注；ブロックチェーンの効率を上げるために、確率論的ファイナライゼーションを採用した場合、network:maxRollbackBlocks によって定められたブロック数まで巻き戻しがおこなわれる。それよりも古いブロックは、巻き戻される確率が極めて低いという意味でファイナライズされたとみなされるが、保証はされない）。一方、決定論的なファイナライゼーションにおいては、絶対に巻き戻されないチェックポイントが形成される。これによって、（修正のために）より深いロールバックが起きることはあるが、より決定的なファイナライズが実行されることになる。どちらのケースでも、ブロックのロールバック（量）とトランザクションの無効化（=検証の頻度）は、ユーザーが決めることができる。

トラストレス性の部分について注目してみると、NEM には以下のような技術が追加されている：

- ブロックヘッダーがトランザクションデータを利用すること無しに同期される。しかし、ブロックチェーンの完全性は損なわれない。
- トランザクションのマークルツリーがブロックに内包される暗号通貨的なトランザクション（空であっても）の検証を可能にする。
- 受領証（レシート、Receipts）が、遠隔地で開始された（ブロックチェーンの）状態変化の透明性を向上させる。
- 状態証明（State proofs）が、ブロックチェーンの特定の状態に対するトラストレスな検証を可能にする。

Symbol では、ひとつのサーバープログラムが、様々なプラグイン（トランザクション補助）や機能拡張（機能性）を読み込むことによってカスタマイズされる方式を取っている。その結果、おもに 3 種類のコンフィギュレーションが、ネットワークごとに設定される。加えて、これらの中間型のコンフィギュレーションも、特定の機能拡張を許可・停止することによって実現できる。

3 種類の主要コンフィギュレーションとは：

1. Peer：ピアノード。これらはネットワークを構成して新しいブロックを作るために働く。
2. API：API ノード。これらは、MongoDB データベースに情報を保存して、NodeJS REST サーバーとともにネットワークからの問い合わせに迅速に応答する。
3. Dual：デュアルノード。これらはピアノードと API ノードの両方の機能を実行する。

強固なネットワーク構築には、多数のピアノードに加えて、API ノードが十分な数存在することが大切である。ネットワークに対する負荷や要求に応じて、ノードの構成割合を動的に変化できるようにしたのは、より効率の良い最適化されたネットワークが形成されることを期待してのことである。

コアブロックとトランザクションパイプラインを disruptor**パターン上に保持したものを基礎に、- つまり、できる限り並列処理することによって - 他の典型的なブロックチェーンよりも高い TPS (transaction per second）を達成している。

NIS1 は、ブロックチェーンの歴史への良いエントリーポイントであったが、Symbol はさらに高みを目指した革命的な価値を持つ。これは完成形という意味ではなく、ここから始まるのである。やるべきことは多い。

*; DLT, Distributed Ledger Technology のこと。分散型台帳技術。一冊の台帳に全てを記載してそこにアクセスするのではなく、複数の台帳によって情報を分散共有することにより、利便性や安全性を高める、ブロックチェーンの根本思想のひとつ。
**; disruptor, JAVA プログラミングにおける、並列データ処理構造のひとつ。リングバッファー（https://wa3.i-3-i.info/word14292.html）を使って、高速で安定したデータのバッファリングをする技術。

さて、イントロダクションからいきなり難しいですね。使用する言語を Java から C++にすることで高速化しただけかと思っていたら、データ処理そのものをかなり頑張って分散・並列化しているようです。

1.1 ネットワークフィンガープリント

個々の独立したネットワーク（＝ブロックチェーン）は、個別のフィンガープリントを持ち、その中身は以下のとおりである：

1. ネットワーク ID - 1 バイトからなる ID であり、ネットワーク全体で共有される。ネットワーク内のブロックチェーンで使用されるアドレスは、すべてこのバイトから始まる。
2. ジェネレーションハッシュシード - ランダムな 32 バイトの数値であり、すべてのネットワークがそれぞれ独自のものを持っている。ハッシュ化や署名化される前に、トランザクションデータ（つまりネメシスブロック）の最初に付与され、ネットワーク間の混線を利用した攻撃を防ぐ。

0.9.6.3 でアドレス導出のハッシュ関数が SHA3 に統一され、以前の Keccak とは互換性がなくなりました。また、各ブロックに付与されるジェネレーションハッシュとの混同を避けるため、最初のネメシスブロックにつけられるジェネレーションハッシュシードが定義されました。

Web や一般的なライブラリで提供される SHA3 を使って現在の NIS1 のパブリックキーからアドレスを生成しようとすると、違うものが導出されることを、以前 NEMLOG 記事（https://nemlog.nem.social/blog/7666）で紹介しました。NIS1 では、SHA3 のオリジナル方式である Keccak が、ハッシュ化アルゴリズムとして使われていたためです。今回、デフォルトを一般的な改変型 SHA3（世間一般で SHA3 と呼ばれているもの）とすることで、他のブロックチェーンとの親和性を高める意図があると思われます。今回、広く用いられている SHA3 の安全性が十分に検証されたと判断し採用したのでしょう。

2. システム

「新しいテクノロジーは社会を変革する。だが、未成熟な状態にあるテクノロジーの真の価値を見抜ける人はほとんどいない」

- 劉慈欣（中国の SF 作家）

Symbol はネットワーク全体レベルでも、ノード単位でも高度なカスタマイズが可能である。ネットワークレベルの設定は network にあり、すべてのノードで一致していないとならない。一方、ノードに関した設定は、ノードごとに異なっても構わず、それは node 設定に記述される。

Symbol は、プラグイン/機能拡張方式で設計されており、チューリング完全なスマートコントラクトをサポートしていない。スマートコントラクトはユーザーに自由度を与える代わりに、ユーザーエラーを頻発させるからである。プラグインモデルは、スマートコントラクトモデルに比べれば、ブロックチェーンが提供できる機能を制限するが、その分脆弱性を下げることができる。さらには、付け加える機能を無制限にせず、限定されたものにすることによって、パフォーマンスの最適化が容易になる。こうして、Symbol は設計当時に期待された通りの高いスループットを実現できた。

2.1 トランザクションプラグイン

ネットワークを形成するすべてのノードは、同じタイプのトランザクションおよびプロセスを正確にサポートしなければならない。それによって、ブロックチェーンの状態がすべてのノードで共有できる。ネットワークがサポートするトランザクションの形式は、トランザクションプラグイン（transaction plugins）で定義され、すべてのノードによってロードされる。そしてプラグインセットは、コンフィグ内の network:plugin によって定義される。プラグインセットに対するすべての変更や追加、削除は、すべてのネットワークノード間で同期される必要がある。ネットワークノードの一部だけが変更を受け入れた場合は、そのグループだけがフォークした状態になるだろう。Symbol に組み込まれたすべてのプラグインは、拡張性を保証するために、このプラグイン形式にのっとっている。

プラグインは、動的にリンクされた独立のライブラリーで、下記のようなエントリーポイントを持つ（脚注；プラグインのフォーマットは OS やコンパイラの種類に依存する。そのため、

プラグインは、同じバージョンとオプションのコンパイラによってビルドされる必要がある）。

```
extern "C" PLUGIN_API
void RegisterSubsystem(
        catapult::plugins::PluginManager& manager);
```

PluginManager は、コンフィグに含まれるいくつかのパラメータにアクセスし、プラグインを初期化する。この PluginManager クラスを通じて後述のリストの中から必要なものを登録する（もちろん登録しない場合もあり得る）。

1. トランザクション - 新規のトランザクションタイプやその処理方法のマッピングについて定義するもの。つまり、このプラグインはトランザクションを解釈して、その後の処理につなげる通知セットを作成する。様々な処理制限についてもここで定義できる。例えば、アグリゲートトランザクションで処理すべきかどうかを指定したりする（6.2: アグリゲートトランザクション参照）。
2. キャッシュ - シリアル化やその逆処理（バイナリー化とそのデコードも含む）が必要なモデルタイプのために新しいキャッシュタイプを定義できる。トランザクション状況に対応したキャッシュは、今後ブロックの状態ハッシュ（StateHash, 7.3: 状態ハッシュ参照）生成にも含むことができるようになる予定である。
3. ハンドラ - いつでも利用できる API。
4. 診断子（Diagnostics） - ノードが診断モードで動いている時だけアクセスできる API とカウンター。
5. バリデータ - ブロックやトランザクションの処理によって生成される通知情報（notification）に対し、付帯情報を持つあるいは持たない検証情報を返す。登録されたバリデータは、組み込み、あるいはプラグインによって定義された通知情報に対して、不適切な内容や状況変化を拒絶することができる。
6. オブザーバー - オブザーバーはブロックやトランザクション処理によって生じた診断情報を処理する。登録されたオブザーバーは、組み込みあるいはプラグイン定義の通知情報を処理して、ブロックチェーン状態をアップデートする。オブザーバーは、バリデータで処理された情報のみを取り扱うため、検証機能を持つ必要がない。

7. リゾルバー - 非翻訳から翻訳へのマッピングを定義できる。たとえば、ネームスペースプラグインはエイリアスの翻訳をサポートする。

今回から、システムの定義に入っていきます。ここでは用語の説明と定義がなされています。Symbol は並列性を高めるために、リングバッファーキャッシュを持ちますが、情報を正確に処理するために、様々な診断や検証システムが導入されていることが伺われます。

2.2 Symbol 機能拡張

ネットワークを形成する個別ノードは、それぞれが違った機能を提供することができる。たとえば、データベースに情報を保存する機能や、イベントをメッセージに乗せて送り出す機能を持つノードなどである。このような個別の機能は、コンセンサス形成に影響を与えない限り、すべてオプションとして提供される。これらは、機能拡張（Extensions)のセットとして定義され、ノードごとに extensions-{process}:extensions という形で設定される。Symbol のビルトイン機能のほとんどが、この機能拡張モデルに則っており、拡張性を担保している。

機能拡張は、それぞれが独立した、動的にリンクしたライブラリーのことであり、下記のようなエントリーポイントを持つ（脚注；機能拡張のフォーマットは、OS やコンパイラによって異なる。そのため、すべてのホストアプリケーションとプラグインは同じコンパイラバージョンとオプションでビルドされなければならない）：

```
extern "C" PLUGIN_API
void RegisterExtension(
        catapult::extensions::ProcessBootstrapper& bootstrapper);
```

ProcessBootstrapper を通してカタパルトの設定やサービス全体にアクセスでき、これらの初期化をおこなう。このスタイルによって、プラグインと比較して、カタパルトの機能拡張はより強力なものとなる。このクラスを通して、機能拡張が実行できるものの例のは以下のとおり：

1. サービス - サービスは独立して実行される。あるオブジェクト情報を受け取ったサービスは、それをもとに様々な処理を行うことができる。また、その他にも診断用のカウンタを加算したり、API（検証可能なものと不可能なもの）を定義して、タスクをスケジューラーに入れることもできる。他の機能に依存したサービスならば、ホストの

実行状況に応じて、自分自身の実行寿命を設定することも可能である。サービスが実行可能な機能にはほとんど制限がなく、大幅なカスタム化が可能になっている。

2. サブスクリプション - 機能拡張はあらゆるブロックチェーンイベントに接続（サブスクライブ）することができる。イベントは、ブロックチェーンに何か変化が起きた時に生じる。ブロック、状態、未承認トランザクション、未完成トランザクション変更といったイベントがサポートされる。例えば、トランザクション状態イベントは、トランザクション処理が完了した時などに発生する。ノードイベントは、新しいノードが発見されたときなどに発生する。

これらに加えて、機能拡張はもっと込み入った方法で、ノードを個別に設定できる。例えば、カスタムされたネットワーク時間を供給するノードなどである。実際、16: 時刻同期に記述されているアルゴリズムに従って、ネットワーク時間を供給するための機能拡張がある。これは、高レベルな機能拡張の一例でしかない。機能拡張によって可能になる事柄については、プロジェクトコードや開発者ドキュメントを参照（脚注；詳細については、https://nemtech.github.io/を参照すること）。

さすがオブジェクト指向言語の C++といった感じで、いろいろなオブジェクトが定義されていきます。ネットワークに関する機能がプラグイン形式で、ノードごとのカスタマイズはプログラムが読み込む機能拡張という形で追加されるため、設計の自由度がかなり高くなると思われます。Symbol のコードを使った dHealth ネットワークでは、プラグインを使って Strava というヘルスケアサービスと連携しています。また、本家の Symbol でもプラグインによるアップデートによって、モザイク撤回（revoke）の機能が付け加わったりして、ネットワーク全体に対して、非常に柔軟なカスタマイズや改良ができるようになっています。

2.3 サーバー

Symbol の最小構成は、シングルサーバーである。ネットワークに必要なトランザクションプラグインと、ノード運用者によって選択された Symbol 機能拡張がロードされ、サーバーを初期化する。

Symbol は data ディレクトリにすべての情報を保存する。data ディレクトリの構成は以下の通りである：

1. ブロックバージョンディレクトリ群 - これらの複数のディレクトリは、予め決められたフォーマットによりブロック情報を保存する。承認を受けたブロックのバイナリーデータ、トランザクションデータ、それらに関連するデータが、.dat と .stmt 拡張子がついた形で、これらのディレクトリに保存される。個々のブロック生成の際に作られたステートメント（7.2: レシートを参照）も、アクセスを高速化するためにここに保存される。たとえば、最初のバージョンディレクトリーは「00000」という名前を付けられて、そこに最初のグループのブロックが格納される。hashes.dat ファイルは、ブロックハッシュ値へブロック高をひもづけるデータを保管している。
   決定論的ファイナライゼーションが使われる場合には、これらのディレクトリはバージョン付きの.proof ファイルを内包し、それぞれのファイナライゼーションに対する証明情報を記録する。
2. audit - Audit（監査）ファイルは、audit コンシューマー（9.1.1: 共通コンシューマーを参照）によって作られ、このディレクトリに保存される。
3. インポータンス – インポータンスを保有するアカウントの情報を記録したバージョン付きファイルは、インポータンス再計算ごとに書換えられる。このディレクトリにあるファイルを使えば、どの再計算タイミングにおいても、すべてのアカウントのインポータンスを知ることが可能である。この機能によって、多数のインポータンス再計算を遡る必要がある深いロールバックも可能になる。このディレクトリは、決定論的（Voting が有効化された）ファイナライゼーションが指定された場合にのみ作られる。
4. logs - ログファイル保存オプションが選択されている場合に、バージョン付きログファイルが保存されるディレクトリ。現在実行中のプロセスのログが、このデータディレクトリに直接書き込まれる。ログファイルには、出力したソースプロセスの名前が先頭に付けられる。
5. spool - サブスクリプション通知は、このディレクトリに書き込まれる。これらは、サーバーからブローカーにメッセージを送り出す際のキューとして利用される。また、予期しないサーバーダウンからの復旧プロセスにおいても利用される。
6. state - Symbol は、基本的な保管情報（proprietary storage）ファイルをこのディレクトリに保存する。supplemental.dat と、_summary.dat で終わるファイルは、データの要約を保存する。Cache.dat で終わるファイルは、完全なキャッシュデータを保持する。

7. statedb - user:enableCacheDatabaseStorage フラグがセットされたとき、このディレクトリに RocksDB ファイルが保存される。
8. transfer_message - user:enableDelegatedHarvestersAutoDetection フラグがセットされているときに、このディレクトリに該当するノードのデリゲートハーベストのリクエスト情報が保存される。
9. commit_step.dat - 最新のコミットプロセスのステップが保存される。主にリカバリー用である。
10. index.dat - ディスクに保存されているブロックの数を数えるカウンター。
11. proof.index.dat – ディスクに保存されたファイナライゼーション証明の数を記録するためのカウンター。
12. voting_status.dat – ノードによって送信された、直近のファイナライゼーションメッセージ情報を記録する。

色々と、内部で使われている変数名やファイル名が定義されています。高速化のために、一部要約されたデータが保存されたりしていることが伺えます。だんだん専門的になってきました。また、ファイナライゼーションのためのいくつかのファイルやディレクトリが追加されています。

2.3.1 キャッシュデータベース

サーバーは、キャッシュデータベース有りでも無しでも動作する。
node:enableCacheDatabaseStorage が有効化された場合は、RocksDB がキャッシュデータを保存するために用意される。検証可能状態（7.3: 状態ハッシュ参照）でのトランザクション処理には、キャッシュデータベースが必要であり、ほとんどのケースで有効化される。

キャッシュデータベースが無効化されるケースは下記の通り：

1. 高い秒間トランザクションレートが必要な場合
2. キャッシュの分散検証が重要でない場合
3. サーバーに十分量の RAM が搭載されている場合

この場合は、すべてのキャッシュ情報がメモリーに残される。シャットダウンされる時には、キャッシュデータは複数のファイルに分割されてディスクに保存される。次にサーバーが起動した時に、これらのファイル内容をメモリーにキャッシュデータとして復元する。

キャッシュデータベースが有効化されている時には、キャッシュデータは要約されてから、複数のファイルとしてディスクに保存される。この要約されたデータは、キャッシュに保存されたデータから構築される。たとえば、network:minHarvesterBalance 以上の資産を持つアカウントのリストなどである。このようなリストは、アカウントステートキャッシュに保存されているすべてのアカウントを読み込んで作られるかたわら、適宜ディスクに保存されることによって、メモリーの効率的な利用が可能になる。

RocksDB というのは、キャッシュデータを整理して保管するデータベースのようですね。ハーベスティングアカウントのリストなどが保存され、メモリーの有効活用にひと役かっているようです。メモリーが潤沢にあり、トランザクションレートを限界まで高めたい時には、無効化もできるようです。プライベートチェーンで、大量のトランザクションを処理したい場合などを想定しているのでしょう。

2.4 ブローカー

ブローカープロセスは、並列ネットワークのパフォーマンスに影響を与えることなく、Symbol に、より複雑な機能を加えるために存在する。ネットワーク（の設定）のトランザクションプラグインと、ノード運用者によって付け加えられた Symbol 機能拡張は、ブローカーによってロードされ初期化される。ブローカーは、トランザクションプラグインのすべての機能をサポートするが、Symbol 機能拡張については、その一部しかサポートしない。たとえば、ネットワーク時刻提供を主目的としたブローカー用の機能拡張は、サポートされない。ブローカー機能拡張は、サブスクライバー（Subscriber）を受け入れて、それらに必要なデータを転送することが主目的である。なので、サーバーとブローカーは異なった機能拡張をロードすると考えてほしい。詳細は、プロジェクトコードや開発者ドキュメントを参照。

ブローカーは spool ディレクトリを監視して、変更を見つけると、機能拡張によって登録されたサブスクライバーに、イベントを通知する。機能拡張は、登録されたサブスクライバーに対してイベントを処理するよう司令を出す。たとえば、データベース機能拡張は、イベントを受

け取るとブロックチェーン全体の状況を正確に反映するように、データベースをアップデートする。

spool ディレクトリは、一方通行のメッセージキューとして機能している。サーバーはメッセージを書き込み、それをブローカーが読み取る。逆に、ブローカーが spool にメッセージを書き込むことはできない。この仕様は、ブロックチェーンのパフォーマンスを低下させないためである。

サーバーは、ブロックチェーンデータにロックをかける時、ブロックチェーン同期コンシューマー（consumer, 9.1.2: その他のブロックコンシューマーを参照）に向けて、サブスクリプションイベントを発行する。サーバーによってブロックチェーンがロックされた時、データベース処理に時間がかかりネットワークを遅延させるのを防ぐため、ブローカーに処理を任せるのである。このしくみによって、サーバーのオーバーヘッドは最小に抑えられる。なぜなら、ブローカーによって使われるデータのほとんどは、突然のサーバーダウンから回復する場合にも必要になるものだからである。

ここで、ブローカーについての解説がされています。ブロックチェーンをつなぐためには、一時的にロックするわけですが、その際にデータベースをサーバーのメインルーチンが処理するのではなく、さまざまな問い合わせ業務としてブローカーに任せることで、パフォーマンスを上げ、サーバーが落ちた場合のリカバリーも楽になるということなのでしょう。このサブスクライブシステムは、ウォレットなどを設計する場合にも、便利に使えます。

2.5 リカバリー

リカバリー（復旧）プロセスは、予期しないサーバーやブローカーのシャットダウン時に、ブロックチェーン全体の状態を修復する。ネットワークのトランザクションプラグインや、ノード管理者に必要な Symbol 機能拡張が再ロードされて、リカバリープロセスによって初期化される。ブローカーが使われる場合は、同じ機能拡張がロードされていなければならない。

特定のリカバリープロセスは、コンフィグや commit_step.dat ファイルの内容に依存する。一般的には、状態（ステート）変化がディスクに書き込まれた後にサーバーがシャットダウンした場合には、その状態変化は有効となる。ブロックチェーンの状態は、シャットダウン直前の状

態変化を取り込んだ形で修復される。一方、状態変化がディスクに書き込まれる前に起きた場合には、それらは破棄されることになる。その場合は、ブロックチェーンの状態は、これらの状態変化が無かったものとして初期化される。

リカバリープロセスが完了した後は、ブロックチェーン状態はノードが不正終了しなかったのと一致するように修復をされるべきである。spool ディレクトリも、修復され復帰する。ディスクに書き込まれたブロックとキャッシュは、調停された後アップデートされる。ペンディングされていた状態変化は、可能な場合のみ取り込まれる。その他の不正終了によって影響を受けたと思われるファイル類は、アップデートされるか、または消去される。

テスト期間中に、Symbol のテストノードが約 1 時間以上に渡ってアップデートされないロック状態になったことがあります。この時、たまたま自分の API ノードにトランザクションを投げていたのですが、まったく承認されなくなりました。しかし、トラブルが去ったと同時にキャッシュされていたトランザクションが何事も無かったようにブロードキャストされ、新しいブロックに書き込まれました。詳細は分かりませんが、ブロックチェーンが長時間ロックした状態からも、ちゃんと復帰できるということだと思います。

2.6 共通のトポロジー

ブロックチェーンを構成するネットワークのトポロジー（構造体）は、様々な形態を取ることができ、また異なるトポロジーを持つサブネットワークの集合体となり得る。しかし、ほとんどのノードは主に 3 種類のトポロジーに分類されると思われる。Peer, Api, Dual である。これら 3 つのトポロジー形態では、まったく同じサーバープロセスが使われる。異なるのは、ノードがそれぞれロードする機能拡張の種類である。

Peer ノードは、軽量ノードである。これらは、ブロックチェーンにセキュリティーを提供するための最低限の機能を持つが、それ以上の機能は持っていない。他のノードと同期して、新しいブロックをハーベストするのみである（軽量ではありますが、ブロックチェーン本体を格納するというもっとも重要な機能を Peer ノードが持っています）。

Api ノードは、もっと重量級のノードである。これらは、他のノードと同期するが、新しいブロックをハーベストすることはできない。結合したアグリゲートトランザクションを処理する

ことができ、それを完了させるための共同署名を集める。MongoDB にデータを記録し、サブスクライバーにパブリックメッセージキューを通じて変更をアナウンスするブローカープロセスを持つ。REST Api はこれらの機能に依存しており、遅延を最小限にしてパフォーマンスを最適化する理由から、Api ノードに含まれる。

Dual ノードは、単純に Peer ノードと Api ノードを組み合わせたものである。2 種類のノードのすべての機能をサポートする。Api ノードの機能をサポートすることから、ブローカーを必要とする。

Peer ノードと Api ノード、およびデュアル（Dual）ノードについての簡単な解説です。これらは、ロードする機能拡張の種類で分類され、パフォーマンス要求は Peer ノードが一番軽く、次が Api ノードで、最もヘビーなのが Dual ノードです。API ノードが単独で運用されることはあまりなく、Dual ノードとすることが多いようです。現在では、これに Voting ノードが加わっていますが、300 万 XYM という大量のステークを必要とします。

3. 暗号化

「公開鍵暗号の重要さは理解していたつもりだが、私の予想をはるかに上回って物事は変化した。まさか、先進コミュニケーションテクノロジーの主役になってしまうなんて。」

- ホイットフィールド・ディフィー（アメリカの暗号学者）

ブロックチェーン技術は、暗号技術に基礎を置いている。Symbol は、楕円曲線暗号（ECC）を利用する。どのような楕円曲線を使用するかによって、機密性とスピードが決まる。

Symbol は、次の式で表される Ed25519 デジタル署名アルゴリズムを使用する。本アルゴリズムは次の式であらわされるねじれエドワーズ曲線である：

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$$

これは、2^{255} - 19 という素数で定義される有限の数値領域に展開される。グループ G に対応するベースポイントは B と呼ばれる。このグループは、q = 2^{252} + 27742317777372353535851937790883648493 個の要素から成る。これは、D. J. Bernstein らによって考え出され、最も安全で高速なデジタル署名アルゴリズムである[Ber+11]。

Symbol では、このアルゴリズムによる 64 バイトの短い署名を使い、検証にかかる時間を短縮している点が重要である。しかし、ブロック生成時には、鍵を生成したり署名をするという事はしておらず、この処理のスピードはその意味では重要ではない。

巨大素数を使った剰余計算です。Symbol を使う上で、この理屈を理解する必要はなく、現在の計算機では簡単には破れない暗号システムを使っているんだなということだけ分かれば良いと思います。この暗号アルゴリズムは、アカウントを作ったりトランザクションを発行するときに利用されるのみですので、ブロックの中身を検証してチェーンにつなぐ時にはあまり使われていなのですね。

3.1 公開鍵と秘密鍵のペア

秘密鍵は任意の 256-bit の整数 k（十進数で 77 桁くらい。宇宙に存在する陽子の数に近いらしい）である。これから公開鍵 A を導き出すためには、下記のような計算をする。

$$H(k) = (h_0, h_1, ..., h_{511}) \tag{1}$$
$$a = 2^{254} + \sum_{3 \leq i \leq 253} 2^i h_i \tag{2}$$
$$A = aB \tag{3}$$

これによって A は群の要素（前の節で出てきた整数のグループのひとつ）になるので、256-bit の整数 A に変換して公開鍵として利用できる。

これだけでわかる人は、もうわかっている人です。楕円曲線というのは、2 次元の曲線上の点の集合体ですから、(x, y)の形になります。B というのはその中の 1 点を表していて、それを a 倍するというのは、点 B を a 回移動させるという意味になります（かなり端折ってます）。そして、秘密鍵から作った a という整数によって一意に決まります。つまり、秘密鍵と公開鍵は、ある特別なルールに従って 1 対 1 に対応します（同じ公開鍵が違う秘密鍵で作られることはない）。そして、秘密鍵から公開鍵を作るのは簡単だけれども、公開鍵から秘密鍵を見つけ出すのはとても大変です。

3.2 署名と検証

M というメッセージ、秘密鍵 k、公開鍵 A が与えられた時、下記の手順で署名を行うことができる：

$$H(k) = (h_0, h_1, ..., h_{511}) \tag{4}$$
$$r = H(h_{256}, ..., h_{511}, M) \text{ where the comma means concatenation} \tag{5}$$
$$R = rB \tag{6}$$
$$S = (r + H(R, A, M)a) \bmod q \tag{7}$$

まずは、署名の作り方を順番に見ていきましょう。

(4)は、秘密鍵のハッシュを取り 2 進数に変換（秘密鍵をそのまま使わずハッシュを使用する）

(5)は、(4)の半分の 256 ビットをメッセージの頭にくっつけて、さらにハッシュを取り r を求める。

(6)は、r の数だけ楕円曲線の上で原点 B を移動させる（r は秘密鍵を知らないと作れないため、R は署名者だけが作れる数値（座標）になる）
(7)は、まず、R, A, M を並べてハッシュを計算し、前回式(2)で作った秘密鍵のハッシュの残り 256 ビット a をかけ合わせる。さらに r を足して素数 q の剰余を取る

(R と S)はメッセージ M の、秘密鍵 k による署名である（つまり R と S が検証のためにメッセージ M、公開鍵 A とともに公開される）。S < q かつ S > 0 でなければ、署名の混乱が起きるので、それ以外は無効である（q は楕円曲線を作る時に出てきた大きな素数で、群に含まれる要素数を表します。それを超えたり、負の値になってはなりません）。

(R と S)が送信者によってメッセージ M と公開鍵 A から作られたことを検証するためには、まず S < q かつ S > 0 であることをチェックしたあとに、以下の計算をする。

$$\tilde{R} = SB - H(R, A, M)A$$

そして、

$$\tilde{R} = R \qquad (8)$$

であることを検証すればよい。S が式(7)によって計算されていれば、

$$SB = rB + (H(R, A, M)a)B = R + H(R, A, M)A$$

となり、(8)が成り立つ。

ここは、前回に出てきた式も使って地道に変形していきます。式(7)に B をかけ合わせて SB を求めると、

$$SB = rB + (H(R, A, M)a)B$$

と書けます。(6)から rB = R、前回の式(3)から aB = A ですから、

SB = R + H(R, A, M)aB = R + H(R, A, M)A

です。つまり(8)の上の式の右辺は、

SB - H(R, A, M)A = R + H(R, A, M)A - H(R, A, M)A = R

となり、公開された署名 R と等しくなります。

こうやって書くと、狐につままれたような感じがしますね。Symbol の場合は、送金アカウントの秘密鍵を使って署名が作られます。トランザクション内容から署名を作れるのは、送金者の秘密鍵だけです。他のノードは送金者の公開鍵を使って、**それが正しい**ということを検証できるわけです。

3.2.1 バッチ検証

大量の署名を処理する場合には、バッチ検証を用いることで 80%のスピードアップが期待できる。Symbol は、文献 [Ber+11]に述べられている方法を使っている。3.2 で出てきた(Mi, Ai, Ri, Si)の組み合わせが大量に存在する場合を考える。ここで、(Ri, Si)は、メッセージ Mi の公開鍵 Ai を使って作られた署名である。この時、それぞれの i に対応するランダムな 128bit の整数 z_i を作る。z は均一に分散していると仮定して、ハッシュ Hi (Ri, Ai, Mi)を計算する。そして、下記の方程式について考えてみる：

$$\left(-\sum_i z_i S_i \bmod q\right) B + \sum_i z_i R_i + \sum_i (z_i H_i \bmod q) A_i = 0 \qquad (9)$$

ここで、Pi = 8Ri + 8HiAi - 8SiBi と置いてみる。この場合式 (9)が成り立つならば、

$$\sum_i z_i P_i = 0 \qquad (10)$$

が、成り立つ。すべての Pi は、q が素数であることから閉じた循環群の要素である。なので、例えばいくつかの Pi (例えば P_2）がゼロではないと仮定すると、整数 z_0, z_1, z_3, z_4....が決まっている場合は、式(10)を満たす z_2 はひとつしかない。つまり、確率的には（128 ビットの整数のうちの 1 つであるから）2^{128} である（多分こういうことが言いたい：ランダムに決めた整数 z_2 が、たまたま z_2P_2 と相殺するような他の z_iP_i の組み合わせを生じるような z_i のセットを生じる確率はほぼゼロ）。すなわち、(9)が成立する場合には、すべての i について、ほぼ $P_i = 0$ と考えて良い。つまり、すべての署名が正しいという事を示す。

もし、式(9)が成り立たない場合は、少なくともひとつの署名が間違っている可能性がある。その場合には、Symbol はすべての個々の署名を検証して、不正なものを見つけ出す。

ここで言いたいのは、式(9)を計算すれば、すべてのトランザクションについて、SB - H(R, A, M)A を計算して R と等しいかどうかを検証しなくても良いということだと思います（検証にかかる時間の節約）。適当なランダム整数の組み合わせ（zi）を用意して、S, R, HA の総和を

別々に計算すれば、計算量を大きく減らせるらしいです。計算機を使った署名の検証時間については、そこそこホットな話題のようですので、興味のある方は文献をあたってみると良いと思います。ちょっとななめ読みしたところでは、CPU のキャッシュをどのくらい有効に使って計算されるかというところに効いてくるようです。

3.3 検証可能な乱数化関数（Verifiable Random Function, VRF）

検証可能な乱数化関数（VRF）が、公開／秘密鍵ペアを擬似的な乱数にするために使われている。秘密鍵の所有者のみが、他者には決して生成できないような数値を作り出すことができる。公開鍵を知っていれば、誰でもその数値が秘密鍵から生成されたものであると検証できる。Symbol では、[Gol+20]で定義された ECVRF- EDWARDS25519-SHA512-TAI を使用する。

まず証明値（脚注；proof，よく検証と混同されるが、秘密鍵から生成した数値であることを、秘密鍵の所有者自らが証明するための数列）を、秘密鍵 SK（Secret Key の略であって s * k の意味ではない） = xB に対応する公開鍵 Y と、シード alpha から生成する（脚注；この章で紹介している数式は、関数定義のすべてを表現していないため、完全な説明については文献[Gol+20]を参照すること）：

$$
\begin{aligned}
H &= \text{map\_to\_group\_element}(alpha, Y) \\
\gamma &= xH \\
k &= \text{generate\_nonce}(H) \\
c &= \text{IetfHash}(3, 2, H, \gamma, kB, kH)[0..15] \\
s &= (k + cx) \mod q \\
proof &= (\gamma, c, s)
\end{aligned}
$$

上の式によって生成された証明値（pfoof）を使って、γ が公開鍵 Y に対応する秘密鍵から生成されたことを、次の計算によって検証が可能である：

$$
\begin{aligned}
H &= \text{map\_to\_group\_element}(alpha, Y) \\
U &= sB - cY \\
V &= sH - c\gamma \\
verification\_hash &= \text{IetfHash}(3, 2, H, \gamma, U, V)[0..15]
\end{aligned}
$$

ここで、計算された検証ハッシュ値（verification_hash）が、証明値（proof）に含まれる c と一致していれば、γ が秘密鍵から x を用いて生成されたことが確かめられる。

また、証明ハッシュ（VRF ハッシュと呼ばれる）は、提供された証明値（proof）に含まれる γ から、以下の計算によって得ることができる。

$$proof\_hash = \mathrm{IetfHash}(3, 3, 8\gamma) \tag{11}$$

3.4 Voting 鍵リスト

ファイナライゼーションに参加する投票者（ノード）は、（投票用の鍵を作る種にあたる）root voting 鍵が影響を及ぼすファイナライゼーションエポック（ファイナライズの単位）の範囲を宣言（アナウンス）しなければならない。この root voting 鍵をアナウンスする前に、投票者は有効なエポック範囲のための voting 鍵を使って、voting 鍵リストを生成する必要がある。リストの構造はほぼ Bellare-Miner [BM99]に記された方法を単純化したものである。リストされた個々の鍵は、それぞれがひとつのエポックに使われ、ファイナライゼーション処理が次のエポックに移った時に破棄される。これによって、前向きな秘匿性が得られる。つまり、たとえ攻撃者が（過去の？）voting 鍵リストを手に入れたとしても、処理が終わったエポックは（鍵が破棄され再利用ができないため？）変更や否認ができないためである。

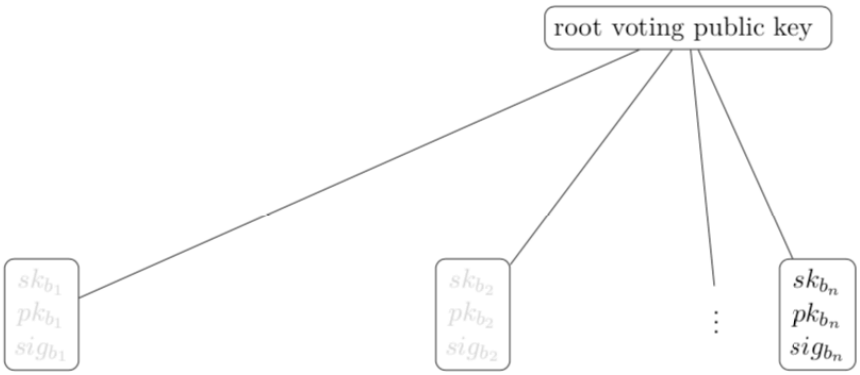


図 1 ：Voting 鍵リスト

このリストは、voting 鍵リンクトランザクションが発行される前に構築される。まず、root voting key pair（ルート voting 鍵ペア）が作られる。この鍵ペアの公開鍵は、アカウントの署名用公開鍵によって署名され、 voting 鍵リンクトランザクションに書き込まれる。

次に、エポックごとの鍵が作られる。それぞれの（b_1〜b_n の）鍵ペアについて、エポック毎に連結された公開鍵に root 鍵ペアが署名する。すべての（n 個の）エポック連結鍵が生成された後、root 秘密鍵は破棄される（ここで秘密鍵が破棄されます）。次の式の中で、i はそれぞれのエポックを表す。

$$sig_{b_i} = \mathrm{Sign}_{root\ secret\ key}(pk_{b_i} || \mathrm{IntToBin}(i))$$

エポックごとに作られる voting メッセージには、（root 秘密鍵はすでに破棄されているために sk_{bi}）を使って署名される：

$$sig_{message\text{-}i} = \mathrm{Sign}_{sk_{b_i}}(message)$$

3.4.1 署名

それぞれのエポックに対してされる署名は 2 組作られる：

- (root voting 公開鍵, sig_{bi})
- (pk_{bi}, $sig_{message\text{-}i}$)

voting 鍵リストは、以下の場合に検証されたとみなされる：

- root voting 公開鍵が、エポックの投票に使われるために登録された時。
- エポックに紐付けられた鍵と、実際にメッセージの署名の署名鍵が一致したとき。
- すべての（エポック内のトランザクションの）署名が、暗号学的に検証されたとき。

ここでは、VRF の関数的な意味のみ定義されています。この VRF 関数は、委任ハーベストに使うアカウントを生成するときや、ブロックのジェネレーションハッシュを作るときに使われて、Symbol のセキュリティー（特にハーベスト関連）を大きく向上させるようです。

ここからは単なる推測ですが、この VRF 関数は、ブロック生成において意図的なフォークを理論的には起こせなくするため、ファイナリティ実現の鍵となっている可能性があります。以前、ネムログ記事（https://nemlog.nem.social/blog/41884）でファイナリティについての考察をしまし

た。これまでの NEM は意図的なフォークをある程度許容するけれども、そのようなフォークしたチェーンは時間的に不利になるようなシステム設計になっていました。VRF の導入によって、次のジェネレーションハッシュやハーベスターのターゲット値を乱数化することで、さらに予測不可能にしたと考えられます。

ファイナライゼーションに使われる秘密鍵は、毎回違うものが使われるため、セキュリティーが非常に高いです。しかし、テストネットの過負荷試験で分かったように、voting 鍵ペアが生成され、それが使用されないうちにネットワークがクラッシュすると、root 秘密鍵が破棄されてしまっていた場合には鍵の再生成ができません。その反省からかどうかはわかりませんが、投票用の voting 鍵は 3 層のツリー構造から大幅に単純化されました。そして、root 秘密鍵が破棄されたあとも、これらの鍵は該当するエポックがすべてファイナライズされるまで、バックアップされるようになりました。

4. ツリー

Symbol では、トラストレスなクライアント軽量化のために、ツリー構造を利用している。マークルツリーを使えば、クライアントノードは暗号化された情報の中に、あるデータが含まれているかどうかを確認できる。パトリシアツリーは、あるデータが含まれているのか含まれていないのかを確認できるようにする。

4.1 マークルツリー

マークルツリー [Mer88]は、ハッシュで構成されるツリー構造であり、あるデータの存在証明を効率的に行うことができる。Symbol においては、すべての基本的なマークルツリーは、バランス化されバイナリー化される。それぞれの葉に当たるノード（図の H(A), H(B)など）は、何らかのデータのハッシュを持っている。葉ではないノードは、（ツリーの下方に接続する）子ノードのハッシュをさらにハッシュ化したものである。Symbol の内部では、幹（Root）以外の層のノード数が奇数の場合、親ハッシュは子ハッシュを 2 つ使って計算される（基本 2 つのハッシュをひとつにまとめるため、奇数だと 1 つ余る。図では E につながるラインが相当）。

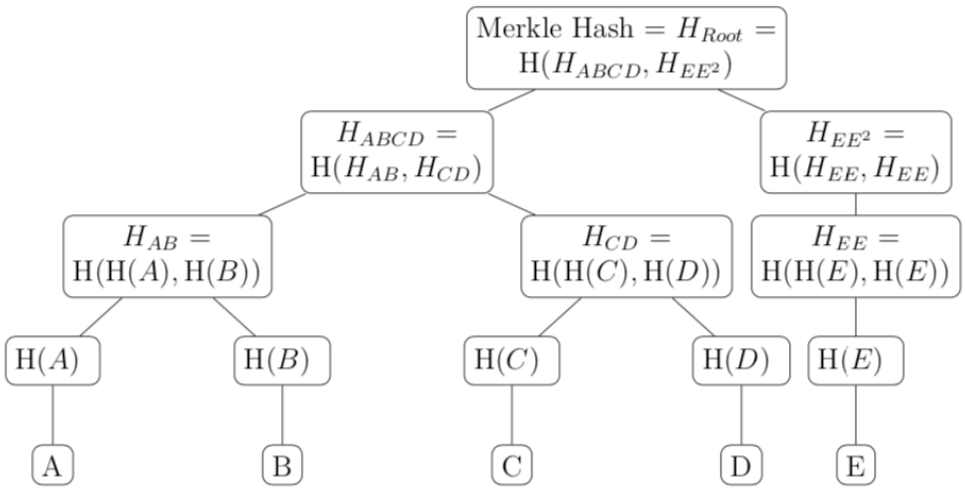


図 2：5 つのデータを持つ、4 層のマークルツリー

マークルツリーを使う利点は、あるハッシュがツリーの中に存在するかどうかを、$\log_2$(N)回の計算（本来なら N 回必要なところ）で済ませることができるところにある。これによってネットワークやノードへの負荷を下げることができる。

ブロックヘッダーにあるルートハッシュが、ブロック内に書き込まれているトランザクションのすべてを反映しているかどうかの検証は、普通ならブロック全体をダウンロードすれば良いと考えます。でも、すべてのノードが、個々にブロックに含まれるトランザクション全体をダウンロードするとなると、ネットワーク負荷が大きく、ブロック生成時間内にすべてのデータが集まる保証もありません。そこで、信頼できるノードが持つトランザクションデータの必要な部分だけ取り寄せて検証しようとするのがマークルツリーです。

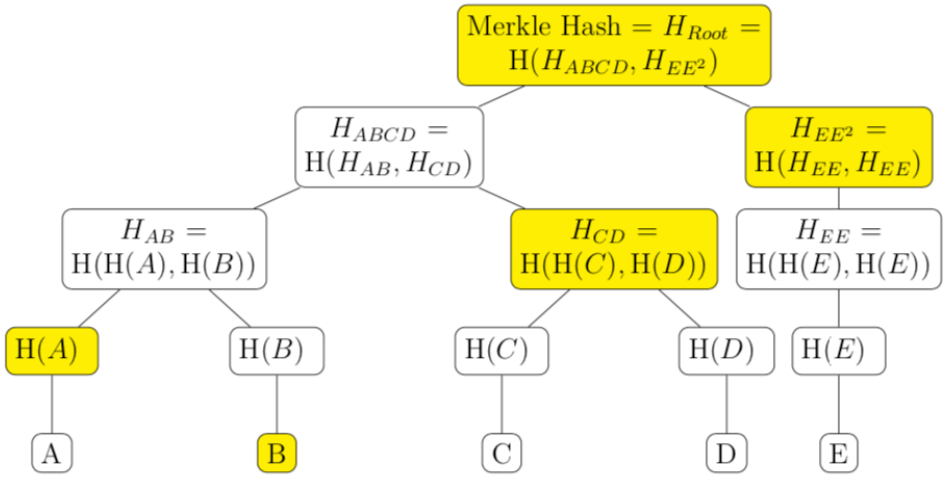


図 3：B の存在証明をするために必要なマークル証明（黄色のデータだけ取り寄せれば良い）

存在のマークル証明は、それぞれの階層から一つずつハッシュを取り寄せれば可能になる。例えば図のように B というデータが改変されていないか検証するためには、クライアントノードは：

1. B のハッシュ（H(B)）を計算する
2. 未承認ブロックのブロックヘッダに含まれる H_{Root} を取り寄せる
3. 他のノードに H(A), H_{CD}, H_{EE2} を要求して取り寄せる
4. H_{Root}' = H(H(H(A), H(B)), H_{CD}), H_{EE2})を計算する
5. H_{Root} と自分で計算した H_{Root}'を比較する；一致していれば H(B)はツリーの中に存在しているのでデータ B の検証成功

ツリー構造を利用して、ブロックチェーンは検証作業を大幅に簡略化できます。例えば Symbol の Peer ノードは、過去のブロックデータをすべて保持していますが、ブロックを生成しようと

するノードが、すべての Peer ノードにトランザクションデータ全体を要求したら、ネットワークがすぐに飽和してしまいます。マークルツリーの一部（マークル枝というらしい）と、それに関連する一部のデータだけを近隣のノードから取り寄せて検証すれば、負荷は小さくて済みます（参考：https://alis.to/mozk/articles/3PYokoOWP7XY）。

4.2 パトリシアツリー

パトリシアツリー [Mor68] は、決定論的に構成されたツリー構造である。（マークルツリーと似た）ハッシュ情報などのキー（秘密鍵や公開鍵のことではなく、検索用の文字列の一部という意味）のペアにより構成され、存在と非存在の両方を log2(N)個のハッシュから導き出すことができる。（マークルツリーではできなかった）非存在の証明がなぜできるかというと、キー（文字列）という形で、ソートされて順番が決定づけられているからである。同じデータ、同じ順番付けであれば、必ず同じツリーが作られる。

新しいキーのペアをツリーに挿入しようと思ったら、そのキーはまず断片化される。そしてそれぞれの断片が、そのツリーの中に決まったルールに従って組み込まれてその一部となる。ツリーの中に組み込まれたすべてのキーは同じ長さであった方が良い。それによってアルゴリズムが少しだけ最適化されるからである。

図を使って説明するために、表 2 のような文字でできたキーの集団を考えてみる。コンセプトの説明をするために文字だけではなく、文字に対応する 16 進数も使って、Symbol への組み込み方法を分かりやすく説明してみたい。

key	hex-key	value
do**	646F0000	verb
dog*	646F6700	puppy
doge	646F6765	mascot
hors	686F7273	stallion

表 2：パトリシアツリーを説明するためのサンプルデータ

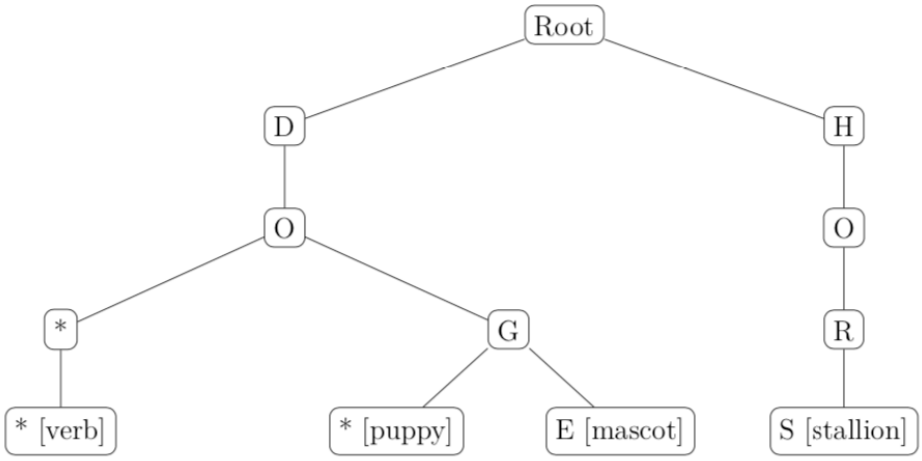


図 4：4 つのデータを使って作ったパトリシアツリー（最大展開済み）

図 4 に、それぞれのキーを構成する文字のひとつひとつをバラバラにして展開したパトリシアツリーを示す。このツリーは論理的には正しいが、もう少しコンパクトにまとめることができる。（Symbol ブロックチェーンで使用される）32 バイトのハッシュは、そのままでは最大 64 個の節（図で四角で囲まれた文字）に展開され、メモリーを消費する。そこで、枝で一直線に連結された節をまとめて、すべての節が 2 本以上の枝か葉（もっとも末端の層にある節）とつながるようにする。そうすることによって、ツリーは元よりずっとコンパクトなもの（図 5）になる。

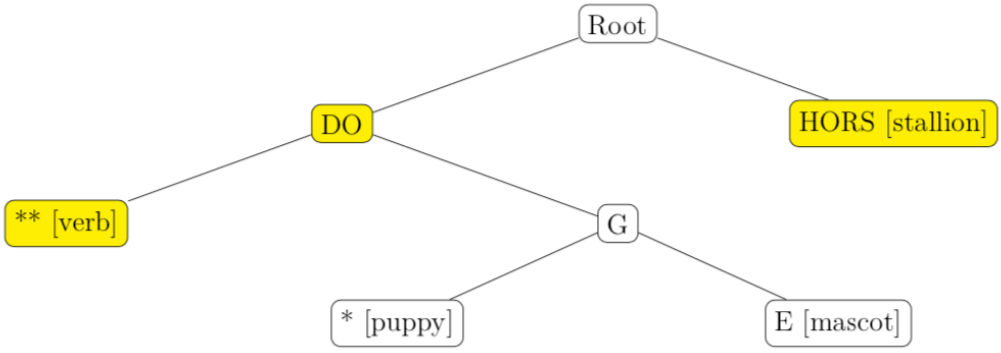


図 5：コンパクトになったパトリシアツリー（黄色がまとめられた節）

ここまで、マークルツリーとパトリシアツリーについて、基本的な原理を学びました。でも、これがハッシュの検証で使われる非存在証明とどういう関係があるのでしょうか？ 読み進めてみましょう。

4.3 マークルパトリシアツリー

マークルパトリシアツリーとは、マークルツリーとパトリシアツリーを合成したものである。Symbol のマークルパトリシアツリーは、2 つのタイプの節（ノード）によって構成されている。すなわち、葉ノードと枝ノードである。葉ノードは、何らかのデータのハッシュを持っている（つまり終端ノードが、実際のトランザクションデータのハッシュを持っている）。枝ノード（図 6 で終端ノードを除く、四角で囲まれたものとそれに下向きにつながる直線で示した数字を合わせた感じ）は、最大 16 個の子ノードへのポインターを保存している（ハッシュを 16 進数表記すると、0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F の 16 種類の文字しか出てこないため）。
基本的なマークルツリーと同様に、マークルパトリシアツリーはルートハッシュを持っている。しかし、マークルツリーに比べてやや複雑になっている。

すべてのノードはパス情報を含んでいる。このパスは、センチネルニブル（パスの頭につける 4bit のインジケータのようなもの：下記 Ethereum 白書からの抜粋参照）に続く 0 個またはそれ以上の他のノードへのパスニブルを伴っている。もしパスが葉ノード（終端ノード）を指しているならば、0x2（0010）がセンチネルニブルにセット（図 6 の b_{leaf}）される。もし、パスが奇数個のニブルで構成される場合は 0x1（0001）がセンチネルニブルにセットされ、2 つ目のニブルが最初のパスニブルになる。もし、パスの数が偶数なら、センチネルニブルには 0x0（0000）がセットされ下位ニブルは 0000 になり、それに続く 2 番目のバイトから最初のパスニブルが記述される。センチネルニブルが 3（0011）の場合は、葉ノードでありかつ奇数個のパスニブルを持っていることになる。

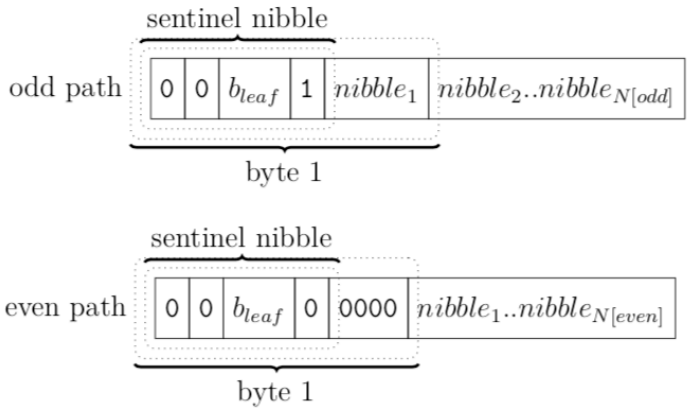


図 6：Tree Node Path の構造

葉ノードを区別する必要性や、パスの数が偶数か奇数かで混乱する問題について、イーサリアムの白書では以下のような説明がなされています。

16 進文字列の「コンパクト符号化」における伝統的方法は、バイナリへの変換です。 というのはつまり 0f1248 という 16 進文字列は [15,18,72]の 3 バイトとなります。 （あるいは string として ¥x0f¥x18¥x72 と表現されます。）しかしこのやり方では、ある小さな問題に直面します。 符号化したい 16 進文字列の長さが奇数だとしたらどうなるのでしょうか？ その場合、いわば 0f1248 と f1248 の区別ができません。 この問題に加えて、Merkle Patricia tree を用いたわれわれのアプリケーションにおいて、16 進文字に加え、終端ノードであることを示す "終端記号" T を加えた 17 文字を アルファベットとして扱うといった仕様の追加が必要となってきます。 終端記号は一度だけ登場し、文字列の最後に置かれます。 違う角度から見てみましょう。 アルファベットに 終端記号 T を加えるのでなく、 その代わりに終端記号の存在を表す bit (b_{leaf})を与え、 それを、与えられたノードの種類を表す 1 bit として扱うという考えに至ります。 この考え方では、他ノードのハッシュ値だけでなく、終端ノードを符号化することができます（終端ノードにおいては、value には実際に使う値が格納されています）。

これら双方の問題を解決するために、 生成されるバイトストリームにおける最初の「ニブル（4bit)」に二つのフラグを符号化します。 それは、（記号 T を無視した上で）長さが奇数であるかを表したフラグと、 ノードが終端状態かどうかを表したフラグです。 これらのフラグは、16 進文字列の先頭のニブルの重要な下位 2bit の中にそれぞれを配置します。 符号化する前の 16 進文字列の長さが偶数の場合は、 符号化した後の 16 進文字列の長さが偶数とするために先頭から 2 番目の「ニブル」を （値０として）導入しなければなりません。 この様にすることで偶奇両方のすべての数のバイトで表現が可能となります。

例：

[1, 2, 3, 4, 5] ；終端ではなく奇数個のパスを持つので、頭に 0001 がつく

'¥x11¥x23¥x45' [0, 1, 2, 3, 4, 5]；終端ではなく偶数個のパスを持つので、頭に 00000000 がつく

'¥x00¥x01¥x23¥x45'

[0, 15, 1, 12, 11, 8, T]；終端で、かつ T を除いて偶数個のパスを持つので、頭に 00100000 がつく

'¥x20¥x0f¥x1c¥xb8' [15, 1, 12, 11, 8, T]；終端で、かつ T を除いて奇数個のパスを持つので、頭に 0011 がつく

'¥x3f¥x1c¥xb8'

葉ノードは、以下の 2 つのアイテムから構成される：

1. TreeNodePath：エンコードされたツリーノードパス（図 6、リーフビット b_{leaf} がセットされる=001 ではじまる）
2. ValueHash：その葉ノードが保有するトランザクションのハッシュ

葉ノードのハッシュは、その内容のハッシュを取ることで計算される。

$$H(Leaf) = H(TreeNodePath, ValueHash)$$

枝ノードは以下のアイテムから構成される：

1. TreeNodePath：ツリーノードパス（図 6、リーフビット b_{leaf} がセットされない＝000 ではじまる）
2. $LinkHash_{0,.....,}LinkHash_{15}$：子ノードへのリンクハッシュ（ゼロハッシュも含めて 16 個並ぶ）。インデックス値として、次のパスへのニブル文字（0 から F のいずれか）がセットされる。もし、子ノードが存在しない場合は、ゼロハッシュが代わりに用いられる。

枝ノードのハッシュは、その内容のハッシュを取ることで計算される。

$$H(Branch) = H(TreeNodePath, LinkHash_0, ..LinkHash_{15})$$

ここで、枝ノードが他のノードを「文字」ではなくて、ハッシュで参照するところが、マークルツリーが導入されている部分です。これによってノード同士が破綻することなくリンクしていきます。

前回のパトリシアツリーで使った 16 進数のキーを使った説明が、Symbol で使っているマークルパトリシアツリーの構造を表現している。それぞれの枝ノードが持つインデックスが、パスを示している。図 7 にその事を示した。

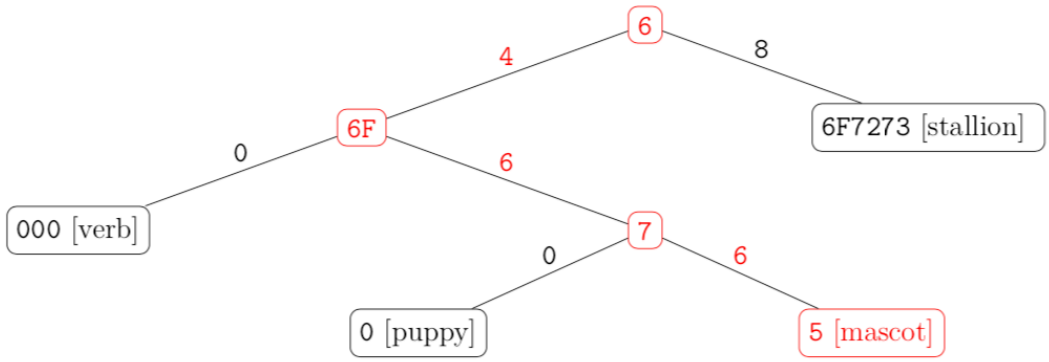


図 7：実際に使っているパトリシアツリー構造。枝ノードと葉ノードによって構成されている。赤で示したのは、mascot[646F6765]へのパスである。

---------- Ethereum 白書からの補足情報 ----------

-

https://github.com/ethereum/wiki/wiki/%5BJapanese%5D-White-Paper を参考に一部改変

ツリー構造においては、すべてのノードは次のように表現されます ：

[value, i0, i1 ... i15]

ここで i0 ... i15 は、ある数字またはアルファベットの記号列（16 進数）を表しており、 そのアルファベット記号単体を表すノードへの**パス**となる値が格納されます。 value には、そのノードを終端とするときのハッシュ値が格納されています。 i0, i1 ... i15 の中の値は NULL もしくは 別のノードへのポインタ です。 図 7 で doge に対応する値 mascot を知りたいとすると、 まず、doge を該当アルファベットに （ここでは 16 進数を使用して 646f6765 に） 変換します。 そして、646f6765 というパスに従って木を下っていき、 パスの最後にたどり着いたら、そこで値 mascot を読み取ります。 つまりまずはじめに、root node を得るためにノードが持っている value の中で、ブロックヘッダに格納されている root hash と一致するものを探します。 そして、次に root node において 6 の接続ノードへの値を参照（4 または 8）し、一つパスを下ります。 そして、次はそこで 4 のノードに接続するノードの値 6F を参照し、さらに 6F のノードへつながる 6 または 0 のノードを参照するという具合に繰り返し、 root -> 6 -> 4 -> 6F -> 6 -> 7 -> 6 -> 5 というパスを辿り、そこで結果として返すノードの値 mascot にたどりつきます。

Example: ('do', 'verb'), ('dog', 'puppy'), ('doge', 'mascot'), ('horse', 'stallion') の４つのペアを含む マークルパトリシア木があるとします。はじめに key を 16 進数形式に変換します（最後の 16 は終端であることを示す印）：

[6, 4, 6, F, 16] : 'verb'

[6, 4, 6, F, 6, 7, 16] : 'puppy'

[6, 4, 6, F, 6, 7, 6, 5, 16] : 'mascot'

[6, 8, 6, F, 7, 2, 7, 3, 6, 5, 16] : 'stallion'

さて、マークルパトリシア木を構築してみましょう:

ROOT: ['¥x16', A] ; 0001(パスが奇数個（1 個）) +'6' という値で A へつながる

A: ['', '', '', '', B, '', '', '', C, '', '', '', '', '', '', '', '', ''] ;

'4' が B につながる, '8' が C につながる

B: ['¥x00¥x6f', D] ; 0000(パスが偶数個（2 個）), '6F' という値で D につながる

D: ['', '', '', '', '', '', E, '', '', '', '', '', '', '', '', '', 'verb'] ; '6' が E につながる, ここで終わり'verb' という値にたどり着く（do）

E: ['¥x17', F] ; 0001（パスが奇数個） + '7' という値で F につながる

F: ['', '', '', '', '', '', G, '', '', '', '', '', '', '', '', '', 'puppy'] ; '6' という値で G につながる, ここで終わりで'puppy' という値にたどり着く(dog)

G: ['¥x35', 'mascot'] ; 0011（葉ノードなので終点かつパスは奇数個（1 個）） + '5' という値で 'mascot' という値につながる

C: ['¥x20¥x6f¥x72¥x73¥x65', 'stallion'] ; 0020（葉ノードで終点かつパスは偶数個(4 個）+ '6F727365' という値で'stallion' という値にたどり着く（horse*）

*; Symbol 白書の図 7 では horse が hors になっているので注意。多分スペルミス。

ちなみにこれは単なるパトリシアツリーの説明であり、ハッシュ化を伴ってはじめてマークルパトリシアツリーになるようです。

マークルパトリシアツリーについては、イーサリアム関連の文献を当たっても完全には理解できませんでした。しかし、コードにすればわかりやすいものになるようで、単に、文章化が難しいのだという気がします。ネットには詳細な解説（https://easythereentropy.wordpress.com

/2014/06/04/understanding-the-ethereum-trie/）もあります（Python2 ですがコードもついています）ので、興味のある方は試してみると良いかもしれません。

4.4 マークルパトリシアツリーの検証

あるハッシュがブロックに存在するかどうかをマークルツリーで検証するためには、それぞれの階層から一つずつノードを取り出せばよい。例えば、図 7 で{key = 646F6765, value = H(mascot)}の存在証明をしようと思ったら：

1. H(mascot)を計算する（すべての葉が持つデータはハッシュである。ハッシュのメリット＝すべて同じ長さで重ならない）
2. key = 646F6765 に連なるすべての葉と枝ノード（$Node_{6}$, $Node_{646F}$, $Node_{646F67}$）をリクエストする
3. ノード 646F67::Link[6](=646F67::6::5, 右下の終端ノード)に保存されているハッシュが H(Leaf(mascot))に等しいことを検証する
4. H(ノード 646F67, 図の中では枠で囲まれた 7)を計算して、ノード 646F::Link[6](=646F::6::7)が H(ノード 646F67)と等しいことを検証する
5. H(Node646F)を計算して、ノード 6::Link[4](=6::4::6F)に保存されているハッシュが H(Node646F)と等しいことを検証する
6. すべての計算値とツリーに保存されていたハッシュ値が一致すれば存在検証成功

マークルツリーを使って非存在を検証するためにも、同様に各ノードレベルから一つずつノードを取り出せば良い。例えば図 6 で｛key = 646F6764, value = H(mascot)}が存在しないことを検証しようと思ったら（赤字の 4 が間違っているため存在しない key である）：

1. H(mascot)を計算する
2. 646F6764 に連なるすべてのノードをリクエストする
3. ノード 646F67::Link[6](=646F67::6::x)が H(Leaf(mascot))であるかどうか確かめる。この場合、6 のリンクは 646F67 のノードにセットされているが、次に 64 となる葉ノードが存在しないため、マークルパトリシアツリーの唯一性によって、その存在を否定できる（原文では 5 の枝が無いと書かれているが、実際は 6 の枝をたどると 5 のはノードに行き着く。そこから葉ノードに 4 のものがないことを見ていると思われる）。

これで、マークルパトリシアツリーの説明は終わりです。ブロックチェーンの安定性を高めるためには、Peer ノードによる必要かつ充分な検証が必要です。そこで、Symbol では高速に大量のトランザクションハッシュを処理するために、イーサリアムで採用されているマークルパトリシアツリーを採用しています。実際どの程度の高速化・省力化がなされているかについては分かりませんが、簡易であっても 16 進数を用いた説明で、何をやっている（やろうとしている）かは、ある程度理解できました。マークルパトリシアツリーについてはイーサリアム関連の情報がたくさんネットにあるので、それを参考にしながら少しずつ理解を深めていけばよいと思います。

5. アカウントとアドレス

「ものの始まりとは、きちんとバランスが取れているかどうかについて、深く考える時のことである」

- フランク・ハルバート（アメリカの小説家、SF 作家）

-

Symbol は、トランザクションの機密性、真正性、責任遂行性を保証するために、楕円曲線暗号を用いている。アカウントは、アドレスによって一意に定義され、その大部分は署名にも使用される公開鍵から生成される（逆変換はできない）。それぞれのアカウントは、ネットワークによってトランザクションが承認されたときに発生する状態（ステート）変更とリンクしている。この状態変更はネットワーク全体で共有されるが、ひとつまたは複数の公開鍵を含んでいる場合と、公開鍵を持たない場合がある。

5.1 アドレス

デコード済みアドレスは、24 バイトの長さを持つ値であり、以下の 3 つのパートから構成される：

- ネットワークバイト
- 署名用公開鍵の 160-bit ハッシュ
- 3 バイトのチェックサム（0.9.6.3 から 1 バイト減らされました）

チェックサムによって、アドレスのタイプミスを検出できる。モザイク（脚注；モザイクとは Symbol で使用されるデジタルアセットのことである。他のブロックチェーンではトークンと呼ばれる）を、過去に一度もトランザクションに関与したことがない（ブロックチェーンに記録がない）アドレスに送ることも可能である。しかし、もしそのアドレスに紐付いた秘密鍵を誰も所有していなかった場合は、ほぼ確実に送られたモザイクは失われる。

デコード済みアドレスを Base32 (脚注；http://en.wikipedia.org/wiki/Base32）でエンコードしたものをエンコードアドレスと呼ぶ。これは、人間にとって理解しやすい形になっている。

バイナリーデータを Base32 でエンコードすると、8/5 倍の長さに引き伸ばされる。しかし、デコード済みアドレスは必ずしも 5 の倍数の長さにはなっていない（5 バイトを 8 バイトに変換するため、5 の倍数でないと困る）。そこで、デコード済みアドレスを Base32 でエンコードするときに、データの付加がおこなわれる。これは必ずしも必要な事ではないが、デコード済みアドレスをエンコードするときには、長さを 5 の倍数にするために 0 を付加することで、ウォレットなどのクライアントアプリが扱いやすいようにしている。また、結果として得られる 40 文字のアドレスから、最後の 1 バイト（上で付け加えた 0＝base32 で A）を取り除くこととする。

5.1.1 アドレス導出

公開鍵をアドレスに変換する場合は、以下の手順によって行う：

1. 公開鍵を 256bit の SHA3 にて変換する
2. さらに、160-bit RIPEMED_160 方式によってハッシュ化する（2 重ハッシュ）
3. ネットワークバージョン（1 バイト）を RIPMED_160 ハッシュの前に付け加える
4. 結果を 256-bit SHA3 によってハッシュ化し、最初の 3 バイトをチェックサムとして取り出す
5. ステップ 3 とステップ 4 によって得られたチェックサムを結合する
6. 得られた結果を Base32 によってエンコードする

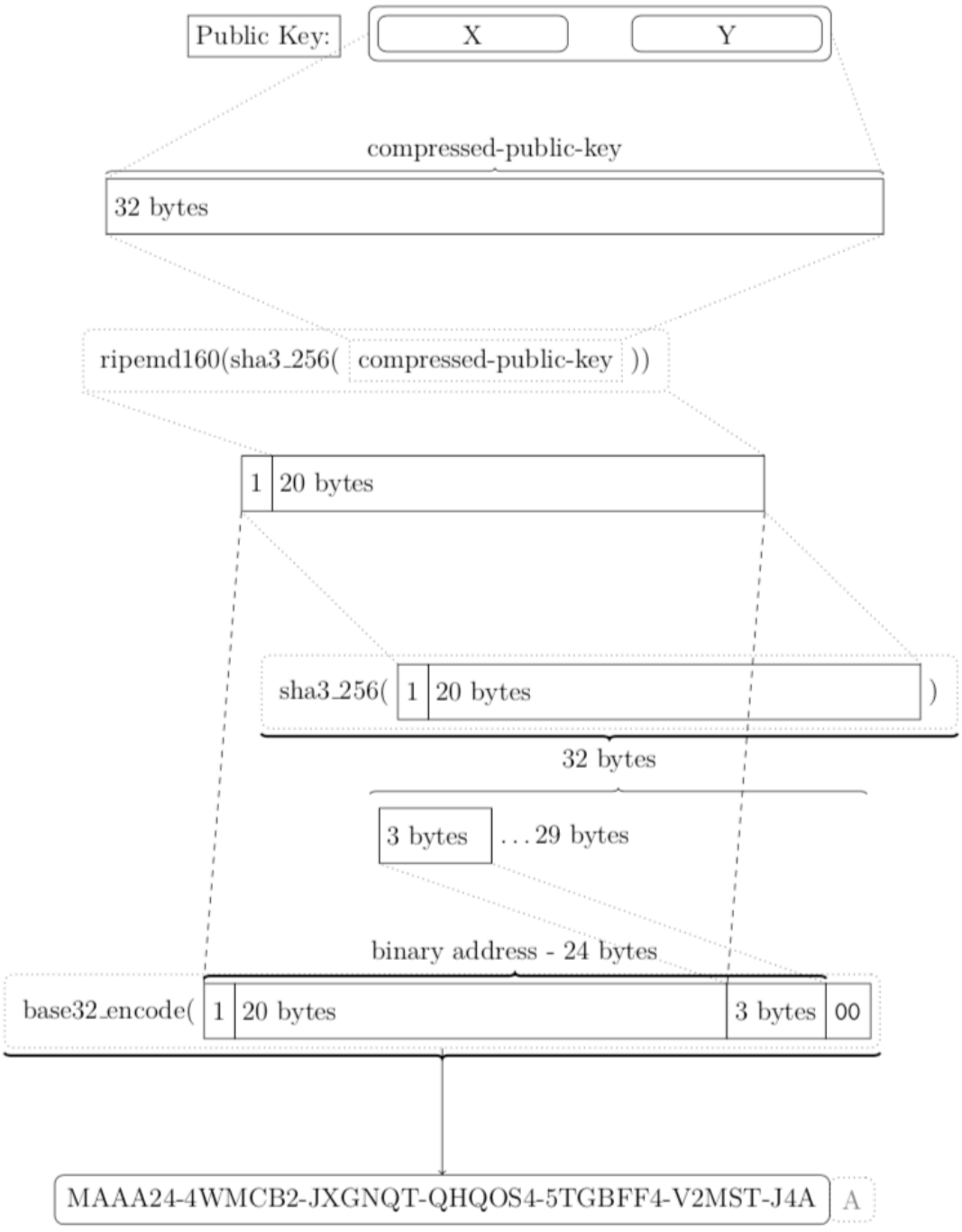


図 8：アドレス生成

Symbol では、チェックサムが 3 バイトに減らされて、バイナリー（デコード済み）アドレスは 24 バイトの長さになっています。それを Base32 でエンコードするときに 5 の倍数である 25 バイトにするために 00 を付加し、エンコード後に最後の一文字「A」を削るということをしています。

5.1.2 アドレスエイリアス

アドレスは、1 つあるいはそれ以上のエイリアスを持つことができ、それはアドレスエイリアストランザクション（脚注；https://nemtech.github.io/concepts/namespace.html#addressaliastransaction）によって生成される。アドレスと関連のあるすべてのトランザクションは、公開鍵から作られたアドレスとアドレスエイリアスの両方をサポートする。その場合、アドレスエイリアスを指定する部分は以下のように表現する：

- 1 と OR 計算されたネットワークバイト（ネットワークバイトはアドレスの最初についているアルファベットのことです。つまりテストネットアドレスの T (0x98)や NIS1 アドレスの N (0x68)など。これは、他のネットワークのエイリアスと簡単に区別するために付けられます）
- エイリアスとして用いる文字列と紐付けられた 8 バイトのネームスペース ID
- 15 個の 0 バイト（結果としてバイナリーアドレスと同じ合計 24 バイトになる）

5.1.3 故意のアドレス衝突

2 つの異なる署名用公開鍵が、同一のアドレスを生成してしまう（衝突する）可能性はある。もし、そのようなアドレスに価値のあるアセットが保有されていて、**かつ**、他の公開鍵に紐付けられていなかった場合（つまり、アカウントから一度も送金がされていない場合など）のみ、攻撃者はそのアカウントからアセットを引き出すことができる。

しかし、そのような攻撃が成功するには、攻撃者は秘密鍵と公開鍵の組み合わせを以下のような条件下で見つけなければならない。すなわち、（偽装しようとする）公開鍵の SHA3_256 ハッシュが、**同時に**、先に説明した ripemd-160 が生成する 160 ビットのハッシュ（アドレス生成に使われたもの）でも同一のものを生成するようにしなければならない。SHA3_256 のハッシュ化は 128 ビットの長さのセキュリティーを提供するため、SHA3_256 の衝突が見つかることは数学上まずあり得ない。Symbol アドレスとビットコインのアドレス生成方法の類似性により、このような Symbol アドレスの衝突の可能性は、ビットコインのそれと同等と考えて良い。

これで、アドレスに関する部分は終わりです。エイリアスによってアドレスにもある種の名前をつけられるようになった（NIS1 でも保有するネームスペースを使ってできてましたが、あまり有名ではなかった）のが、新しい点だと思います。アドレスやモザイク ID は永久保有できて、エイリアスのほうに保有期間が自由に設定できるので、使い勝手が良くなっています。またアドレスは複数のエイリアスと関連付けることができます。

5.2 公開鍵

アカウントは、公開鍵を持たないか、1 つ以上持つことができる。以下の複数のタイプの公開鍵が定義されている：

1. 署名鍵：ED25519 で生成された公開鍵であり、データの署名と検証に使われる。すべてのアカウントがデータを受け取れるが、データを送り出せるのはこの公開鍵を持ったアカウントのみである。この署名用公開鍵のみがアドレス導出に利用される。
2. リンク鍵：この公開鍵は、リモート（委任）ハーベストアカウントをメインアカウントとつなぐ鍵である。**メインアカウントにとって**は、この公開鍵は（自身の署名用鍵ペアの）代理鍵として、ブロックに署名するリモートハーベストアカウントとつながっている。また、**リモートアカウントから見れば**、この公開鍵はブロックに署名するための自身のメインアカウントと接続していることになる。このリンクは双方向性であり、常に（メインアカウント署名鍵との）一対で定義されている。
3. ノード公開鍵：この公開鍵は、ノード所有者がノードに設定した**メインアカウント**と対である。また、リモートハーベストアカウントにとっては委任先ノードの公開鍵である。重要なのは、この公開鍵が（ウォレットなどで見た時）存在しているからと言って、必ずしもそのノードでリモートハーベストをしているとは限らない。あくまで許可されているというだけのことである。リモートハーベストアカウントとノード所有者のどちらかが正直である限りは、同時に複数のノードで同じ鍵が使用されることはないはずである。
   正直なハーベスターは、リモートハーベスト用の秘密鍵を、ひとつだけのノードに置いているはずである。委任状態を変更する場合は、それより前に得たリモートハーベスト許可を、変更先ノード以外のすべてのノードから取り消すはずである（その結果、委任鍵の前方セキュリティが保証される）。過去のリモートハーベスターの秘密鍵は、委任を解除されたノードにおいては無効であり、ブロックをハーベストするのには使われない。正直な

ノードオーナーは、*現時点*でノードの公開鍵にリンクされている、リモートハーベスターの秘密鍵を使ってのみ、ハーベストをおこなうはずである。

4. VRF：ED25519 で定義される公開鍵であり、ランダムな数値を生成・検証するために使われる。ハーベスターのメインアカウントにセットされる必要があり、ハーベストの資格がある事を証明するために使われる。
5. Voting（投票鍵）：BLS 公開鍵であり、ファイナライゼーションの署名と検証に使われる。すべての投票鍵には寿命が設定されており、開始と終了、両方のエポック高とともに登録されなければならない。この公開鍵は、ノードのメインアカウントごとに設定されねばならず、投票権を保有することを証明する。登録された開始と終了のエポックの間だけ有効である。鍵の交換がスムーズにおこなわれるために、アカウントは network:maxVotingKeysPerAccount の数まで、一度に登録することができる。

NIS1 時代に比べて委任ハーベスト周りが複雑になっています。VRF によってランダムに複数の委任ハーベスト用鍵ペアを生成するようになり、安全性が高まりました。

6. トランザクション

「実際、経済活動というものは大抵、フリードリッヒ・ハイエク（ドイツのノーベル賞経済学者）が述べたような、高いトランザクションコストを払っていては達成できない、あるいはトランザクションコストを減らす、または無くすために、個々人が自由な交渉をするように広まった知的行動なのである。」

- ロナルド・コース（アメリカのノーベル経済学者）

トランザクションは、グローバルチェーン（ブロックチェーン全体）の状態（ステート）を変更する命令である。それらは分割不能単位として処理され、それが集積したものがブロックに書き込まれる。トランザクションの一部でも処理に失敗した場合、グローバルチェーンの状態は、トランザクション試行前の状態にリセットされる。

トランザクションには、ベーシックトランザクションとアグリゲートトランザクションの 2 つの基本的なタイプがある。ベーシックトランザクションは単一のトランザクション操作を表し、単一の署名が必要になる。アグリゲートトランザクションは、複数の署名を必要とする 1 つ以上のトランザクションを詰め込んだコンテナである。

アグリゲートトランザクションにより、ベーシックトランザクションを（スマートコントラクトのような）複雑な操作が可能な状態に結合し、アトミックに実行できる。これにより、個々のベーシックトランザクション単位の分割不能性のみを保証する従来のシステムに比べて、開発者の利便性が向上するが、トランザクションによって可能な操作は限定される。しかし、（イーサリアムが目指すような）チューリング完全言語や、それに付随する欠点もまた使用せずに済む。開発者は、特殊な新しい言語を習得することなく、スクラッチからコントラクトの実装を開発する必要もない。連続するトランザクションの実行は、エラーから解放され、計算機科学的に完璧な実装をするよりも、はるかにバグを生じにくくなるはずである。

6.1 ベーシックトランザクション

ベーシックトランザクションは、暗号技術的に検証可能なデータ部分と、検証不要なデータ部分で構成される。検証可能なデータはすべて連続し、ひとかたまりになっており、トランザクション署名者によって署名される。すべての検証不要なデータは無視されるか、検証可能なデ

ータから確定的に導出可能なものである。それぞれのベーシックトランザクションは、1 つの署名の検証が必要になる。

検証不要なヘッダーフィールドは以下のものを含む。Size は、トランザクションのバイト数であり、検証可能なトランザクションデータから取得できる。Singature は、署名操作の出力結果であり、検証に必要なデータである。SignerPublicKey（署名者の公開鍵）は、署名と検証の両方に必要である。トランザクション T が署名を用いた検証に合格するには、Signature と SignerPublicKey の両方が、Size によって規定される長さを持つ、検証可能データを用いた計算結果と一致する必要がある。

$$\mathrm{verify}(T{::}\mathrm{Signature}, T{::}\mathrm{SignerPublicKey}, \mathrm{VerifiableDataBuffer}(T))$$

検証作業は、3.2 署名と検証で出てきた手順どおり、署名（R と S）と公開鍵を使って、トランザクションの内容（Verifiable Data Buffer）をチェックします。

予約（Reserved）バイトは、すべての整数フィールドと暗号関連文字列が自然なアライメントを保つように、トランザクションの整形（8 バイトの桁揃え）に使用される。これらのバイトは意味を持たないため、暗号化によるセキュリティーに影響することなく無視できる。

すべてのトランザクションタイプのバイナリレイアウトは、Symbol のオープンソーススキーマ言語で指定されている（脚注；スキーマについては https://github.com/nemtech/catbuffer で見ることができる）。最新の仕様については、公開されているスキーマを参照。

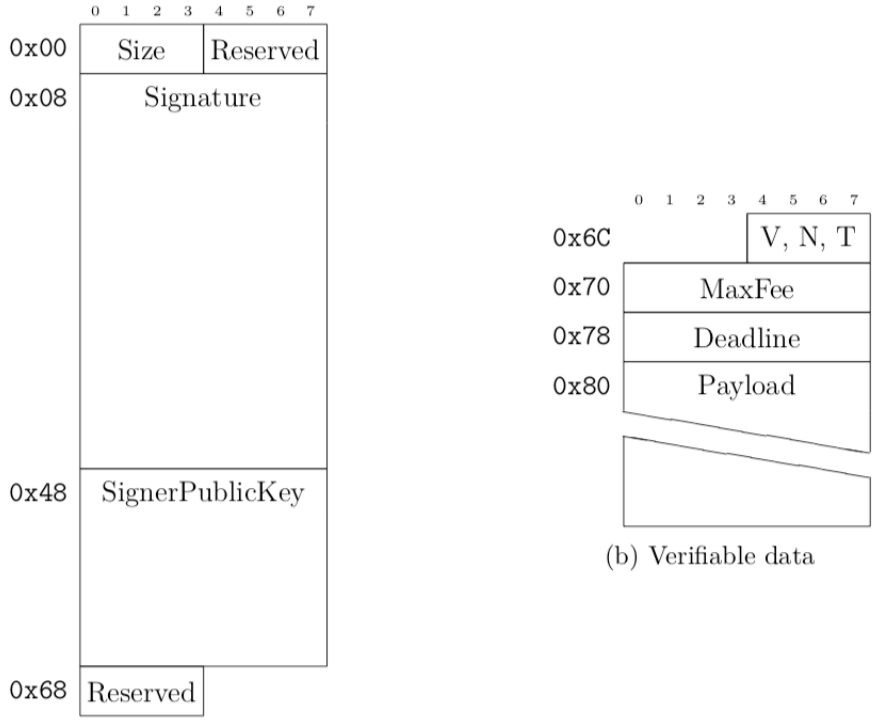


図 9：ベーシックトランザクションのバイナリーレイアウト

V, N, T はそれぞれ(V)ersion, (N)etwork, (T)ype を表す。スペースの関係で省略。

補足：Symbol オープンスキーマ言語で書かれたトランザクションのバイナリー構造例

```
import "entity.cats"
 # binary layout for a transaction struct Transaction
       inline SizePrefixedEntity
       inline VerifiableEntity
       inline EntityBody
       # transaction fee
       fee = Amount
       # transaction deadline deadline = Timestamp
 # binary layout for an embedded transaction header
struct EmbeddedTransactionHeader
       inline SizePrefixedEntity
       # reserved padding to align end of EmbeddedTransactionHeader on
         8-byte boundary
```

```
        embeddedTransactionHeader_Reserved1 = uint32
 # binary layout for an embedded transaction
struct EmbeddedTransaction
        inline EmbeddedTransactionHeader
        inline EntityBody
```

トランザクションのバイナリー構造（https://github.com/nemtech/catbuffer/blob/master/schemas/transaction.cats）の一部を抜き出してみました。これだけ見てもよくわからないですが、8 の倍数バイトの単位でデータの区切りが出てくるように Reserved が配置されて、図 9 の構造体が定義されていることがわかります。

複雑に見えるトランザクションも、本体は 16 進数で書かれた数字の列です。この数字の列から必要な数値や文字列を読み出して検証をおこなうために、形式が厳密に定義されています。

6.2 アグリゲートトランザクション

アグリゲートトランザクションのレイアウトは、ベーシックトランザクションに比べて複雑である。しかし、似通っている部分も多い。アグリゲートトランザクションは、ベーシックトランザクションと同じ「検証不要なヘッダー」を持ち、この部分の処理はまったく同じである。さらに、アグリゲートトランザクションにはフッター（検証可能データの後に置かれる部分）にも、検証不要データが存在し、埋め込み（embedded）トランザクションと共同署名が連なる形になっている。

アグリゲートトランザクションは、必要とされるすべての共同署名と一緒にネットワークに投入される場合もある。その際は、即時「完了」したと判断され、特別な工程無しに他のトランザクションと同様に処理される（すぐに検証工程に入りブロックチェーンに刻まれるための作業に移る）。

API ノードは、完成していない共同署名がなされた、債券付き（bonded）アグリゲートトランザクションを受け入れることもできる。トランザクションの発行者は、期限が切れる前にすべての共同署名が集められた場合にのみ返金される債券を発行しなければならない。この前金制

度のもとで、API ノードは共同署名を集める作業をする。この作業は、必要な数の署名が集まるか、債券が期限切れになるまで続けられる。

TransactionsHash（トランザクションハッシュ）は、アグリゲートトランザクションの最も重要なフィールドである。これは、アグリゲートトランザクションに埋め込まれたすべてのトランザクションハッシュの、マークルルートハッシュとなる。埋め込まれた複数のトランザクション（のハッシュ？）は、自然な順番でソートされ、マークルツリーが作られる。その結果がルートハッシュとなり、TransactionsHash として検証可能データの一部として組み込まれる。

そのため、検証不要フィールドにあるデータは、あらためて検証される必要がなくなる。PayloadSize は、TransactionsHash を計算するのに使用された、埋め込みトランザクションが入っている部分の正確なサイズである。Reserved バイトは、今回も桁揃えのために使われて、中身に特に意味はない。

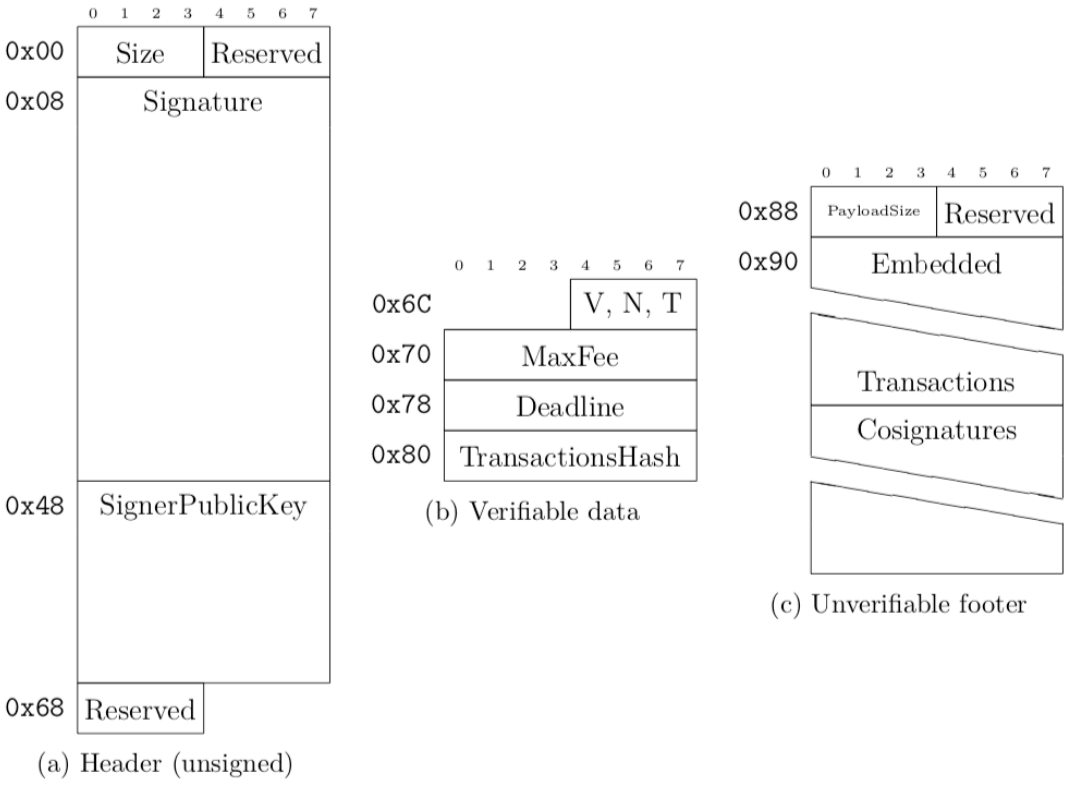


図 10：アグリゲートトランザクションのヘッダーのバイナリーレイアウト

アグリゲートトランザクションは、複数のトランザクションと複数の署名をひとつにまとめたものですが、マークルパトリシアツリーを使って、全体のルートハッシュ（TransactionsHash）を計算しているのがポイントです。また、投入された時に、債券という形で手数料が支払われ、API はその前金を使って働く感じになっていますね。期限までに共同署名が集まらない場合は、支払った手数料は没収され、ハーベストしたアカウントのものになります。

6.2.1 埋め込みトランザクション

埋め込みトランザクションとは、アグリゲートトランザクションに内包されるトランザクションのことである。ベーシックトランザクションと比較した場合、ヘッダーは少し小さい（署名データなどが省かれる）。しかし、トランザクションに関連するデータとしては同じものが含まれる。Singature（署名）が無いのは、すべての共同署名のデータが別の場所にまとめて置かれるからである。MaxFee（最大手数料）と Deadline（期限）もまた、上位のアグリゲートトランザクションに記述されているため、省略されている。

クライアントプログラムは、ベーシックと埋め込みトランザクションの作成に、共通のコードを使うことができるようになっている。違いは、異なるヘッダーを作成し適用することである。

すべての種類のトランザクションが埋め込みトランザクションに入れられるわけではない。例えば、アグリゲートトランザクションを、他のアグリゲートトランザクションに入れることはできない。

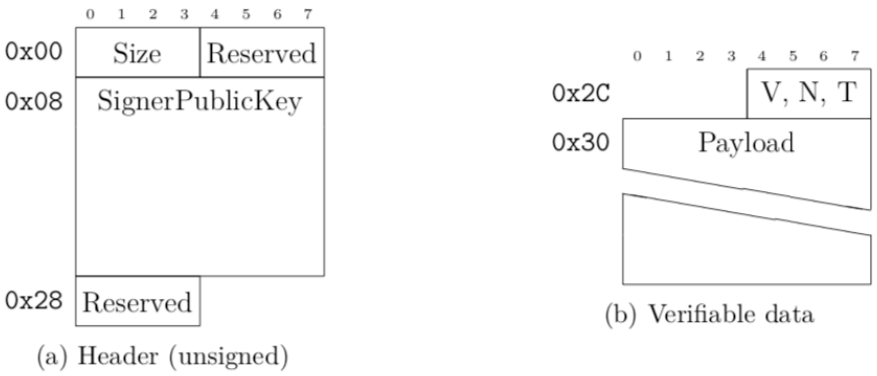


図 11：埋め込みトランザクションのバイナリーレイアウト

アグリゲートトランザクションの内容がだんだん分かってきました。アグリゲートトランザクションには、トランザクションが複数埋め込まれ（Embedded）ており、その形式はベーシックトランザクションとよく似ています。ヘッダーから署名、手数料、期限の項目が取り除かれている点が違います。

6.2.2 共同署名

共同署名は、バージョン（脚注；将来のアップデートに備えたもので、現在のところすべて0である）、公開鍵とそれに対応する署名のペアで構成される。ゼロ個またはそれ以上の共同署名は、アグリゲートトランザクションの最後に追加されていく。共同署名は、複数のパーティー（内包される個々の埋め込みトランザクションに関わるアカウントの集団のことをパーティと呼ぶ？）を含んだアグリゲートトランザクションを暗号学的に検証するために使われる。

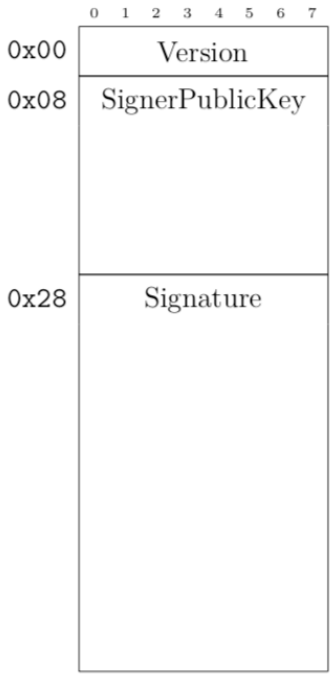


図 12：共同署名のバイナリーレイアウト

アグリゲートトランザクション「A」が、検証に合格するためには、まずベーシックトランザクションとしての署名の検証を経た後、すべての埋め込みトランザクションの共同署名を集め

なければならない（脚注；共同署名アカウントがアグリゲートトランザクションを作る場合には、マルチシグのルールに従って n of m 個のマルチシグ署名を集めていなければならない）。

他のトランザクションと同様に、アグリゲートトランザクションはまず、ベーシックトランザクションとしての検証を受ける。

$$\text{verify}(A\text{::Signature}, A\text{::SignerPublicKey}, \text{VerifiableDataBuffer}(A))$$

次に、すべての共同署名者が、署名検証（signature verification, 署名が揃っていることを確認する作業）を受ける。ここで気をつけるべきことは、これらの共同署名者はアグリゲートトランザクションのデータに対してではなく、アグリゲートトランザクションデータのハッシュに対して署名作業を行えば良いということである。

$$\sum_{0 \le i \le N_C} \text{verify}(C\text{::Signature}, C\text{::SignerPublicKey}, H(\text{VerifiableDataBuffer}(A)))$$

最終的には、それぞれの埋め込みトランザクションについて、対応する共同署名が存在しなければならない。

0.9.6.3 から、バージョンという新しい項目が付け加わりました。つまり、このアグリゲートトランザクションの形式はまだ暫定的なもので、将来より使いやすいものにアップデートされる可能性があるということです。アグリゲートトランザクションは、スマートコントラクトをシンプルで柔軟性を持った形式に落としこむことで、ユーザーエラーを防ぐ仕組みですので、現在の実装形式が最適であるということにはならないということです。

アグリゲートトランザクションの最初の検証は、トランザクション A を作成したアカウントの署名に対しておこなわれるはずです。その後、共同署名者が次々と A の作ったハッシュ（マークルルートハッシュ？）に署名して、それがアグリゲートトランザクションの末尾に付け加わっていきます。これらの共同署名者は、それぞれの埋め込みトランザクションの発行者に加わります。埋め込みトランザクションの全部を代表するマークルルートハッシュに署名することで、共同署名したトランザクションを承認したとみなすことができます。

しかし、内容を確認せずに安易に署名してしまうと、自分のウォレットからの送金までも承認してしまいます。そのスキを狙った詐欺も発生しているので注意が必要です。

6.2.3 拡張用レイアウト

アグリゲートトランザクションのレイアウトについて先に紹介した内容に、一部補足をする。全ての埋め込みトランザクションの末端が、8 バイトごとの区切りと一致するように、パディングバイトを追加できるようになっている。こうすることにより、すべての埋め込みトランザクションの先頭も、8 バイト区切りと一致するようになる。これらのパディングバイトは署名には含まれず、またハッシュにも影響を与えない。

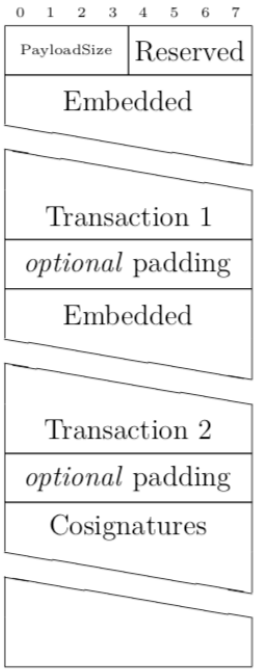


図 13：アグリゲートトランザクションのフッターにパディングバイトを付け加えて桁揃えしたもの

アグリゲートトランザクションのフッターには、埋め込みトランザクションが並びますが、これらはかならずしも 8 バイトの倍数になるとは限らない部分です。ですから、桁合わせ用のパディングバイトを付け加えて 8 バイト区切りに次の埋め込みバイトや署名の先頭がくるようにしてあります。こうすることで、プログラムが高速化できるのと、トランザクションの内容を無理やりサイズ調整する必要が無くなります。このあたりの細かい配慮も、Symbol の高速化に効いてくるのでしょう。

6.3 トランザクションハッシュ

個々のトランザクションには 2 つのハッシュ値が付加される - 実体ハッシュとマークルコンポーネントハッシュである - 。実体ハッシュは、トランザクションを他のトランザクションと区別する単一のハッシュ値として機能し、同じトランザクションが同じノードで何度も検証されるのを防ぐ。マークルコンポーネントハッシュは、トランザクションハッシュ値（6.2：アグリゲートトランザクション参照）を計算するための、特別なハッシュ値である。

トランザクションの実体ハッシュは以下のようにして計算される：

1. トランザクション署名（Singature） - この項目が存在しないと、攻撃者は特定のトランザクションに似せて作ったニセ署名付きトランザクションを忍び込ませて、ターゲットのトランザクションがネットワークに記録されないように工作できてしまう。
2. トランザクションを署名したアカウントの署名用公開鍵（SignerPublicKey） - この項目がない場合、攻撃者はやはり特定のトランザクションの偽物を作ることで、ターゲットのトランザクションを妨害できることになる。
3. ネットワークジェネレーションハッシュシード（GenerationHashSeed） - この項目は、複数のブロックチェーンネットワークが混在する場合に、他のネットワークとの混線を利用した繰り返し攻撃を防ぐことができる（脚注；さらに、トランザクションデータへの署名や検証の際に、このジェネレーションハッシュシードが先頭にあることで、このシードが一致するネットワークのデータのみを取り扱えば良い）。
4. トランザクションの検証可能データ（トランザクション本体）。

すべての承認済みトランザクションは、独自の実体ハッシュを持つはずである（ハッシュ値の非衝突性）。アグリゲートトランザクションの実体ハッシュ値は、共同署名とは関係なく計算される。こうすることで、同じアグリゲートトランザクションが新しい共同署名が付け加わる度に、異なる共同署名のセットによって何度も再検証されることを防ぐ。

普通の（ベーシック）トランザクションのマークルコンポーネントハッシュは、実体ハッシュと同一になる。アグリゲートトランザクションのマークルコンポーネントハッシュは、実体ハッシュのあとに、すべての共同署名者の公開鍵を付け加えたもののハッシュを計算する（脚注；共同署名の（トランザクション payload に対する）署名は、特定の公開鍵とトランザクション payload に対してひとつの署名値しか持てないルールのため、おこなわない）。こうする

ことで、トランザクションハッシュは、アグリゲートトランザクションを承認したすべての共同署名をきちんと反映したものになる。

ベーシックトランザクションに付けられるマークルハッシュ値はこれまでと同じトランザクション内容のハッシュですが、アグリゲートトランザクションには、トランザクションのハッシュに共同署名者の公開鍵をならべたものから作られたマークルハッシュが使われるようです。共同署名フィールドを含めて何度もハッシュを取らずに、一度作ったハッシュに、公開鍵を付け加えながらハッシュを作り直していく事で、同じ内容（payload）に対するハッシュ計算を何度もやらなくても、その内容を保証することができます。

7. ブロック

「悲観主義者がつまづくだけの石で、楽観主義者は飛び石遊びをする」

- エレノア・ルーズベルト

Symbol のコアシステムはブロックチェーンである。ブロックチェーンとは、順序が決められたブロックの連なりと定義される。シンボルのブロック構造について知ることで、Symbol プラットフォームの可能性を理解できるようになる。

ブロック内のレイアウトは、アグリゲートトランザクション（6.2 アグリゲートトランザクション参照）のそれと良く似ている。ブロックは、アグリゲートトランザクションと同様に、検証不要なヘッダー（脚注；確認しておくと、このヘッダーは検証可能なブロックヘッダー全体のことを指しているのではない。検証可能なヘッダーの前に置かれる署名フィールドなどのことを指している）を持ち、それを使って検証作業をするようになっている。さらに、ブロックヘッダーの後ろには検証不要なフッターとトランザクションデータが続く。アグリゲートトランザクションと異なり、ブロックはベーシックトランザクション（アグリゲートトランザクションで使われる埋め込みトランザクションではなく）が連続していて、ブロックに含まれるそれぞれのトランザクションは、ブロック署名者とは独立に署名されている（脚注：アグリゲートトランザクションにおいては、トランザクションを生成したアカウントが署名することで有効なものとみなされる。ゆえにブロックの署名者はそれぞれのトランザクションについてはあらためて署名する必要がない）。この仕組みにより、有効性が認められた任意のトランザクションが、任意のブロックに含まれることを可能にする。

検証不要なフッターフィールドは、あえて検証される必要はない。パディング用のリザーブドバイトは、今回も特に意味は無く、桁揃えのみに用いられる。

ここはわかりにくいですね。アグリゲートトランザクションは、トランザクション生成者が署名し、ピアノードによって検証済みなので、ブロック署名者がアグリゲートトランザクションの有効性を重ねて検証する必要はないという内容だと思います。また、ブロックに含まれるトランザクションは、独立なベーシックトランザクションの連なりとみなされ、ノードはその組み合わせや順番を自由に変えることができるとういことだと思います。

7.1 ブロックフィールド

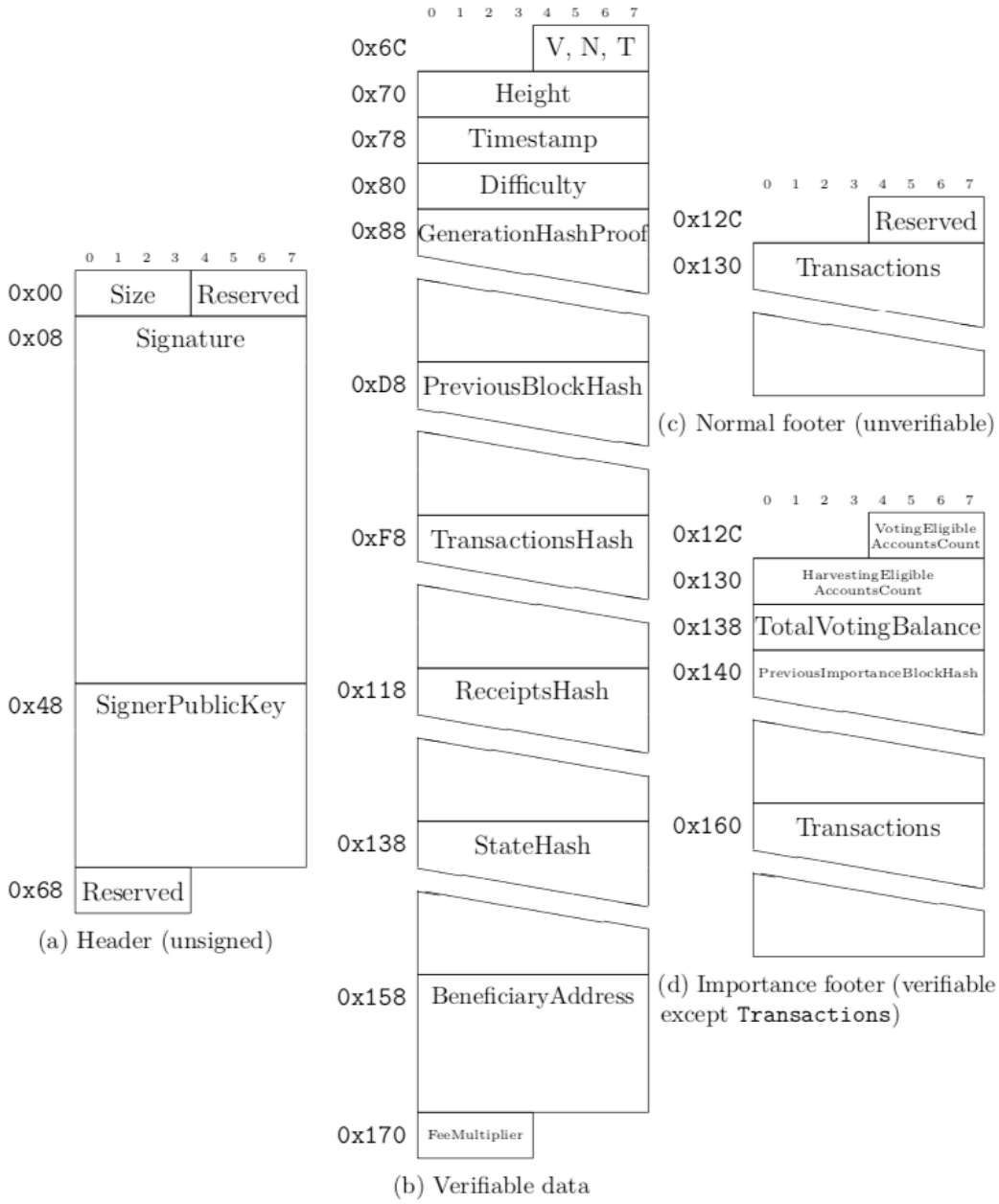


図 14：ブロックヘッダーのバイナリー構造（d のインポータンスフッターが 0.10.0.4 で追加）

ブロック高（height）は、ブロックの連番である。最初のブロックはネメシスブロック（一般にはジェネシスなんですが、ネムだからネメシスとしたのは賛否両論）と呼ばれ、ブロック高「1」を持つ。それに続くブロックは前のブロックのブロック高に 1 を足す。

タイムスタンプ（timestamp）は、ネメシスブロックからの時間経過を示し、ミリ秒単位で計算される。すべての連続したブロックは、ひとつ前のブロックよりも大きな値を持つタイムスタンプを持たなければならない。なぜなら、ブロック時間というのは厳密に増加し続けなければならないからである。それぞれのネットワークノードは協力して、ブロック生成の時間差を、目標値（target block time）にできるだけ近づけるように調節する。

難度（difficulty）は、一つ前のブロックに新しいブロックを追加する承認をするハードルについて定義する。これについては、「8.1：ブロック難度」で詳細に記述する。

GenerationHashProof は、ブロックハーベスターの VRF 秘密鍵によって作られた VRF 証明である。それは、32 バイトのγ、16 バイトの検証ハッシュ（c）、そして 32 バイトのスカラー値（s）から構成される（詳細は 3.3: 検証可能なランダム関数（VRF）を参照）。このフィールドは、次のハーベスターを予測不可能にするのに使われる（8.3: ブロック生成を参照）。

前ブロックハッシュ（PreviousBlockHash）は、一つ前のブロックのハッシュ値である。ブロックにこのハッシュ値を含めることで、ブロックチェーンに含まれるすべてのブロックが時間軸に沿って決定論的につながっていることを保証できる。

トランザクションハッシュ（TransactionsHash）は、ブロックに含まれるトランザクションの個々のハッシュ値から作ったマークルルートハッシュである（脚注：このフィールドはアグリゲートトランザクションのそれとまったく同じ意味を持つ）。このハッシュ値を計算するために、それぞれのトランザクションのハッシュ値を順番に並べてマークル木を作る。その帰結点となるルートハッシュが、ここに書き込まれる。

レシートハッシュ（ReceiptHash）は、ブロックを処理する際に生成されるレシートのハッシュ値から作ったマークルルートハッシュである。ネットワークが network:enableVerifiableReceipts 値をセットしていない場合には、このフィールドはすべてのブロックで 0 になるはずである（7.2: レシート参照）。

状態ハッシュ（StateHash）は、ブロックを処理した後のブロックチェーンの状態をハッシュ化したものである。ネットワークが network:enableVerifiableState 値をセットしていない場合に

は、このフィールドはすべてのブロックで 0 になるはずである。セットされていれば、「7.3: 状態ハッシュ」に記述された方法で計算される。

受益者アドレス（BeneficiaryAddress）は、ネットワークの network:harvestBeneficiaryPercentage 値がセットされている場合に、ブロック生成手数料の一部を受益する権利を持つ「受益者」のアカウントである。このアカウントは何でもよく、たとえまだ登録されたことのないものであっても良い。このフィールドは、新しいブロックをハーベストした「ノード」の所有者によって記入される。もしも、そのノード所有者自身のアカウントがセットされていれば、ブロックに含まれるすべてのトランザクション手数料の一部を、そのノード所有者が回収できる。つまりノード所有者が、たくさんの委任ハーベスターを集めてハーベストするインセンティブとして働く。

手数料係数（FeeMultiplier）は、ブロックに含まれる個々のトランザクションサイズあたりの実効手数料を規定する。node:minFeeMultiplier（最低手数料係数）はノード所有者がセットすることができ、手数料収入を最大化するか、ブロックに含める承認トランザクションの数を増やすかなど、ノードを特徴付けることができる。ブロック B が、あるトランザクション T を含むとき、実効トランザクション手数料は以下の式によって計算される。

$$\text{effectiveFee}(T) = B\text{::FeeMultiplier} \cdot \text{sizeof}(T)$$

もしも、この実効トランザクション手数料が、MaxFee（図 9 などより、署名者が書き込んだ最大手数料であると思われる）よりも大きければ、トランザクション署名者は差額分をキープする（キープの意味がわからないですが、とりあえず得したり損したりがあるということで）。トランザクション署名者から徴収される金額は、実効トランザクション手数料によってのみ決定されハーベスターに分配される。より詳しい情報は、「8.3 ブロック生成」の項を参照してほしい。

ここで、NIS1 にはなかった考え方がいくつかでてきます。まずレシートハッシュと、状態ハッシュというのが出てきました。次の項で説明されますが、ブロックチェーンの状態が変わったことをノードやウォレットに伝えます。次にブロックの「受益者」です。Symbol ではノード設定に受益者のアドレスを記述することで、委任ハーベスターによって発生したブロック報酬の一部をそのアカウントに送ることができます。ノードごとの手数料係数という考え方も新しくて、大きくしすぎると、ノードがブロックに含められるトランザクションが減り（支払える手数料が

高いトランザクションだけ取り扱う高級ノード）、小さくすればトランザクションを大量に含めることができる（手数料が安くて済む庶民ノード）ため、自分のノードにどのような性格付けをするかを選べます。

7.1.1 インポータンスブロックフィールド

インポータンス計算の単位となるブロックには、検証可能データを含む拡張フッターが存在する。この情報が、ブロック生成プロセス後にブロックチェーンの状態遷移を反映する（脚注；その結果 network:votingSetGrouping は、network:importanceGrouping の整数倍になる）。

VotingEligibleAccountsCount というのは、次のインポータンスグループ（計算するブロック高？）までをカバーするファイナライゼーションエポックに投票権を有するアカウント数である。これは厳密に、投票に必要な条件を満たすアカウントのみを含む正確な数である。たとえば、（他の条件を満たしていても）voting 鍵リストを登録していないアカウントは含まれない。

HarvestEligibleAccountCount は、network:minHarvesterBalance 以上の残高を持つアカウントの数である。これは最大予定数の推測値である。実際のハーベスト可能アカウントの数は、これより小さくなるはずである。なぜなら、（残高以外の）ハーベスト条件を満たしていないアカウントも含まれているからである。たとえば、VRF 鍵の登録をしていないアカウントなども含まれる。

TotalVotingBalance は、次のインポータンスグループまでをカバーするファイナライゼーションエポック投票に参加するアカウントの合計残高である。この数値は、ファイナライゼーション証明の検証に関するトラストレスの度合いをあらわす。つまり、ファイナライゼーションに参加する投票者によって保有される投票ステーク（残高）が、全体の残高に占める割合を計算するのに使え、いわばトラストレス保証度数（原文では authoritative denominator）である。

PreviousImportanceBlockHash は、前のインポータンス計算ブロックのハッシュ値である。これは PreviousBlockHash よりも長い周期で現れ、ブロックチェーンの中でブロックが決定論的に接続されていることを保証する、もうひとつのハッシュチェーンを形成する。これは、高速なトラストレス同期プロトコールを可能にし、ノードは全てのブロックヘッダーをダウンロードする代わりに、インポータンスブロックヘッダーのみをダウンロードして検証すれば良い。

7.2 レシート

ブロック生成を実行する間に、ゼロ個またはそれ以上のレシートが生成される。レシートは、（ウォレットなどの）クライアントへの何らかの働きかけがきっかけとなって起きる、状態遷移についての情報発信（コミュニケーション）である。つまり、状態遷移という複雑なブロックチェーン内の変化を、クライアントアプリなどが簡単に知ることができるようにする。

例えば、ネームスペースの期限切れについて考えてみよう。ネームスペースが作られた時、何ブロック先に期限切れになるかが指定されており、そのブロックに到達した時には期限切れのトリガーが生じる。しかし、このトリガーはブロックチェーン内で確かに生じるのだが、実際の期限切れイベントが起きた時のブロックには、何の状態遷移情報も記録されない（つまり、レシートとは、過去のブロックに記録されたネームスペースの期限情報を監視して、期限が切れた時に自動的に情報を発行するしくみであって、その発生事実はブロックに記録されない）。レシート無しでは、ネームスペースに関係するクライアントが、期限切れまでブロック高を監視する必要がある。レシートがあれば、クライアントは期限切れレシートの発行を（サブスクライブして）モニターしておけばよくなる。

もう一つの例として、ハーベスト報酬を考えてみよう。レシートは、メインの—委任ではない—ハーベストアカウント報酬と、それに付随する受益者報酬に関して発行される。このレシートは、インフレーション（スーパーノード報酬など資産保持用アカウントから放出されるXYM など）の金額についても通知できる。

レシートは 3 つのタ異なるイプに分類され、発行元と照合される。3 つのタイプとは、トランザクション、アドレス照合、モザイク照合である。

7.2.1 レシート発行元（Receipt Source）

ブロックの各部分が処理される時、2 つのインデックス（Primary と Secondary Id）によって発行元（Receipt Source）の特定がなされる。発行元（0, 0）は、ブロックによって引き起こされるイベントであることを示しており、ブロックの中にトランザクションがいくつ含まれるかについては影響されない。下の表から推測すると、（x, 0）はレシートを発行するトランザクションのブロック内での位置情報であり、（x, y）はブロックの x バイト目からはじまるアグリ

ゲートトランザクション内の y バイト目からはじまる埋め込みトランザクションによって発行されたレシートという意味になるのではないかと思います。

発行元	プライマリーId	セカンダリーId
ブロック トランザクション 埋め込みトランザクション	0 ブロック内でのバイト位置（1 から開始） ブロック内でのアグリゲートトランザクションのバイトインデックス（1 から開始）	0 0 アグリゲートトランザクション内でのバイトインデックス（1 から開始）

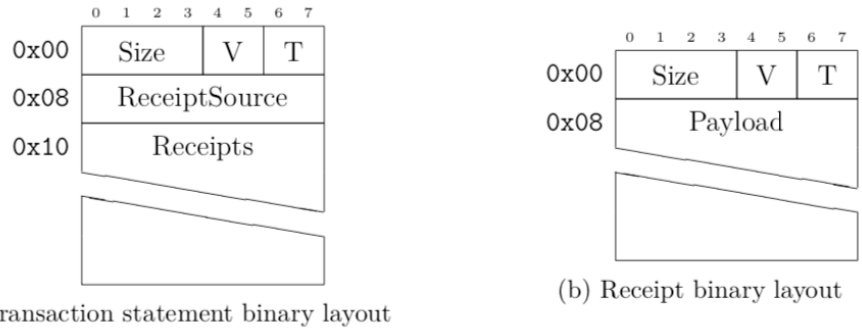
表 3：レシートの発行元を示すインデックス値

注；インデックスは、起点からのバイト数を使った位置情報と解釈しました。

レシートは、イーサリアムのスマートコントラクトで使われる手法のひとつです。もともとは、イーサリアムに登録された DApp アドレスに、何らかの実行命令を投げた時、その結果を返してくれるのがレシートです。でも、Symbol には DApp という概念はなく、プラグインや機能拡張のような形でアプリケーションが登録されています。この章では、残念ながらレシートについてはいつ、誰が、どのように発行するかが詳しく記述されていません。ブロックチェーンの状態遷移（ネームスペースの期限切れやハーベスト報酬発生など）がレシートとして自動的に発行され、ウォレットはサブスクライブして結果を待つことでアカウントの状態が変化したことを知るといった仕組みに使われているようです。ブロックチェーンには、トランザクションハッシュ以外に、レシートハッシュも記録されているので、特定のブロックで発行されたレシートについて、発行時にはブロックに保存されるものと思われます。イーサリアム関連の情報にもレシートについてわかりやすく記述されているものがなく、正直もう少し読み込んでいかないと分からない部分です。僕が、かなり間違った理解をしている可能性もありますので適宜修正していきます。

7.2.2 トランザクション明細書

トランザクション明細書は、発行元が同じであるレシートを束ねて作られる。それぞれの明細にはレシート発行元と、ひとつかそれ以上のレシートが記述されている。その結果、それぞれの発行元が、レシートを発行した場合には、必ず対応するトランザクション明細書を作ることになる。



(a) Transaction statement binary layout

(b) Receipt binary layout

図 15：トランザクション明細のレイアウト（a; トランザクション明細の構成、b; レシートの構成）

トランザクション明細書のデータは、パディング（桁揃え）されない。なぜなら、ブロックが処理されているプロセス中に書き込まれるだけで、読み取られる事はないからである（桁揃えは読み込みの効率は上げるけれども書き込みの効率はむしろ下げる）。つまり桁揃えをすることには、サーバーのパフォーマンス上のメリットが無い。トランザクション明細のハッシュは、計算可能なサイズフィールドを除いた部分（V, T 以降）について計算される。

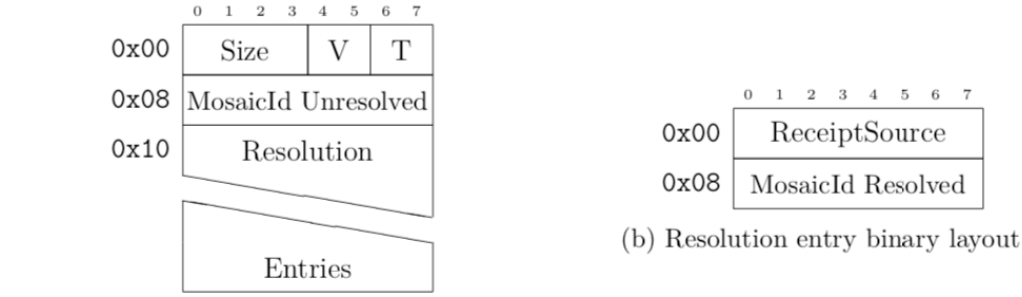
ブロックなどが発行するレシートを束ねて作るのが明細書（ステートメント）です。この明細書のハッシュを集めてマークルツリーを作り、そのルートハッシュをブロックに書き込むわけですが、パディングに関する記述を見る限りやや使い捨てっぽい雰囲気がありますね。同じ発行元（ReceiptSource；表 3 から、ブロック、トランザクション、アグリゲートトランザクションのどれか）から出たレシートの集まりがトランザクション明細書であることが分かります。また、表 3 に出てきたレシート発行元を示す ReceiptSource がプライマリー4 バイト、セカンダリー4 バイトで構成されている事もわかります。これから、ブロック中やトランザクション内で、起点から何バイト目からデータが始まるかを示す位置情報であることが予想できます。

7.2.3 解決明細書

解決明細書は、エイリアス（ネームスペース）の解決結果を調査するのに非常に便利に使える。たとえひとつのブロックの中でエイリアスの参照先が変更されたとしても、クライアントが常にそれを解決できるようになる。理論的には、未解決のエイリアスが、1 ブロックの中に 2 つ存在したとしても、それぞれの使いみちが違っていれば、きちんと異なる値（モザイクま

たはアドレス）に行き着くことができる。そのために、明細書は未解決の（モザイクやアドレス）値と、それを解析するのに必要なひとつまたはそれ以上の解決結果を含んでいる。

明細書には 2 種類あり、アドレスとモザイクを参照する 2 タイプのエイリアスに対応する。図 16 は、モザイクの解決明細書のバイナリーレイアウトであるが、アドレスに関する明細書のレイアウトもこれに準拠している。違いは、解決後の値が、モザイク ID なら 8 バイト（0dc67fbe1cad29e3 など）で、アドレスなら 25 バイト（図 8 で最後の 00 も含む 25 バイト？）であることくらいである。



(a) Mosaic resolution statement binary layout

(b) Resolution entry binary layout

図 16：モザイク解決明細書のレイアウト

(a) モザイク解決明細のバイナリーレイアウト (b) 解決結果のバイナリーレイアウト

MosaicId Unresolved がネームスペース id で、MosaicId Resolved がモザイク id を指すと思われます

解決明細書のデータにも、桁揃えはない。なぜなら、ブロックの生成中には書き込まれるのみであり、読み出しは行われないからである。トランザクション明細同様に、桁揃えすることにサーバー負荷軽減のメリットはない。解決明細書のハッシュは、すべての明細データからサイズフィールド（他のデータから計算可能な検証不要フィールド）を取り除いたもので生成される。

この解決明細書は、エイリアス解決の操作があったときだけ生成されるという点が重要である。つまり、エイリアスが登録されたり、変更されたブロックでは、そのブロック内で「使用」されなければ、エイリアス解決はなされないため明細書も生成されない。一方、エイリアスが使用されるために解決されれば、それがなされた時のブロックに解決明細書が含まれることになる。

7.2.4 レシートハッシュ

ブロックのレシートハッシュを計算するために、ブロックに含まれるすべての明細書が集められる。そしてすべての明細書のハッシュからマークル木が構成されるが、その順番は以下のようになる。

1. レシートソース順でソートされたトランザクション明細のハッシュ
2. 解析前のアドレス順でソートされたアドレス解決明細のハッシュ
3. 解析前のモザイク id 順でソートされたモザイク解決明細のハッシュ

ネットワークの全体設定に network:enableVerifiableReceipts がセットされていれば、このようにして生成されたマークル木のルートハッシュが、ブロックのレシートハッシュになる。クライアントは、ある明細が、特定のブロック内で生成されたかどうかを確かめるためには、マークル検証すればよい。

シンボルでモザイク id やアドレスとネームスペースを紐付ける機能ができて便利になったと思っていましたが、ここでもレシートが大活躍なんですね。あるモザイクにネームスペースを紐づけたら、そのブロックにトランザクションが書き込まれるだけと思っていました。でも実際は、ネームスペースを使ってモザイクを送金するとエイリアス解決がされ、それが利用されるとレシートが発行されて、結果が明細書としてブロックに書き込まれるようです。ネームスペースやモザイクには使用期限があり、モザイクに紐付けられたネームスペースも 1 つとは限らない（https://nemtech.github.io/ja/concepts/namespace.html）ため、あるネームスペースがどのモザイクを参照しているのかが、わかりにくくなりました。もしレシートと明細書がなければ、モザイクが送られてくるたびに、過去のブロックをさかのぼって、エイリアス解析をしないとならくなるでしょう。ノードはそれを見越してレシートという形で保存しているとも言えますね。レシートの詳細は開発者文書（https://nemtech.github.io/ja/concepts/receipt.html）に詳しく記載されています。また、Planethouki さんがレシートハッシュを実際に求めてみた Qiita 記事（https://qiita.com/planethouki/items/ed45018267b24d262b84）を書いてくれていてすごく参考になりました。ShizuiNet にとっても大切なネームスペースですので、きちんと理解したいところです。

7.3 状態ハッシュ

Symbol は、ブロックチェーン全体の状態を、複数のタイプの状態保持レポジトリに保管している。例えば、アカウント状態は、あるレポジトリに保管され、マルチシグ状態は、それとは別のレポジトリといった具合である。それぞれのレポジトリは、単純なキー値として保管されている。ネットワーク内に存在する特定のレポジトリは、ネットワークが許可したトランザクションプラグインによって定義されている。

ネットワークが network:enableVerifiableState オンの状態で立ち上げられている時、それぞれのレポジトリごとにパトリシアツリーが作られる。パトリシアツリーは、1 つのハッシュにまとめられ、ハッシュはそれぞれのレポジトリの状態の指紋のようなものである。同様に、N 個のレポジトリと、それに対応する N 個のハッシュが、ネットワーク全体の状態の指紋として扱われる。

この時、N 個のハッシュすべてをブロックヘッダーに収納することもできるが、あまり好ましいことではない。それぞれのブロックのヘッダーはできる限りコンパクトであるべきである。なぜならば、すべてのクライアント（ウォレットなど）が、最低限、すべてのヘッダーがネメシスブロックにつながることが検証できる形で、同期していないとならないからである（つまり、すべてのユーザーがヘッダー部分だけでも読み取ろうとするので、ここが大きいとそのままネットワーク負荷が跳ね上がるのだと思います）。また、ある機能を付け加えたり、取り除いたりすることで、レポジトリーの数（N）は変化する。それに伴い、ブロックヘッダーの構造も変化してしまう。

そのため、すべてのレポジトリルートハッシュは結合され（脚注；順番はレポジトリーid によって決められる）、そのハッシュが状態ハッシュとして計算される。このハッシュによって、ブロックチェーン全体の状態の指紋を取るのである。

$$
\begin{aligned}
RepositoryHashes &= 0 \\
RepositoryHashes &= \sum_{0 \leq i \leq N} \mathrm{concat}(RepositoryHashes, RepositoryHash_i) \\
StateHash &= \mathrm{H}(RepositoryHashes)
\end{aligned}
$$

7.4 拡張レイアウト

この章に示したブロックのレイアウトは、一部を単純化している（脚注；アグリテートトランザクションの拡張レイアウトと同様）。フッターに含まれるすべてのトランザクションは桁揃えをされ、8 バイトごとの区切りにフィットするようになっている。これによって、すべてのトランザクションも 8 バイトの区切りで開始されることが保証される。これらの桁揃え用のデータ列は署名には含まれず、その結果ハッシュにも含まれない。

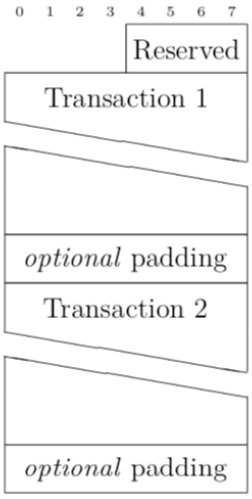


図 17：桁揃えされたブロックのトランザクションフッター

ここまでで、ブロックのおおまかな構造や、レシートハッシュ、状態ハッシュなど、新しいハッシュ群の説明は終わりです。NIS1 に比べるとずいぶん複雑になっていますが、これらの定義を理解していれば、ブロックのデータを効率よく読み取って、高速に情報を取り出すことも可能でしょう。

7.5 ブロックハッシュ

それぞれのブロックは、そのブロックを代表するいわゆる実体ハッシュ（entity hash）を持っている。このハッシュ値は、ブロックごとにユニークな値であり、同じブロックが複数回検証される事などを防いでいる。

ブロックの実体ハッシュは、以下のデータをハッシュ化したものである：

1. ブロックの署名値（Signature）
2. ブロックを署名したアカウントの署名公開鍵（SignaturePublicKey）
3. ブロック（ヘッダー）の検証可能データ

ブロックの実体ハッシュの計算方法は、トランザクションのものと似ている。一箇所異なっているのは、ハッシュ計算過程にネットワークのジェネレーションハッシュシード（GenerationHashSeed）が含まれていないことである。トランザクションハッシュと違って、このシード値をブロックに含めることは無意味だからである。すべてのブロックは少なくともひとつ、他のネットワークでは使われていないトランザクションを含んでいるはずであり、そのトランザクションハッシュはネットワークごとに異なっている。つまり、ブロックのハッシュ（TransactionHash）も、必ず違うものになることが保証されている。

ここもちょっとわかりにくいですが、ネットワークごとに異なる GenerationHashSeed をわざわざ各ブロックに含めなくてもハッシュが衝突したりはしないということを言いたいのではないかと思います。トランザクションの中身は異なるネットワークでかぶることはあるけれど、ブロックの内容全部がネットワーク間で重複することは絶対にありえないということで、GenerationHashSeed が除かれているわけです。

8. ブロックチェーン

「遠くを見すぎるのは誤りである。運命の鎖は、たった一つの輪を手繰り寄せるのがやっとのものなのだから」

- ウィンストン・チャーチル

Symbol は、ブロックを連結したブロックチェーンと呼ばれる公開台帳を中心に動いている。完全なトランザクションの記録が、ブロックチェーンには書き込まれている。すべてのブロック、そしてそこに書き込まれたトランザクションは、決定論的に、そして暗号論的なルールで順番を決められている。ひとつのブロックに含まれるトランザクションの最大数は、ネットワークごとに異なる数値に設定できる。

8.1 ブロック難度（difficulty）

ネメシスブロックは、ブロック難度の初期値として 10^{14} という数値を持つ。その後のブロックの難度は、下限が 10^{13}、上限は 10^{15} の間に制限される。

新しいブロックのブロック難度は、直近に承認された複数のブロックの難度と時刻（timestamp）によって決められる（実際には、時刻と時刻の間のブロック生成にかかった時間）。いくつ前のブロックまで考慮するかは、ネットワーク設定で変更できる。

network:maxDifficultyBlocks が満たせない時（主にブロックチェーン開始直後で、必要なブロック数が確保できない場合）は、可能な限り多くのブロックが考慮の対象となる。その他の場合は、直近の n 個のブロックから、以下のようにして難度（difficulty）が計算される。

$$d = \frac{1}{n}\sum_{i=1}^{n}(\text{difficulty of } block_i) \qquad \text{(average difficulty)}$$

$$t = \frac{1}{n}\sum_{i=1}^{n}(\text{time to create } block_i) \qquad \text{(average creation time)}$$

$$difficulty = d\,\frac{\text{blockGenerationTargetTime}}{t} \qquad \text{(new difficulty)}$$

d は平均難度、t は平均ブロック生成時間。これらを元に次の difficulty が計算される。

この計算式により、目標とする network:blockGenerationTargetTime に、ブロック生成時間が近づくようになっている。

もしも、新しいブロック難度が、前回より 5%より増加または減少する場合は、変化量は 5%に制限される。変化量を 5%以内に制限することによって、全体の 50%にまったく満たないようなインポータンスしか持たない攻撃者が、秘密裏に都合よくブロックチェーンを伸ばすのを防いでいる。ブロック難度は、その時点でハーベストに参加しているアカウントの合計インポータンスとだいたい比例するため、攻撃者の持つ合計インポータンスはそれに比べれば低いものになる。攻撃者の秘密チェーンのブロック難度は、すぐには減少せずに、ブロック生成時間は長くなって、メインチェーンに大きく遅れを取ることになる。ブロック難度が、攻撃者の持つインポータンスに適した値になり（結果として図 18 の例でのブロック生成時間が 15 秒以下になったとしても）、その頃にはメインチェーンはずっと先に進んでしまっているだろう。（つまり、もし攻撃者がネットワークを意図的にフォークさせて、自分の持つインポータンスを使ってブロックを速く伸ばそうと思っても、）ブロック生成時間は、攻撃者の秘密のブロックチェーンの開始時において、network:blockGenerationTargetTime を大きく上回ってしまうはずである。

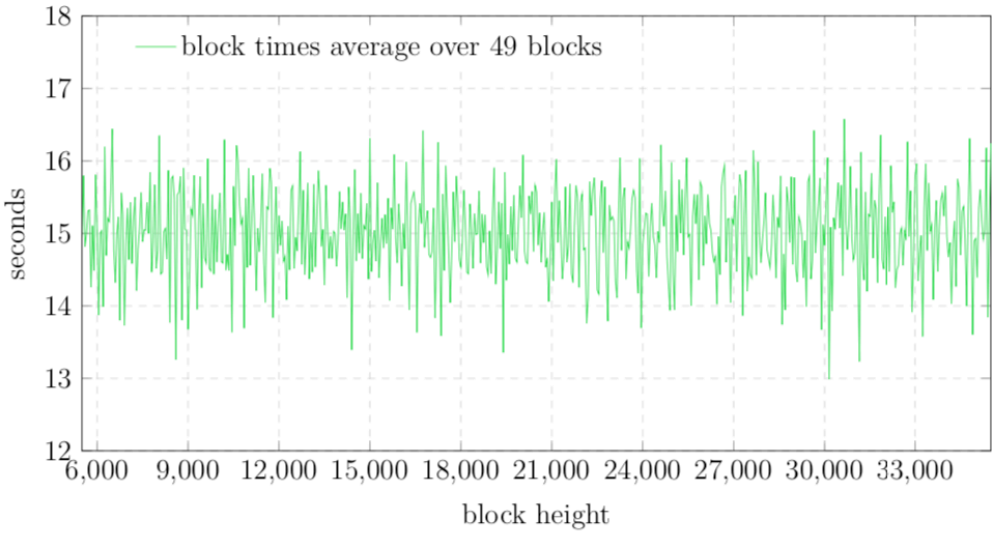


図 18: 開発中のブロックチェーンにおける平均ブロック生成時間。この例でのターゲットブロック時間は 15 秒。

ブロックの生成難度を大きく変化させない 5%制限は、NIS1 でも採用されていました。ここでは、フォークした裏チェーンが、十分なインポータンスを集めることができないので、攻撃側のブロック生成時間が大きく延びてしまうことにより、メインのチェーンが守られます。さらに、

VRF 導入によって意図的なフォークはほぼ不可能になり、Symbol は不正フォークを防ぐ 2 つの仕組みを持つことになりました。

8.2 ブロックスコア

ブロックのスコアは、ブロック難度（difficulty）と、1 つ前のブロックが生成されてから、そのブロック生成までにかかった時間（秒）から計算される。

$$score = difficulty - time\ elasped\ since\ last\ block \qquad \text{(block score)}$$

ブロックスコアは、ブロック難度から、ブロック生成時間を引いたもの。不正なフォークではブロック生成時間が大きく伸びてしまうため、スコアが一時的に大きく下がることになり、メインチェーンに追いつくのがほぼ不可能になります。

8.3 ブロック生成

新しいブロックを生成する工程を、ハーベストと呼ぶ。ハーベストするアカウントは、そのブロックに含まれているトランザクション手数料の大部分を得ることができる。これによって、ハーベストへの、インセンティブが与えられ、できるだけ多くのトランザクションをブロックに含むモチベーションになる。

ブロックごとの手数料は、3 つに分割される。network:harvestBeneficiaryPercentage がゼロでなければ、ハーベストノードが指定した受益者（beneficiery）のアドレスにその割合分が送金される。また、network:harvestNetworkPercentage がゼロでなければ、その割合は network:harvestNetworkFeeSinkAddress というシンクアドレスに送金される。このシンクアドレスはネットワークが自己継続的であるために、たとえば投票ノードへのインセンティブ報酬などに使うことができる。これらの割合分を差し引いた残りが、ハーベストしたアカウントへの報酬となる。

アカウントは、以下の条件をすべて満たした時に、ハーベストする権利を与えられる。

1. インポータンス計算がされた直近のブロック高（height）におけるインポータンススコアが、ゼロではないこと。

2. ネットワークによって設定されている network:minHarvesterBalance（最低ハーベスト残高）を上回る残高があること。
3. 同様に、残高が network:maxHarvesterBalance（最大ハーベスト残高、脚注；この数値は、原則としてコアファンドや取引所のアカウントがハーベストに参加するのを防ぐために設けられている）未満であること。
4. VRF 公開鍵が、アカウントに設定されていること。

アカウント所持者は、残高を持つアカウントの秘密鍵を外部に漏洩しないために、他のアカウントにインポータンスを委任することができる（脚注；詳細は https://nemtech.github.io/concepts/harvesting.html#accountlinktransaction 参照）。

ハーベストアカウントが得る実際の手数料報酬は、ネットワークの設定によって決められる。もしインフレーション（inflation）の設定が 0 以外の数値になっていれば、それぞれのブロック生成に対してインフレーションブロック報酬が上積みされる。これによって、ハーベストの利益率が上がる。もし、ハーベスト受益者分配が許可（network:harvestBeneficiaryPercentage がプラス）されている場合には、ノードを運用しているアカウントによって報酬の一部が回収される。これは、ノードを運用することへのメリットとなるが、ハーベストするアカウントにとっては報酬の減額となる。

それぞれのブロックには、手数料率（fee multiplier）を設定しなければならない。これはブロックに含まれるすべてのトランザクションによって支払われる有効手数料（effective fee）を決めるものである。つまり、ハーベストするノードの所有者が node:minFeeMultiplier（最小手数料率）を設定すれば、その料率はそのノードによってハーベストされるすべてのブロックに適用される。下記の式を満たすトランザクションだけが、そのノードの未承認トランザクションキャッシュにエントリーでき、ノードがハーベストする時に、ブロックに書き込まれる：

$$\textit{transaction max fee} \geqslant \text{minFeeMultiplier} \cdot \textit{transaction size (bytes)} \qquad (12)$$

トランザクション最大手数料設定値　≧　ノードの最小手数料率　x　トランザクションサイズ（バイト）

あるノードで拒否されたトランザクションでも、より低い手数料率を設定した他のノードによってハーベストされるブロックに含まれることができる。ノードがどのようなトランザクションを選択するかについては、node:transactionSelectionStrategy 設定に書き込んでおけば良い。Symbol では 3 種類の設定の中から選ぶことができる（脚注；下記のどのケースにおいても、前提としてノードが設定する node:minFeeMultiplier によって計算される手数料を支払わなければならない）：

1. 最も古いものから（oldest）：このセッティングは、最もリソース負荷が小さいストラテジーである。なので、ネットワークが時間あたりに多くのトランザクションを捌かなければならない場合（高 TPS）に推奨される。トランザクションは、受け取られた順番で選ばれていき、期限切れになる可能性を最小化する。よって、ハーベストする側にとっては、利益を最大化する事にはならない。
2. 手数料最大化（maximize-fee）：ひとつのブロックに含まれるトランザクションの合計手数料が最大になるようにしたい場合のセッティング。手数料利益を追求するノードがこのストラテジーを選択する。ブロックの合計手数料を最大化することは、必ずしも最大数のトランザクションを含むことにはならない。実際、多くのケースでは、ネットワークに投入されたトランザクションの一部だけをブロックに書きこんでハーベストをおこなうようになるだろう。
3. 手数料最小化（minimize-fee）：最も少ない最大手数料率（maximum fee multipliers、6. トランザクションの図での MaxFee）を設定されたトランザクションを優先して選択する、利他的なノードのストラテジー。そのようなノードは、とても低い node:minFeeMultiplier を設定しているはずである。もしこの node:minFeeMultiplier をゼロに設定すれば、MaxFee が 0 にセットされたトランザクションが最初にハーベストされるだろう。つまり、トランザクション手数料無料でトランザクションをブロックに書き込むことができる！実際には、このような設定をするのはごく少数のノードに限られるだろう。よって、手数料無料のトランザクションは、ネットワーク内で最も少ないノードによってのみハーベストチャンスが与えられ、ブロックに書き込まれる可能性も低くなる。

トランザクションは、定額での送金だけでなく、動的に変化する価値の送付もブロックチェーンに投入できる。定額送金は、外部の要因に影響されない、固定額の送金である。たとえば、送付トランザクションによる送金は、定額送金になる。トランザクションには、固定された金額が明示され、必ずその額が送金者から受領者に送られる（脚注；

https://nemtech.github.io/concepts/transfer-transaction.html 参照）。一方、変動するトランザクションのコストに連動して、送付量（価値）が変化するトランザクションも存在する。このようなトランザクションは、特別なネットワーク連動要素（artifacts）、つまりネームスペースやモザイクなどを取り扱う時に払う手数料計算に使われたりする。このような連動要素の価値は、市場の影響を受けて変化するため、固定された手数料率は好ましくない。また、FeeMultiplier（トランザクション内に記載する手数料率）だけで決定してしまうと、ゼロ手数料を設定して、それを自分の無料ノードを使ってハーベストすることで、ネームスペースやモザイクを無料発行できてしまう。これは問題であり、変動する料率を導入した。この料率は、network:maxDifficultyBlocks で設定されたブロック数までさかのぼって、FeeMultiplyer の中央値を得て設定する（つまりネットワークで使われている料率に近くなります）。（トランザクションが無いブロックが連続したりして）充分なデータが得られない場合には、0 ではなく network:defaultDynamicFeeMultiplier が用いられる。この調整により、実効手数料は、0 に設定できないようになっている。実効手数料を計算する場合は、基礎料率（バイト数をもとに計算した基準値？）に変動手数料率をかけて計算する。

$$effective\ amount = base\ amount \cdot dynamic\ fee\ multiplier$$

Symbol では、ノードごとにトランザクションバイトあたりの手数料率を設定できます。それぞれのトランザクションにはいくらまで払えるかを MaxFee として書き込みますから、多額の手数料を指定すれば、より多くのノードがそのトランザクションをブロックに書きこもうとしてくれます（手数料額は、実際にハーベストを行ったノードの minFeeMultiplier x バイト数）。一方手数料をまったく払わない場合でも、どこかのノードが最低手数料率を 0 にしてくれていれば、それらのノードによって「いつかは」ハーベストされ、ブロックに書き込まれる可能性があります。ただし、現在の Symbol メインネットではゼロ手数料率のノードはほとんど存在せず、minFeeMultiplier = 10 の低手数料ノードが充分数存在する状態で、低い手数料でもそれほど待たずに承認されるようになっています。また、ネームスペースやモザイク発行まにはネットワークが設定する特別な手数料率をかけて、無料にできないようになっています。

8.4 ブロックジェネレーションハッシュ

ブロックのジェネレーションハッシュ（generation hash, gh）は、前のブロックのジェネレーションハッシュと、ブロックに書き込まれた VRF 証明から、新しいハッシュを生成する：

$$\mathrm{gh}(1) = \mathrm{generationHash} \qquad \text{(generation hash)}$$
$$\mathrm{gh}(N) = \mathrm{verify\_vrf\_proof}(\mathrm{proof}(\mathit{block}(N)), \mathrm{gh}(N-1), \mathit{VRF\ public\ key\ of\ account})$$

ジェネレーションハッシュを、VRF によって充分にランダムで予測不能な値にすることができる。他のすべての情報を使ったとしても、次のハーベストアカウントを予測できないようにするのである。ブロックがネットワークに（承認のために）送り込まれるまで、ジェネレーションハッシュはハーベストアカウント以外にとって不明な値になる。そうすることで、最良の（最高のブロックスコアを持つ）ブロックがネットワークに投入されるまでは、次のジェネレーションハッシュを予測することはできないことになり、つまり、前のブロックが確定する時点まで次のブロックのための計算はできないことになる（あたりまえのようですが、VRF 鍵導入までは予測ができる仕様でした）。

VRF 公開鍵は、ブロックがハーベストされるまでに登録されていなければならない。そうでなければ、攻撃者によって、前もって VRF のスキャンがおこなわれ、ブロックハーベスト時に最大の hit 値を生み出す余裕を与えてしまうからである。登録制にした結果、攻撃者は複数のブロックを用意して hit 値を上げようとする試みも不可能になる（target 値ではなく hit 値を上げることで、ブロックの難度を上げ、他のアカウントがハーベストできなくする事で、秘密裏にフォークさせたチェーンを伸ばす時間を稼ぐなどの攻撃が考えられます）。

8.5 ブロックの Hit 値と Target 値

そのアカウントが、ある時刻（t）にハーベストする権利を持つかどうかの判定には、以下の 2 つの数値の比較をする。

- hit: ブロックごとの目標値
- target: ハーベストアカウントごとの力量（攻撃力みたいな感じ）。前のブロック生成時刻から経過した時間とともに増大する。

hit < target となった時に、そのアカウントはブロックを生成することができるようになる。target 値は、時間に比例して増加するため、すべてのアカウントが高い hit 値を持つジェネレーションハッシュしか作れずなかなかハーベストに到達しない場合でも、いつかはブロックは生成される。

委任ハーベストの場合は、委任先のアカウントの代わりに、委任元（オリジナル）アカウントのインポータンス値が用いられる。

target の計算方法は以下のとおりである（脚注；丸め誤差の問題を避けるため、この計算には浮動小数点演算ではなく 256-bit 整数演算を用いる）：

$$
\begin{aligned}
multiplier &= 2^{64} \\
t &= time\ in\ seconds\ since\ last\ block \\
b &= 8999999998 \cdot (account\ importance) \\
i &= total\ chain\ importance \\
d &= new\ block\ difficulty \\
target &= \frac{multiplier \cdot t \cdot b}{i \cdot d}
\end{aligned}
$$

ブロック生成時間の急激な変化を緩和するには、設定によって以下の計算を追加することもできる（脚注；丸め誤差を避けるため、この計算には浮動小数点ではなく 128-bit の固定小数点演算を用いる。上位 112bit は整数部分を表し、下位 16bit が小数点以下を表す。log2(e)は、14426950408/10000000000（約 1.44 くらいです）で近似する。power 値が一定値を下回って（smoothing が）大きな負数になってしまう場合は、smoothing は 0 に設定する）。；

$$
\begin{aligned}
factor &= \text{blockTimeSmoothingFactor}/1000.0 \\
tt &= \text{blockGenerationTargetTime} \\
power &= factor \cdot \frac{time\ in\ seconds\ since\ last\ block - tt}{tt} \\
smoothing &= \min\left(e^{power}, 100.0\right) \\
multiplier &= \text{integer}\left(2^{54} \cdot smoothing\right) \cdot 2^{10}
\end{aligned}
$$

一方、hit は $2^{54}\ |ln(gh/2^{256})|$を 64bit で近似した数値である。ここで gh は、（アカウントごとに異なる）新しいジェネレーションハッシュである（脚注；この計算でも、丸め誤差を避けるために、浮動小数点演算ではなく 128-bit の整数演算を用いる。この場合、log2(e)は 14426950408889634/10000000000000000 で近似する）。

まずは、上記の hit 値を計算する式を変形してみよう：

$$hit = \frac{2^{54}}{\log_2(\mathrm{e})} \cdot \left| \log_2\left(\frac{gh}{2^{256}}\right) \right|$$

gh/2^{256} は常に 1 より小さいため（gh はハッシュ値なので 32 バイトの整数＝256-bit の整数値になる。よって常に 2^{256} よりは小さくなる）、対数を取ると必ず負の値になる。また、log2(gh/2^{256})は log2(gh) - log2(2^{256})と書くことができる。

絶対値計算をはずして（常に負なのでひっくり返す）、scale 値を定義して書き換えると：

$$scale = \frac{1}{\log_2(\mathrm{e})}$$
$$hit = scale \cdot 2^{54}(\log_2\left(2^{256}\right) - \log_2(gh))$$

さらに簡略化して：

$$hit = scale \cdot (2^{54} \cdot 256 - 2^{54} \cdot \log_2(gh))$$

実際の計算は、新しいジェネレーションハッシュ（gh）の上位 32-bit（最初の 4 バイトが 00000000 でない場合）を使っている。特殊なケースについては、別に定める。

また、hit 値は指数関数的な分布をしている。よって、インポータンスを多くのアカウントに分散しても、新しいブロック生成にかかる時間には影響がない。

ごちゃごちゃしていますが、total chain importance (i) = 1、block difficulty (b) = 10^{14}、smoothing = 1 で試しに計算してみたところ、

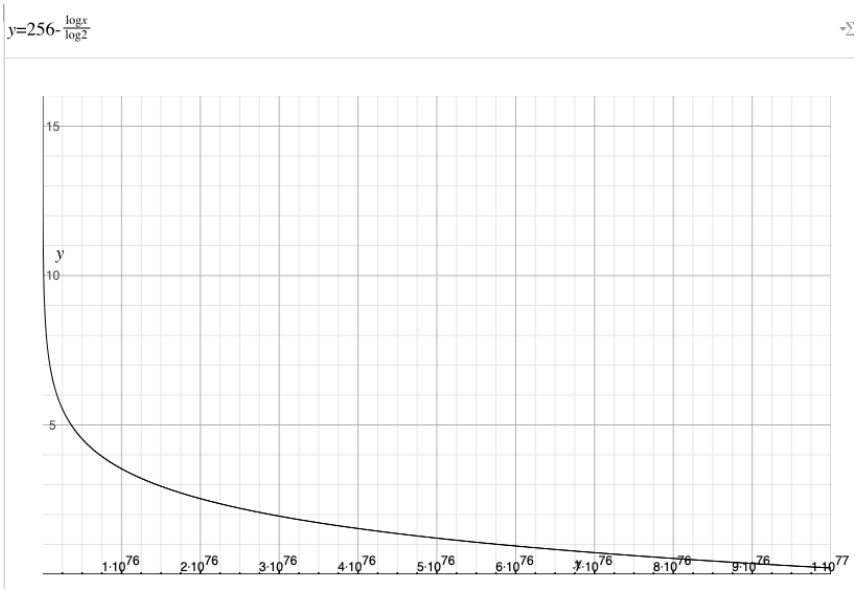
target = 1.66tx * 10^{15} (x = アカウントのインポータンス)

くらいになるようです。前のブロック生成後 10 秒（t = 10)で、自分のインポータンスが x = 0.00001 の時、target は 1.66 * 10^{11} って感じで計算できます。

さて、hit についてもなぜ 2^{54} をかけているのかとか、log2(e)は何のためかなどについての説明は特にありませんので、実際に適当な数値を設定して計算してみましょう。scale = 0.693、2^{54} = 1.8 * 10^{16} と近似して、hit 式は、

$$hit = 1.24(256-\log_2(gh)) * 10^{16}$$

となります。y = 256 - log2(x)のグラフを描いてみると：



横軸（x）は gh を 10 進数に変換した値で最大値は 2^{256}-1（約 1.15 * 10^{77}）になります。ハッシュ値はランダムに生成されると仮定すると、その 9 割は 1.15 * 10^{76} 以上となり、その場合 y は大体 3.5 未満になることが分かります。つまり hit 値の 9 割は 1.24 * 3.5 *10^{16} = 4.34 * 10^{16} 未満になるということです。

仮にインポータンス 1 を持つひとつのアカウントがハーベストをおこなうとすると、

$$target = 1.66t * 10^{15}$$

ですので、26〜27 秒以内には、hit < target を満たして、ハーベストできると思われます。インポータンスを多数のアカウントに分散しても、このブロック生成時間は変化しない（と書いてある）ので、計算時間を含めて 30 秒で新しいブロック生成するのには問題ないように定数設定をしている感じですね。

8.6 委任ハーベスターの自動検出

user:enableDelegateHarvestersAutoDetection 設定がオンになっていれば、サーバーは特別な送付メッセージ（メッセージはウォレットなどのクライアントとノード間で使われる通信手段のひとつです。16 章で詳しく説明されます）を通じて、委任ハーベストのアカウント登録リクエストを受け入れることができる。サーバーは、ノードに設定された公開鍵に向けたすべての送付トランザクションを検査して、ノード証明公開鍵と一致する（委任ハーベストをリクエストしてきた）アカウントをファイルキューに書き込む。定期的に繰り返し実行されるよう設定されたタスクが、すべてのキューに保存されたメッセージをチェックしている。想定されていない、あるいは不正なメッセージは無視され破棄される。有効なメッセージのみが、解読後に処理に回される。

メインアカウント（脚注；下記のすべての条件が満たされていたとしても、ノードオーナーは委任ハーベスティングを許可しないことが可能）は、下記の条件すべてが満たされていた場合に、委任ハーベストを依頼できる。

・メインアカウントが委任ハーベストを要求するトランザクションに署名していること。
・リンクされた委任ハーベスト用の公開鍵が、暗号化された対応する秘密鍵とペアになっていること。
・VRF 公開鍵が、暗号化された VRF 暗号鍵とペアになっていること。
・ノード公開鍵が、ノードの保証用公開鍵と対応していること（12.2：ハンドシェーク参照）。

ファイルキューに記録されたメッセージは部分的に暗号化されることで、攻撃者がリモートハーベストをしているアカウントや VRF 秘密鍵を取り出せないようになっている。暗号化に使用される AES256 GCM は、（対称性）暗号化スキームである。暗号用の鍵は、サーバーノードの証明用鍵ペアと、クライアントによって作成される時限付のランダムな鍵ペアである。

個々のメッセージの内容は以下のものを含んでいる：

名前	サイズ (bytes)	暗号化	内容
0xE201735761802AFE	8	No	ハーベスト要求であることを示すマジックバイト
時限付公開鍵	32	No	対称性暗号化用公開鍵
AES GCM Tag	16	No	暗号データ全体に対する MAC タグ
AES GCM IV	12	No	AES GCM アルゴリズムの開始ベクトル
署名用秘密鍵	32	Yes	リモートハーベスターの署名用秘密鍵
VRF 秘密鍵	32	Yes	メインアカウントとリンクした VRF 秘密鍵

表 4：ハーベスト要求メッセージのフォーマット

もし合格な場合、委任ハーベストアカウントの秘密鍵（表 4 の署名用秘密鍵？）は、そのサーバーのハーベストに利用されることになる。サーバーは、harvesting:maxUnlockedAccounts の数まで委任ハーベストアカウントを受け入れることができる。この制限いっぱいまで受け入れた後は、harvesting:delegatePrioritizationPolicy（委任ハーベスト優先順位ポリシー）の設定に従って、新しいアカウントを受け入れるかどうか判断する。このポリシーが Age（古さ）に設定されている場合は、先着順である。つまり、新しいアカウントは制限数を超えて登録されることはできない。ポリシーが、インポータンスに設定されている場合は、大きなインポータンス値を持つアカウントが優先される。つまり、低いインポータンスを持つアカウントをはずして、代わりに新しい高いインポータンスを持つアカウントが登録できる。

登録許可されたアカウントの登録情報は、harvesters.dat ファイルに保存される。一度許可を受けた委任ハーベストアカウントは、サーバーが再起動されても消えることはない（NIS1 では消えてましたね）。サーバーは、現在受け入れ中の委任ハーベストアカウントを使っているかどうかについて、明示的には確認のアナウンスはしない（インポータンス優先設定の場合は、突然解除もあり得ますから、委任した側は時々生存確認をする必要があるかもしれません）。ブロックチェーンには、委任ハーベストに関係するすべてのリンク（公開鍵とか、VRF 鍵とか）が保存されるが、その時点での活動（ハーベストをおこなっているかどうかに関するリアルタイム情報）は記録していない。

委任アカウントの登録においても、ハーベスト用の秘密鍵や VRF 秘密鍵が暗号化され、その方式がきちんと明記されました。ここで重要なのは、ノード運用者は NIS1 の時のように先着順で

受け入れるか、インポータンスが高いアカウントを優先的に受け入れるかを選べる点と、サーバーの再起動によって登録が勝手に解除されることはないという事でしょう。しかし、ノードが障害を受けて最初から同期し直す場合にはその限りではありませんので、最低限 harvesters.dat はバックアップを取っておくと良いと思います。インポータンス優先サーバーに委任した場合は、時々チェックしないといつの間にか高インポータンスのアカウントに追い出されていることもありそうです。

8.7 ブロックチェーン同期

ひとつながりのブロックであれば、どのような長さに対しても、それに含まれるブロックのスコアを合計することでブロックチェーンスコアが計算できる：

$$score = \sum_{block \in blocks} block\ score \qquad \text{(blockchain score)}$$

ブロックチェーンの同期は、分散型コンセンサス（合意形成）を維持する上で、最も重要な事項である。ローカルノードは、定期的に他のノードにチェーンについて問い合わせをする。問い合わせ先ノードは、パートナーノードの中から、ノードの評判（13 章：評判システムを参照）などの、様々な条件によって選ばれる。

問い合わせ先ノードが、自分より高いスコアを持ったチェーンを提示してきた場合、ローカルノードは自分の持つチェーンのハッシュと比較することによって、直近の共通ブロックを見つけ出す。決定論的なファイナライゼーションが有効化されていた場合には、共通ブロックは（より厳密な）バイナリー検索によっておこなわれる。そして、その場合の探索範囲は、（チェーン全体ではなく）直近のファイナライズされたブロックから現在の自分の保有する最新ブロック高までである。もし、そのようなブロックが見つかった場合には、設定（最大ロールバックブロック数 network:maxRollbackBlocks のこと？）が許す範囲で問い合わせ先ノードからできるだけ多くのブロックを取り寄せる。

もし、取り寄せたブロックチェーンが有効なものであれば、ローカルノードは自分のチェーンを置き換える。もし、取り寄せたチェーンが無効なものであれば、ローカルノードはそれを拒絶して、そのノードが失敗(fail）しているとみなす。

図 19 に、このプロセスの詳細を図解する。

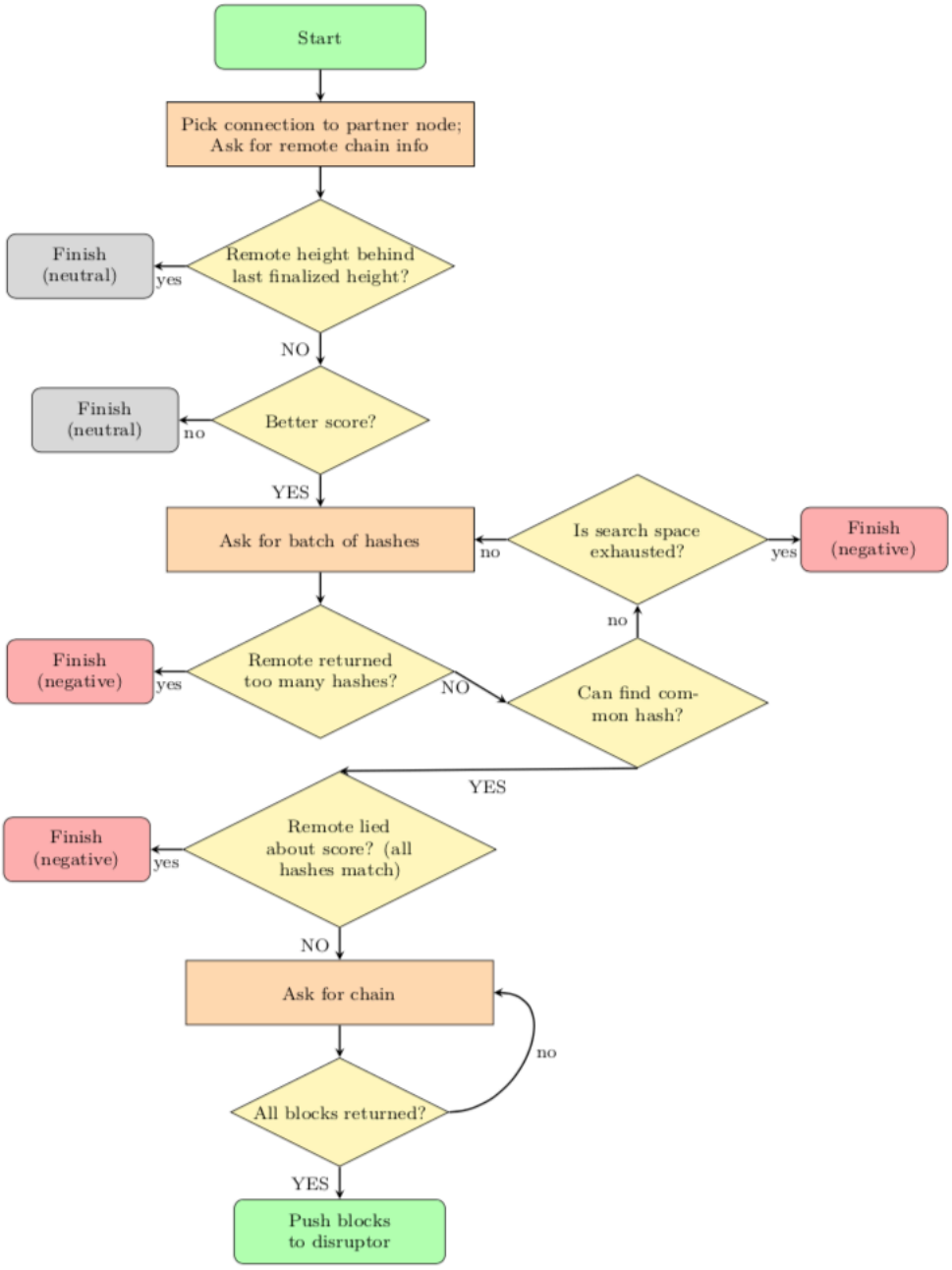


図 19：ブロックチェーン同期のフローチャート

ブロックスコアはブロック難度から、ブロック生成にかかった時間を引いたものです。ブロックチェーンスコアは連続したブロックのスコアを足し合わせるので、ブロック難度が高く、最も長くて、かつ、かかった時間の短い優秀なチェーンを選ぶのに使えます。すべてのノードがそれに合わせようと同期するので、ブロックチェーンの分岐は抑制され、1 本のチェーンだけが伸びていきます。

8.8 ブロックチェーン処理

実行

概論的には、ノードがブロックを受け取った時に、一連の処理をおこなう（脚注；詳しい説明は 9.1：コンシューマーを参照）。処理の前に、ブロックとトランザクションは、一連の通知（notification）に分解される。通知とは、Symbol で使われる基本的な処理単位のことである（2.1 プラグインでも出てきましたが、単なるメッセージ情報というよりは、ブロックやトランザクション処理の基本単位という意味らしいですね）。

ブロックから通知を取り出して、順序よく並べるために、トランザクションは、一旦バラバラに分解される過程で、順番に並べられ、ブロックレベルデータ（block-level data）と結合される。それぞれの分離過程で作られたパーツは、一連のストリーム（stream）状につなげられていく。結果、この通知ストリームは、ブロック内で起きたすべての状態変化とトランザクションを描き出している。

通知ストリームの準備ができたら、通知ごとの処理がはじまる。まず、ブロックチェーン状態（blockchain state）とは切り離された内容の検証である（署名が間違っていないかなどの検証？）。次に、現在のブロックチェーン状態と突き合わせての検証（アカウントの状態変化等に矛盾がないかなど？）がおこなわれる。この検証処理のどこかで矛盾が見つかった場合には、そのブロックは棄却される。合格すれば、通知によって引き起こされた状態変化は、新たなブロックチェーン状態としてメモリー上に書き込まれ、次の通知の処理が始まる。この処理方法によって、同じブロック内のトランザクションが、それ以前のトランザクション状態と矛盾なく連続するようになる。

ブロックに含まれるすべての通知処理が終わったら、レシートハッシュ（7.2.4：レシートハッシュ参照）と状態ハッシュ（7.3：状態ハッシュ参照）が計算される。さらに、network:enableVerificationState がオンになっていれば、状態（ステート）のパトリシアツリーがすべて最新のものにアップデートされる。

ロールバック（巻き戻し）

一度承認されたブロックが、後で破棄されなければならない状況がときどき起きる。フォーク（ブロックチェーンの分岐）を解決するためである。たとえば、現在のブロックを、より良い（ブロックチェーンスコアが高い）ブロックと置き換える時などである。Symbol では、network:maxRollbackBlocks が許すブロック数のロールバックが可能になっている。この数を超えてしまうと、フォークの解決はできなくなる。

ブロックがロールバックされる場合には、一旦通知ストリームに還元される。このストリームは先の実行過程で作られるストリームとは逆向きに進む。トランザクションは同じブロックに含まれる、より古い他のトランザクションに影響を受けることがあるので、（トランザクション間の）依存関係に従って解体するのではなく、まず後から実行されたトランザクションから解体するのである。

通知ストリームができたら、それぞれの通知はふたたび独立に処理される。ロールバックは以前の状態にブロックチェーンを巻き戻すので、検証は必要ない（時間的に古いブロック状態は、さらにそれ以前のブロック状態との関係で検証済みであり、未来の通知から影響を受けることはない）。その代わりに、通知によって起きる状態変化は、現在のメモリ上のブロックチェーン状態から過去に巻き戻されて、次の（より古い）通知の処理へと移っていく。

すべての該当するブロックに含まれていた通知の巻き戻し処理が終わった後、（メモリー上の）ブロックチェーン状態はそれ以前のブロックチェーン状態へと戻る。network:enableVerifiableState がオンになっている場合は、メモリー内の状態ハッシュもアップデートが必要である。その場合も、パトリシアツリーの再構築をおこなわず、ロールバック前の共通ブロックまで、強制的にメモリー内の状態ハッシュを巻き戻す。

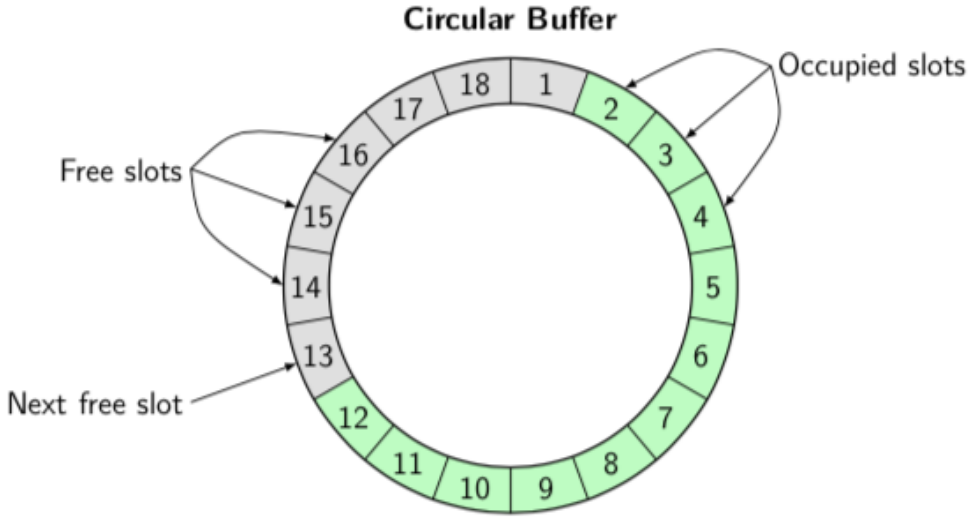
かなり抽象的にですが、メモリー内に新しいブロックやブロックチェーン状態を保存したり、ロールバックの場合はどのようにして処理を巻き戻していくかの手順が書かれています。必要ない処理はおこなわず、処理を軽くするというポリシーがここでも徹底していますね。そしてファイナライズの実装によって、より完璧な形でフォークの制御がおこなわれるようになりました。

9. ディスラプター

「死とは、偉大なディスラプターである。われわれを命の鏡の向こう側へ投げ込み、真実の旅にいざない、再び生まれ変わってすべてをやり直すことができると教えてくれる。」

- B. G. バウワーズ（オーストラリアの詩人）

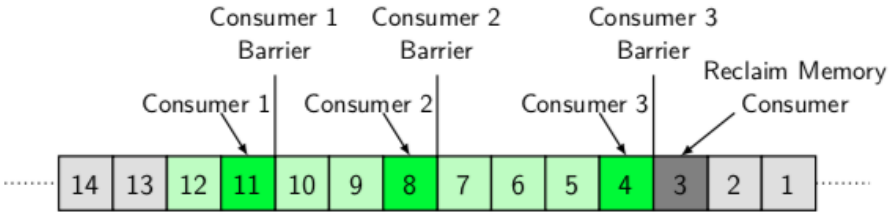
Symbol の重要なゴールのひとつは、高いスループットを得ることである。そのために、ディスラプターを利用して、ほとんどのデータを処理している（脚注；https://en.wikipedia.org/wiki/Disruptor_(software)）。ディスラプターはリング状のデータ構造内にデータを保持して、必要な処理をおこなう。新しいデータは、このリングバッファーの空いているスロットに、順次入れられていく。処理が完了したデータは、新しいデータを入れるために取り出される。このリングバッファーは、限られた数のスロットしか持たないため、処理が情報の流入量に追いつかない場合は、空きスペースを使い切ってしまう。そのような場合、Symbol はサーバーを止めるか、スロットが空くのを待つかを、設定できる。



リングバッファー上のそれぞれのデータ要素は、ひとつまたはそれ以上のコンシューマー（消費者, Consumers）によって処理される。それぞれのコンシューマーは、一つのデータ要素を入力として使用する。コンシューマーのうちいくつかは、入力データを処理して、その結果をデータ要素に追加する。また、他のコンシューマーは、データ要素の検証をおこなったり、データを利用してブロックチェーン全体の状態キーを変化させる。また、あるコンシューマーの処

理は、前のコンシューマーの処理に依存している。そのため、コンシューマーは常にデータ要素の順番に気を配る必要があり、それぞれのコンシューマーは、障壁（barrier）を立てることができる。障壁は、処理が済んでいないデータ要素が、次のコンシューマーによって先に処理されてしまうのを防ぐ。最後のコンシューマーは、使用済みのメモリー領域を開放する。

コンシューマーはまた、データの処理が終了している場合や、何らかの理由でエラーが生じていると判断したときには、CompletionStatus::Aborted という状態関数を使って、強制的に処理を終わらせることもできる。その後に続くコンシューマーは、この終了されたデータ要素を含むスロットを無視するようになる。



ディスラプターとは、マルチコア CPU のキャッシュを効率的に使えるように開発されたデータキャッシュシステムです。ちょうど、世界の経済取引（特に FX 取引）が活発になる一方で、CPU のクロック数の増加が頭打ちになり、複数のコアを使って並列に処理を進めることで処理能力を上げようとしていた時期ですね。このブログ（https://kanatoko.wordpress.com/2014/12/31/why_lmax_disruptor_is_insanely_fast/）を読むと、LMAX という企業が、とことんまで CPU の性能を引き出すためにチューニングしたオープンソースライブラリのようです。Symbol がディスラプターを採用したのは、Dual Core 程度の CPU でも Peer ノードとして十分に機能するようにという方向を狙ったのだと思います。Symbol が比較的低性能のマシンで動くのは、ディスラプター形式を採用したおかげかもしれません。

9.1 コンシューマー

Symbol のブロックディスラプターは、新しく送られてきたブロックと、ブロックチェーンパーツを処理している。ブロックチェーンパーツというのは、いくつかのブロックが連なったデー

タ入力要素である。このディスラプターは、ブロックを検証し、再構成し、ブロックチェーンを伸長させる作業をする。

一方でトランザクションディスラプターは、送られてくる未承認トランザクションを処理する。ディスラプターによって処理が完了したトランザクションは、未承認トランザクションキャッシュに追加される。

すべてのディスラプターは、入力データを処理する一連のコンシューマーによって構成されている。異なるコンシューマーを組み合わせることで、ディスラプターは機能の違いを生み出している。すべてのコンシューマーはリングバッファーのデータを閲覧することができ、一部のコンシューマーはデータを変化させることもできる。

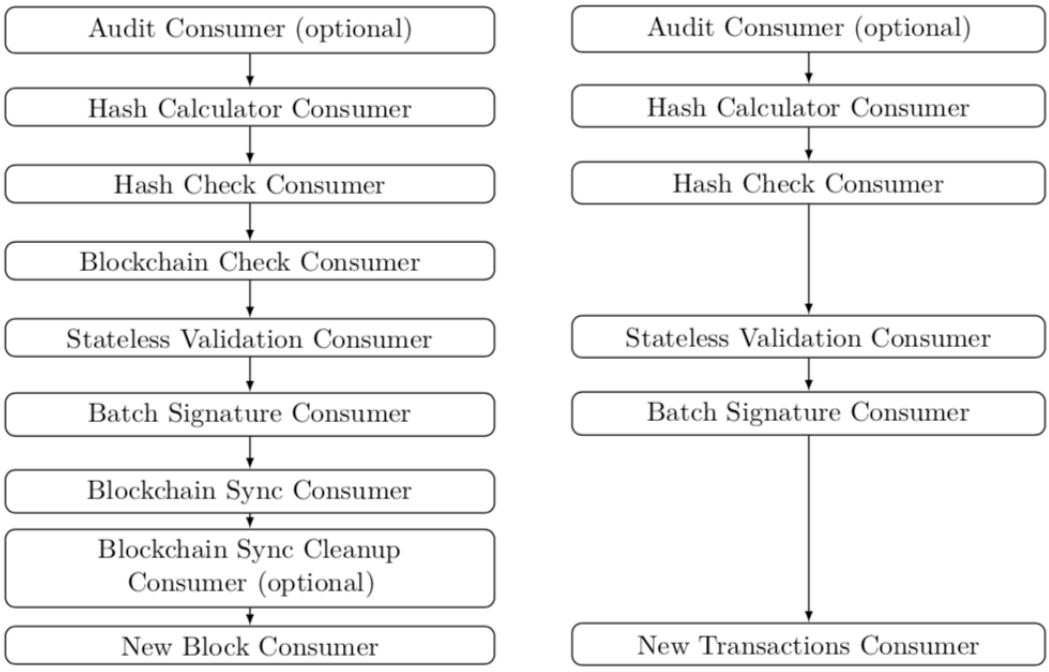


図 20：Symbol のコンシューマーチェーン

9.1.1 共通コンシューマー

ブロックディスラプターとトランザクションディスラプターは、多くのコンシューマーを共有している。

監査コンシューマー

このコンシューマーは、必須ではなく、ノード設定でオン・オフを選択できる。もしオンになっている場合、すべてのデータ要素はディスクに保存される。それによって、ネットワークから入ってくる情報を後で再現することができるため、デバッグを助ける。

ハッシュ計算・検証コンシューマー

ブロックチェーンにおいては、サーバーが何度も同じデータ要素を受け取るのが普通である。なぜなら、ネットワークはたくさんのサーバーから成り立っており、常に他のサーバーにブロードキャストによってデータを送り出しているからである。ネットワークのパフォーマンスを上げるためには、早い段階で送られてきたデータがすでに処理済みのものかを見極める必要がある。そうしないと、同じデータを何度も処理することになる。

ハッシュ計算コンシューマーは、ひとつのデータ要素に関連するすべてのハッシュを計算する。またこのコンシューマーは、受け取ったことのあるすべてのハッシュ値を格納している「ハッシュデータキャッシュ」を検索する。トランザクションディスラプターのコンシューマーも、同様にハッシュキャッシュ（こちらは承認済みのハッシュを格納している）や、未承認トランザクションキャッシュを検索する。これらのキャッシュに、ハッシュ値が含まれていれば、そのデータ要素には CompletionStatus::Aborted がつけられて、その後の処理が省略される。

状態（ステート）非依存検証コンシューマー

このコンシューマーは、ネットワーク状態に依存しない個々のデータ要素内を検証する。この処理は、多くのスレッドを使って並列に処理できる。プラグインによって追加できるのは、この状態非依存検証コンシューマーである。

たとえば、ネットワーク設定の上限を超えない数のトランザクションを含むブロックの検証などは、ネットワーク状態に依存しないと考えられる。そのようなブロックの検証は、ネットワーク設定にもよるが、実際の全体ネットワーク状態には影響されないからである（ノードごとに異なる数や組み合わせのトランザクションをブロックに書き込もうとするためか？）。

バッチ署名コンシューマー

このコンシューマーは、データ要素に含まれるすべての署名を検証する。個々のデータを検証する状態非依存検証コンシューマーと違い、バッチ検証を使用する。処理スピード向上のため、このコンシューマーは複数の署名を一度に検証することができ、同時に多数のスレッドで並列に処理が可能である。

メモリー解放コンシューマー

このコンシューマーは、処理を完了させて、データ要素を格納していたメモリーをすべて解放する。一連の作業によって変更・検証されたトランザクション状態は、より下流の作業に渡されていく。このような同期処理の結果は、主にパートナーノード（13：評判システムを参照）の評判評価に使われる―場合によってはパートナーノードからはずされる― 。

ブロックチェーンでは、多くのノードが同時に同じブロックを処理するため、同じトランザクションデータが繰り返し違うノードから送られてくるという問題があります。そのようなデータもリングバッファーに載せられて、まずハッシュコンシューマーが過去に処理したデータかどうかをハッシュ値で見分けて、その後の処理を行うかどうかのタグを付けていきます。エラーや処理完了のタグがついたデータは、下流のコンシューマーによって無視され、最終的には破棄されます。ディスラプター方式の「処理を止めない」という特徴が、うまく活用されています。

9.1.2 その他のブロックコンシューマー

ブロックディスラプターは、いくつかのブロック処理特有のコンシューマーを使っている。

ブロックチェーンチェックコンシューマー

このコンシューマーは、処理中のデータ要素の中で、状態（ステート）に依存しないブロックチェーン部品の品質チェックをする。すなわち以下の項目をチェックする：

- チェーンに含まれるブロック数が多すぎないか？
- 最後のブロックのタイムスタンプが進みすぎていないか？
- チェーンに含まれるブロック同士は適切に繋がれているか？
- チェーンの中に重複しているトランザクションはないか？

ブロックチェーン同期コンシューマー

このコンシューマーは最も複雑な機能をこなす。ローカルノードがいかなる状態であれ、その状態変化そのものか、あるいは状態を変化させるすべての作業が、このコンシューマーによっておこなわれる。（ファイナライズと深い関係があり、ブロックがつながったチェーン部品（parts）をチェックしてロールバックを可能にするコンシューマーです。）

まず、コンシューマーは処理データに含まれる新しいブロックチェーン部品が、既存のチェーンに追加可能かをチェックする。もしも、最終ブロックよりも過去のブロックに繋がっているならば、接続可能になるまで最終ブロックを取り外していく。

次に、ブロックの状態依存の検証を順次おこない、その過程を記憶する。状態非依存検証は、他のコンシューマーによってすでになされているため、スキップする。処理データに含まれるブロックに、なんらかの齟齬が生じていた場合は、データ全体を却下する。矛盾がなければ、ブロックチェーンの状態（ブロックとキャッシュの両方において）を書き換えて、未承認トランザクションキャッシュをアップデートする。

最後に、このコンシューマーは最新のファイナライズされたブロックを特定して、それ以上のロールバックが起きないようにする。確率論的ファイナライゼーションが採用されている場合、最新のファイナライズされたブロックは、現在のブロック高から network:maxRollbackBlocks の数だけ遡ったブロックになる。決定論的ファイナライゼーションが採用されている場合は、直近のファイナライゼーション検証済みブロックがそれに当たる。コンシューマーは最新のファイナライズされたブロックまでのロールバックを可能にするために必要な、すべてのデータに関係するブロックチェーン状態（ステート）を刈り取る。

このコンシューマーは、Symbol において、チェーン状態を変更しブロックチェーンに対して書き込み許可を持つ唯一のパーツである。

ブロックチェーン同期後処理コンシューマー

このコンシューマーは設定によって省略可能である。有効化されている場合には、ブロックチェーン同期コンシューマーが作ったすべてのファイルを消去する。このコンシューマーはブローカー（クライアントアプリに通知をするためのプログラム）が動いていないノードにおいてのみ有効化すべきである。

新規ブロックコンシューマー

このコンシューマーは、新しく作られたブロックを他のノードに送る機能を持つ。新しいブロックは、自分で作成したものでも良いし、他のノードからプッシュされてきたものでも良い。

9.1.3 その他のトランザクションコンシューマー

トランザクションディスラプターは、トランザクション固有のコンシューマーを 1 つだけ持つ。

新規トランザクションコンシューマー

このコンシューマーは、有効な署名を持ち、状態（ステート）非依存（stateless）検証に合格したトランザクションを、すべてネットワークに送り出す機能を持つ。一方、状態依存性（stateful）の検証は、トランザクションが未承認トランザクションキャッシュに入るまでは実行されない（ネットワークにも送り出されないということ？）。（状態非依存）トランザクションのネットワークへの転送は、状態依存検証前にあえて実行される。その理由は、あるトランザクションについて、受理するサーバーと、受理しないサーバーがあるからである（例えば、トランザクションで提示された手数料によって、サーバーが受理しないということが起こり得る）。その後、状態依存性の検証がおこなわれて、有効性が認められたものだけが、未承認トランザクションキャッシュに追加されるのである。

この章は、重要なブロック生成に関する説明です。状態（ステート）変化が通知として処理され、検証を経て、ブロックにまとめられます。とりあえずは、リングバッファーを使うディスラプターによって高速に処理しているんだなということが分かれば良いような気がします。このディスラプターによって検証作業が完了したトランザクションが、未承認トランザクションとしてキャッシュ（メモリやディスク）に書き込まれます。

10. 未承認トランザクション

「現実逃避のために読書するのではない。人は読書を通じて、体験したことが無い知識が、実在するか確かめるのである。」

\- ロレンス・ダレル （イギリスの小説家、詩人）

どのようなトランザクションも、ブロックに書き込まれるまでは未承認トランザクションである。当然、有効であるものも、無効であるものも存在する。有効な未承認トランザクションは、ハーベストされ、ブロックに書き込まれることになる。トランザクションがブロックに書き込まれ、ブロックチェーンの一部となってようやく、それは承認されたことになる。

未承認トランザクションは、以下のいずれかの方法でノードに到達する：

1. クライアントがノードに直接トランザクションを送った時。
2. 債権付きアグリゲートトランザクションが完成し、すべての共同署名者が署名した後、未完成トランザクションキャッシュから推薦された時。
3. ピアノードが他のノードにトランザクションをブロードキャストした時。
4. ピアノードが、他のノードの未承認トランザクション要求に答えた時。ネットワーク効率化のために、要求するノードは、自分にとって既知のトランザクションのリストを示す必要がある。そうしないと、同じトランザクションがネットワークで何度も送信されることになる。また、トランザクションを要求する側のノードは、自分がブロックを生成する場合の最小手数料率（8.3：ブロック生成参照）を公表する。そうすれば、他のノードは最小手数料に満たないトランザクションをわざわざ送って即棄却される必要が無くなる。

未承認トランザクションをノードが受け取った時、まずトランザクションディスラプターに入る（9: ディスラプター参照）。新規のトランザクションは、状態非依存の検証を受けた後、一旦ピアノードにブロードキャストされる。ブロードキャスト後でも、ノードはトランザクションを拒否することができる。なぜなら、状態依存性の検証は、ブロードキャスト後におこなわれるからである。ノードごとに設定が異なるため、あるノードが受理した未承認トランザクシ

ョンを、他のノードが拒否することがあり得る。たとえば、ノードごとに異なる node:minFeeMultiplier（最小手数料率）がセットされている場合などである。

ウォレットから一台のノードに送られた未承認トランザクションが、どのように複数のピアノードのディスラプターに入っていくかが、少し見えてきました。状態非依存（stateless）と状態依存（stateful）の 2 種類のトランザクションが存在し、非依存の方が先にネットワークにブロードキャストされます。この時、すでに検証したことのあるトランザクションをノードは受け入れずに済むので、ネットワークを効率的に使うことができます。また、ノードはそれぞれ異なる最小手数料率を設定しているため、あまりお金を払えないトランザクションは、受け入れてもらえるノードが少なくなり、ブロックへの書き込みが遅れます。Symbol では、ノード運用者の協力により、低い手数料でもトランザクションが承認されるようになっています（参考；https://nemlog.nem.social/blog/66040）。

10.1 未承認トランザクションキャッシュ

未承認トランザクションはピアノードによる検証を受けた後、ハーベストによってブロック登録ができる状態になる。その際、ノードは検証済みのトランザクションを未承認トランザクションキャッシュに入れようと試みる。この時、2 つの関門がある：

1. node:unconfirmedTransactionsCacheMaxSize によって規定された、キャッシュの最大値を超えないこと。
2. 1 つのブロックに含まれるのに十分な数の未承認トランザクションがキャッシュに存在し、かつ、スパム制限によって拒絶されていないこと。

ブロックチェーンに新しいブロックが付け加えられる時、ブロックチェーン状態（ステート）が変更され、未承認トランザクションキャッシュがそれによって影響を受ける。しかし、キャッシュ内に存在するすべての検証済みトランザクションが、その後も永久に検証済みとして取り扱われるわけではない。たとえば、そのトランザクションはすでに他のノードによってハーベストされてブロックに書き込まれてしまっているかもしれないし、他のトランザクションとの間でコンフリクトが起きているかもしれない。つまり、キャッシュにあるトランザクションは過去には完全に有効であったけれども、ブロックチェーンの状態変化の後に問題が生じることがある。さらに、トランザクションを含んだブロックが巻き戻された場合にも、そこに含まれていた承認

済みトランザクションが新しいチェーンにも含まれるという保証はない。これらの巻き戻されたトランザクションもまた、一旦キャッシュに差し戻される。

これらのことを考慮すると、未承認トランザクションキャッシュは、ブロック状態が変化するごとに完全に再構築されなければならない。そのとき、個々のトランザクションは状態依存の検証を再び受け、その結果が不良であったり、すでに他のチェーンのブロックに書き込まれているなら破棄される。また、そうならなくても一度キャッシュに戻されるのである。

ブロックチェーンは書き込まれたら終わりというわけではなく、フォークした時には巻き戻され（ロールバックされ）たりもします。その場合に、中に入っていたトランザクションも一旦未承認状態に戻り、無効化されたり再び検証を受けて別なチェーンのブロックに取り込まれたりするわけです。このあたりの処理がうまくできるかどうかも、ブロックチェーンの性能と密接に関係しています。

10.2 スパム制限

未承認トランザクションをノードのキャッシュに投入することに対しては手数料を払う必要はない。しかし、キャッシュはノードのリソース（メモリや CPU）を使用するため、ノードは大量の未承認スパムトランザクションが投入されることに対処しなければならない。特に、手数料率をゼロにした無償のノードにとっては、深刻な問題となり得る。

ノードが受け入れることができる未承認トランザクション数に上限を設けるのは、簡単ではあるがあまり好ましいとは言えない。なぜなら、悪意あるユーザーが大量にスパムを送り込んでいる時でも、一般のユーザーはトランザクションを送れるようにしたいからである。また、アカウントごとに送付できる未承認トランザクションの数を制限してしまうのも、解決にはならない。なぜなら、悪意あるユーザーは自由にいくつでもアカウントを作って、大量のトランザクションを生じさせることができるからである。

Symbol は、キャッシュが攻撃者のトランザクションでいっぱいになってしまうのを防ぐために、有効な方法でスパム制限を設けている。そして、その場合でも誠実なユーザーは新しい未承認トランザクションをキャッシュに送り込むことができる。
node:enableTransactionSpamThrottling を設定することで、このようなスパム制限を利用できるよ

うになる。例として、キャッシュがまだいっぱいになっていない場合には、以下のように処理される：

1. 未承認トランザクションの数が、ひとつのブロックに内包できるトランザクション数より少ない場合はスパム制限を適用しない。
2. 投入されたトランザクションが、債権付きアグリゲートトランザクションである場合には、スパム制限を適用しない。
3. それ以外の場合には、スパム制限を適用する。

curSize をキャッシュに存在するトランザクションの数、maxSize をキャッシュの最大サイズとする。また、rel. importance of A をアカウント A の相対的インポータンスとし、0 から 1 までの数値を取るとする。もし新しい未承認トランザクション T が A の署名とともに投入されたとき、A の fair share は以下のように計算される：

$$
\begin{aligned}
maxBoostFee &= \text{transactionSpamThrottlingMaxBoostFee} \\
maxFee &= \min(maxBoostFee, T::\text{MaxFee}) \\
eff.\ importance &= (rel.\ importance\ of\ A) + 0.01 \cdot \frac{maxFee}{maxBoostFee} \\
fair\ share &= 100 \cdot (eff.\ importance) \cdot (maxSize - curSize) \cdot \exp\left(-3\frac{curSize}{maxSize}\right)
\end{aligned}
$$

式の説明の補足：

maxBoostFee は、トランザクション手数料によってインポータンスにブーストをかける場合の最大値を規定する。

maxFee は、トランザクション内に記載された MaxFee と maxBoostFee の小さい方が選ばれるようになっている。トランザクションに設定した手数料が少ないと、インポータンスブーストも小さくなり、投入できるトランザクション数は減る。

eff. importance（有効インポータンス）は、トランザクションを発生させたアカウントのインポータンスに、最大 0.01 のブーストを加えたものである。

fair share は、ひとつのアカウントが投入可能な最大トランザクション数。

もしアカウント A がキャッシュに fair share 数いっぱいのトランザクションを投入している場合は、新規のトランザクションは投入できない。fair share 内であれば許可される。トランザクション内に設定された最大手数料（MaxFee）が大きければ大きいほど、キャッシュに投入できる

トランザクションは多くなる。それでもやはり、有効インポータンスのブーストには限界があり、node:transactionSpamThrottlingMaxBoostFee がその最大値を決めている。

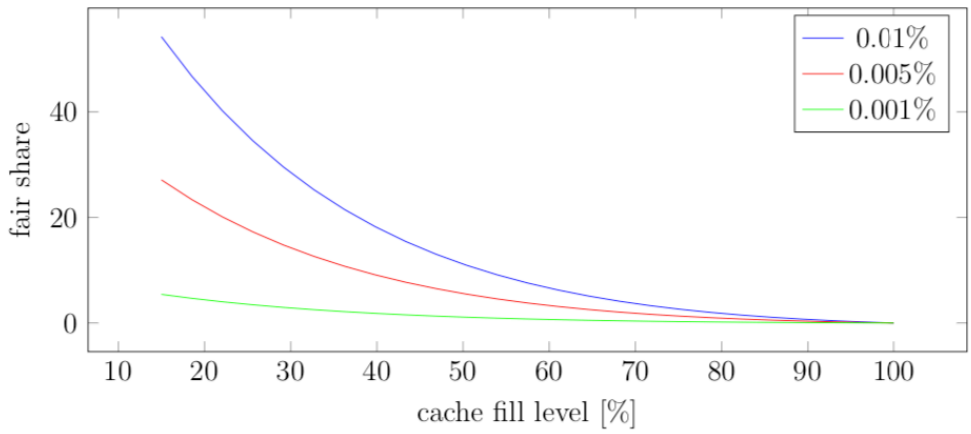


図 21：様々な有効インポータンス（0.001 – 0.01）下での、fair share（アカウントが投入できるトランザクション）の数　最大キャッシュサイズ 10000 の場合

図 21 に、さまざまな有効インポータンス下での、未承認トランザクションキャッシュの充足率と fair share の関係を示す。攻撃者が、多くのアカウントを作ってキャッシュを専有しようとしても、各アカウントのインポータンスが（分割されて）低くなってしまい、効果は薄い。また、トランザクション手数料を多く払って fair share を上げようとしても、コストが大きくなり、手持ちの資産がより早く尽きるだけである。

NIS1 では、1 ブロックあたりが保持できるトランザクション数が 120 しかありませんでしたが、それでもスパム制限の仕組みが働いて、混雑時でもトランザクションの投入が長く待たされる事は無かったように思います（遅くとも数分以内に処理された）。Symbol においては、手数料とインポータンスの関係がより分かりやすくなり、それによってスパム制限も手数料を軸に再設計されたようです。この説明だけだと、ノードごとに変更できる MaxBoostFee の解釈が難しいですが、ノードは高い MaxBoostFee を設定して高い手数料設定が意味を持つようにしたり、逆に低く設定してすぐに飽和してトランザクションごとの手数料設定による差が出にくくもできますので、サーバーの処理性能と相談の上で、どのようなトランザクションを受け入れるかを調整できる仕組みと考えると良いのかもしれません。

11. 未完成トランザクション

「全体は、個の合計よりも大なり」

- アリストテレス

債権付きアグリゲートトランザクション（6.2：アグリゲートトランザクションを参照）は、未完成（partial）トランザクションとも呼ばれる（partial の訳は、部分、分割、未完了などいろいろ訳は考えられますが、とりあえず未完成でいきます）。未完成という語句は、トランザクションが不十分な共同署名しか持たず、追加の署名を集めないと検証を通ることができないところから来ている。

未完成トランザクションを取り扱うための機能拡張として、partial transaction extension が用意されている。ネットワークが債権付きアグリゲートトランザクションをサポートする場合には、この機能拡張をすべての API と Dual ノードで有効にしなければならない。

未完成トランザクションは、この機能拡張が有効化されたネットワークのすべてのノードで同期される。ノードは、他のノードから未完成トランザクションと未完成な共同署名を受け取る。そして、pull partial transactions task によって、他のノードに繰り返しトランザクションと署名を要求する。ネットワーク最適化のために、これらを要求するノードは自分がすでにどのトランザクションと共同署名を保有しているかを提示して、データが重複するのを避ける。

未完成トランザクションがネットワークで利用できるようにするために、ハッシュロックプラグインが有効化される。そして、トランザクションに紐付けられたハッシュロックが生成される。ハッシュロックは、基本的には共同署名を集めるサービスを利用するための債権（bond; アカウントから見ると仮払金＝トランザクション投入時に 10XYM 質に取られるやつです）である。ハッシュロックに紐付いた未完成トランザクションが完成して、ハッシュロックの期限が切れる前にブロックチェーンによって承認されたならば、債権は支払いをしたアカウントに返却される。承認されることなく期限が切れた場合の仮払金は、ハッシュロックが期限切れになった時のブロックをハーベストしたアカウントが回収する。このしくみは、未完成トランザクションを大量発生させるスパム行為を防ぐことになる。なぜなら、キャッシュに未完成トラ

ンザクションを投入するためには、node:lockedFundsPerAggregate で定められた保証金を、一時的にせよ支払わなくてはならないからである。

ノードが新しい未完成トランザクションを受け取ると、partial transaction dispatcher（ここで初めて dispatcher; ディスパッチャーという言葉が出てきますが、ディスラプターで利用されれるコンシューマーをいくつか連ねたもののようです）。受け取った共同署名を集めて、キャッシュに入っている未完成トランザクションと照らし合わせて、もしも該当するトランザクションがなければ即時棄却する（脚注；つまり、共同署名を集める前に、キャッシュの中に該当するトランザクションが存在しなければならないということ）。トランザクションと共同署名が揃った時に、それらは個々のトランザクションに分解されて検証処理される。

一般のトランザクションのディスパッチャーと比較して（9.1 コンシューマー参照）、未完成トランザクションのディスパッチャーは機能省略型である。

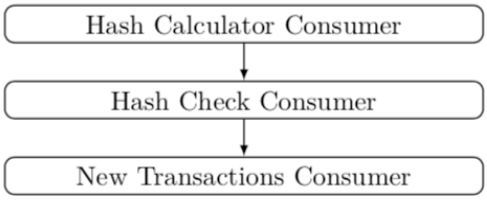


図 22：未完成トランザクションコンシューマー

ハッシュ計算コンシューマーと、ハッシュ検証コンシューマーは 9.1.1：共通コンシューマーに出てきたトランザクションディスパッチャーと同じ働きをする。違いとしては、この場合のハッシュ検証コンシューマーは、処理済みトランザクションキャッシュではなく、未完成トランザクションキャッシュを検索する。新規(new)トランザクションコンシューマーは、9.1.3：追加のトランザクションコンシューマーに出てきたものとほぼ同じ機能を持つ。違いは、未承認トランザクションではなく、未完成トランザクションをブロードキャストして処理する点である。言い換えると、新規トランザクションコンシューマーは未完成トランザクションをネットワークにブロードキャストしてから、有効なものを未完成トランザクションキャッシュに追加する。

スマートコントラクト様の機能をビルトインで持つ Symbol は、アグリゲートトランザクション

を自動的に処理してくれる高機能ブロックチェーンです。複数のトランザクションと署名の組み合わせを結びつけて、順番に処理するために、期限付き債権が生じたりして一見複雑に見えますが、基本は普通のトランザクションと変わりません。トランザクション受け入れ＞状態非依存検証＞未完成トランザクションキャッシュに格納＞ブロードキャストと署名集め＞トランザクションディスパッチャーで検証＞未承認キャッシュに格納＞ブロックチェーンコンシューマーのリングバッファーに入れて検証＞ブロックチェーンに書き込みという感じの流れだと思います。

11.1 未完成トランザクションの処理

未完成トランザクションのキャッシュには、追加の共同署名を待っている未完成のトランザクションが置かれている。新しい未完成トランザクションは、すべての内容検証を経た後にキャッシュに追加される。未完成トランザクションは、十分な数の共同署名を集めるか、または期限切れなどで無効になるまで、キャッシュにとどまる。たとえば、（未完成トランザクションの保存期間を定めている）ハッシュロックが期限切れになれば、その未完成トランザクションはキャッシュから消去される。一方、十分な数の共同署名が集まれば、ただちに、未完成トランザクションはトランザクションディスパッチャーに渡されて、未承認トランザクションとしての処理が開始される。

未完成トランザクションキャッシュは、新しい共同署名を受け取ると、収集済みのすべての署名と照合をおこなう。キャッシュ（の中にある未完成トランザクション）に（新しい署名が）追加されるためには、その共同署名は新しいものでなければならない。また、その時点での未完成トランザクションのどれかと関連づけられて、検証ができるものでなければならない。すでに追加済みの共同署名が、後で無効になる場合もある。その場合には、無効になった共同署名は（トランザクションから）切り離される。たとえば、未完成トランザクションに署名をしたマルチシグアカウントから、該当するアカウントが除名された場合などには、共同署名そのものが無効になる。共同署名の照合処理は、このような込み入った事象にも対応できるよう設計されている。

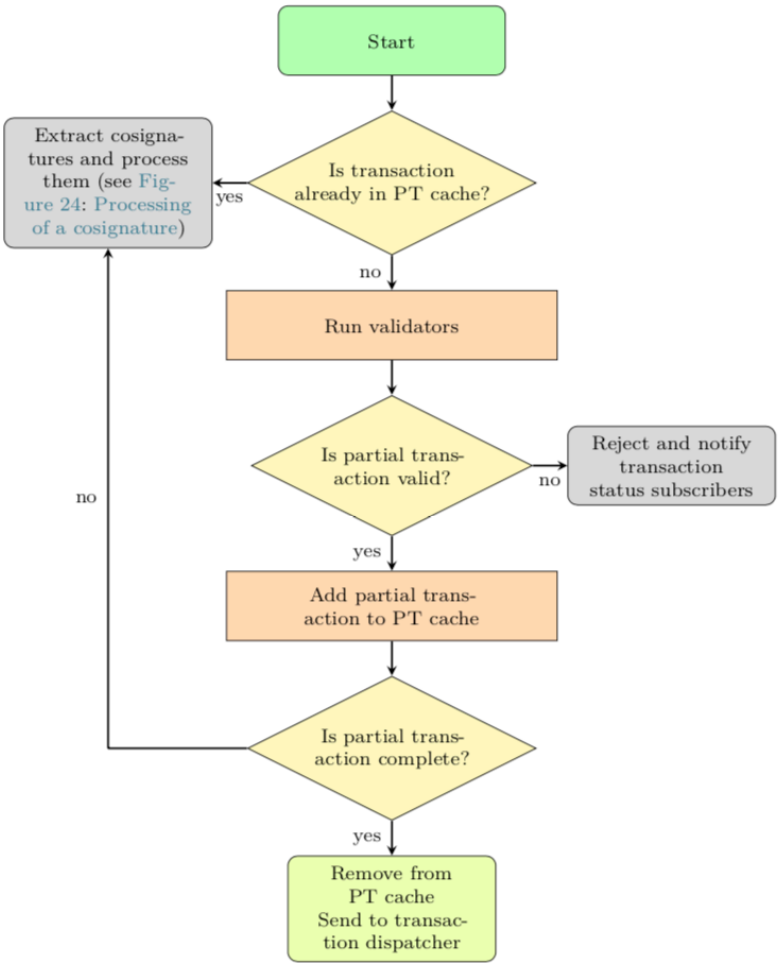


図 23：未完成トランザクションの処理

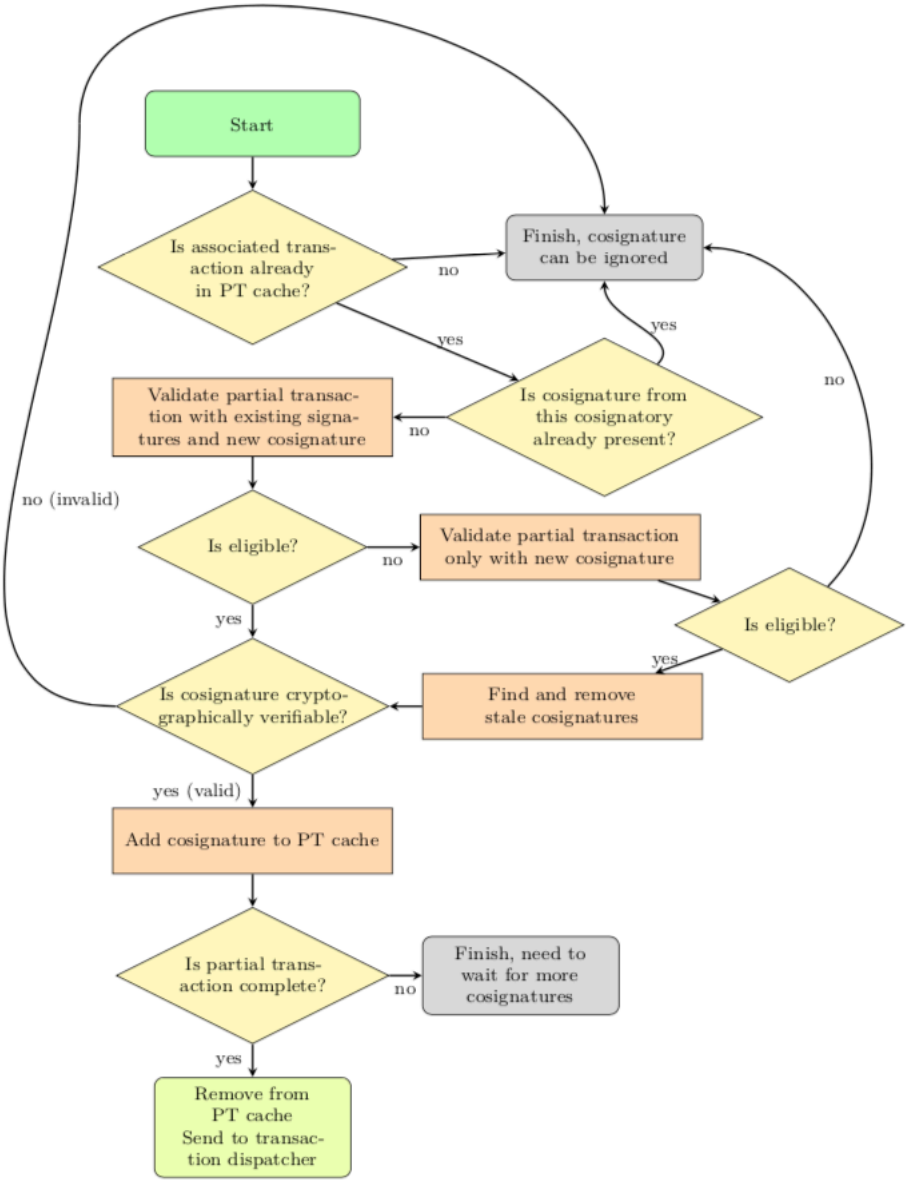


図 24：共同署名の処理

未完成トランザクションキャッシュの存在理由の大部分は、アグリゲートトランザクションを適切に処理するために、十分な数の署名が集まるまで「待つ」ことにあると思います。NIS1 では、マルチシグアカウントによってトランザクションが発生した場合には、共同署名者に NEM Wallet を通じて通知が行くようになっていました。共同署名者が確認するまで、そのトランザクションはブロックチェーンに刻まれずに待っていたわけです。Symbol でも基本は同じですが、アグリゲートトランザクションはより高機能になっています。署名が集まっていない未完成ト

ランザクションは、キャッシュに保存され、追加の署名を待つことになります。このようなトランザクションが無期限にキャッシュを専有することがないように、保存期限が決められていて、それをすぎるとトランザクションそのものがキャッシュから消去されるようです。その場合、仮払金は没収され、ハーベストしたアカウントに渡ることになります。

12 ネットワーク

「優秀なネットワークを引き寄せるためには、努力と誠実さ、そして時間が必要である」

- アラン・コリンズ

ノードを動的に見つけながら、ピアツーピアネットワークは成長する。Symbol は、この動的な作業を node discovery extension (ノード探索機能拡張）によって実現している。パブリックネットワークは、一般に公開されており、どのようなノードでも参加できる。プライベートネットワークは参加ノードを限定でき、より組織的な振る舞いをする（脚注；注意深くハーベストモザイクと通貨モザイクを配分することで、プライベートネットワークでもそれぞれのアカウントに異なった役割を許可できる。たとえば、一定量のハーベストモザイクを保有するアカウントだけがブロックを生成でき、通貨モザイク（手数料モザイク）を保有しているアカウントだけが、手数料付きのトランザクションを発行できるなどである）。Symbol の柔軟性によって、プライベートネットワークでも、設定ファイルによって、すべてのノードの役割分担をうまく制御することができる。

Symbol は、network:nodeEqualityStrategy（ノード均質化ストラテジー）によって、参加ノードを見つけたり、規格外ノードを排除できる仕様になっている。ノードの標識としては、解決済み IP アドレス（host, 脚注；ノードの解決済み IP アドレスは、host 名が提供されていない場合にのみブロードキャストされる。ダイナミック IP を使用しているノードをサポートするために、host 名が優先される仕様になっている）、または、ブート公開鍵（public-key）が利用できる。パブリックネットワークでは前者を利用することを薦める。

12.1 ビーコンノード

立ち上がったばかりのノードは、他のノードから分離されていて、どのピアノードとも接続されていない状態で始まる。まずネットワークに参加しなければ、ブロックの検証やハーベストなど、意味のある貢献ができない。Symbol においては、まず固定（static）ビーコンノードのリストがピア設定ファイルに書き込まれており、新規のノードがネットワークに接続する場合には、これらの固定ビーコンノードにまず接続を試みる。この固定ビーコンノードのリストは、ネットワーク内のすべてのノードで同じである必要はない。

パブリックネットワークは、高い接続能力を持ったビーコンノードを候補として設定することが推奨される。それぞれのノードのピア設定ファイルには、このビーコンノード候補全体から、ランダムに選ばれた部分リストが含まれるようにすると良い。ビーコンノード候補の選択は、ネットワーク接続に先立って一度だけ行われる。また、ノードの再起動によってもおこなわれる。ビーコンノードは地理的にできるだけ分散されていた方が良い。それによって、個々のビーコンノードに負荷が集中することがなくなり、DoS 攻撃なども受けにくくなるからである。これらのノードは、他の（非ビーコン）ノードに比べて、わずかに選択されやすくなっている（13.2：重み付けされたノード選択を参照）。ビーコンノードは、ほぼ確実に接続できることが期待されるからである。それ以外は、ビーコンノードに特別な権利や責任が存在するわけではない。いわば、ネットワークに通じるドアみたいなものである。

一部の機能拡張は、それぞれに特化したビーコンノードを必要とするかもしれない。たとえば、未完成トランザクション機能拡張（partial transaction extension）は、ピア設定ファイルに独自のビーコンノードリストを持っている。この機能拡張が有効化されているノードは、（一般のトランザクションとは）別に、同じ機能拡張が動いているノード同士だけで同期をおこなう必要がある（11: 未完成トランザクションを参照）。

ノードの役割（role）は、そのままノードがサポートする機能を示している。すなわち、ノードが適切なパートナーノードを見つける際に使われる。Peer の役割を与えられたノードは、基本的なブロックチェーンの同期をサポートする。API の役割を持つノードは、未完成トランザクションの同期をサポートする。投票（Voting）の機能を持つノードは、決定論的ファイナライゼーションが採用されている場合に、ファイナライズのための投票に参加する。IPv4 をサポートするよう設定されたノードは、IPv4 の通信を使う。同様に IPv6 をサポートする設定のノードは IPv6 を使うことになる（脚注；ノードが明示的に IPv4 か IPv6、あるいはその両方を指定していない場合には、IPv4 のみをサポートしているものとする）。これらの役割は互いに排他的ではなく、複数の役割を持つノードも存在する。

設定ファイルの中には、ほぼ確実に接続できるビーコンノードのリストが書き込まれています。NIS1 の Alice サーバーなどに相当するでしょう。NGL ノードが廃止された時に、このリストも一新されました。この白書では、これらのビーコンノードには選択されやすさがあるけれども、それ以外には特にメリットはないとも書かれていますので、ビーコンノードを大幅に入れ替えても問題ないようです。

12.2 ハンドシェイク

Symbol のノード間接続は、TLS1.3 準拠のカスタム化された手順でおこなわれる。それぞれのノードは、2 階層の X509 チェーン（電子署名の一種で、公開鍵と署名を使ったチェーン構造を持つ。詳しい解説についてはこちらが参考になります;
https://qiita.com/TakahikoKawasaki/items/4c35ac38c52978805c69）
を持ち、それらはルート証明書とノード証明書を発行する。すべての証明書は X25519 形式の証明書でなければならない。Symbol はその他のタイプの証明書をサポートしない。

ルート証明書は、アカウントの署名用秘密鍵で自己署名される。このアカウントは、ノードの所有者のものであると考えて良い。暗号論的には、ノードとアカウントを結びつけることで、ノードの所有者が持つインポータンスを使った、ノードの重み付けが可能になる。さらに、パートナーノードは、このアカウントを使って評判スコアを収集することもできる（13: 評判システムを参照, 脚注 ; パブリックチェーンでは、ノードは基本的には IP アドレスで区別される）。重要なのは、この証明書はノードの自己証明にのみ使用されるということである。そのため証明書への署名が終わったら、セキュリティー上、アカウントの秘密鍵を動作中のサーバー上に置くべきではない（ハッキングされれば当然アカウントが持つ資産もインポータンスも一緒に流出します）。

ノード証明書は、ルート証明書を使って（別の秘密・公開鍵ペアによって）署名される。署名用の鍵は、ランダムな秘密・公開鍵ペアでよい。この証明書は、TLS セッションの確立のために使われ、暗号化のための鍵をノード間で共有できるようにする（脚注 ; 現在のところ、委任ハーベスター検出（8.6：ハーベスターの自動検出　参照）の時に利用されるメッセージの暗号化および復号化用の暗号化キーを共有するためにだけ使われている）。必要に応じて鍵をローテートすることができる。

この接続確立プロトコールは、パートナーノードごとに独立しておこなわれる。もし、ハンドシェークに失敗した場合は、そのノードとの接続はただちに切断される。

12.3 パケット

Symbol は、TCP を使ってネットワーク通信をおこなっている。使用される TCP ポートは node:port で指定される。主要な通信は、TCP パケット上に構築された、より高度で特殊なパケットモデルによっておこなわれる。すべてのパケットは、パケットのサイズとタイプが記述された 8 バイトのヘッダーから始まり、パケット全体の受け取りが終わった後に、情報処理に回される。

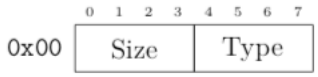


図 25: パケットヘッダーのバイナリーレイアウト

接続には、長寿命型と短寿命型がある。長寿命型の接続は、ブロックやトランザクションの同期など、繰り返しておこなわれる処理に使われる。プッシュ形式と、リクエスト/レスポンス形式の両方がサポートされる。接続は、node:maxConnectionAge が指定する回数試行されるまで維持され（13.1 : 接続マネジメントを参照）、その後別用途にリサイクルされる。この設定よりも古くなった接続がリサイクルされるのは、他のパートナーノードとの接続に再利用するためであり、または、ゾンビ化する接続への対策が目的である。

短寿命型の接続は、ノード間のもっと複雑な多段階接続に使われる。たとえば、ノードの探索（12.6 : ノード探索を参照）や、時刻同期（16 : 時刻同期を参照）などである。短寿命型接続は、長寿命型接続回線がすべて使用されてしまったときに、新たな同期パートナーノードを見つけられなくなるのを防いでいる。

パケットを処理するのには、ハンドラが使われる。それぞれのハンドラは、特定のタイプを持ったすべてのパケットに紐付けられている。パケットが完成して処理に回される時、そのパケットタイプに紐付いたハンドラに受け渡される。すべてのハンドラは、自分と紐付けられたパケットを必ず受け取らなければならない。一部のハンドラは、リクエスト/レスポンス型プロトコールに対して、レスポンスのパケットを送り返すこともできる。

Symbol は、パケットレベルでもかなりカスタマイズされているようで、非専門家の僕はなかなか理解が追いついていかないです。もっとも重要なノード所有者のアカウントを証明するのには、TLS1.3 方式が使われて、安全な接続で情報が暗号化された上でやりとりされるようです。頻繁なやり取りが必要なノード探索などには、短寿命のパケットが使われ、ブロックやトランザクションの同期には、長寿命の物が使われるみたいです。

12.4 接続型

Symbol では、長寿命の接続は、まず読み手か書き手に分類される。すなわち、データが入力されるか、データを出力するかである。次に、それぞれの接続は目的によって区別される。その目的は service identifier によって識別される（脚注；言葉としては似ているが、「2.2：Symbol 機能拡張」に出てきたサービスとは無関係である）。

読取接続は、ほとんどの場合受け身で、他のノードからデータを受け取るために利用される。個々のサーバーは、読取接続を通じて、非同期的に接続先サーバーからデータを入手する。新しいパケットの受け取りが完了すると、サーバーはそのパケットデータを該当するハンドラに渡す。適合するハンドラが存在しない場合は、接続はただちに切断される。

node:maxIncomingConnectionsPerIdentity は、すべてのサービスに渡って、長寿命型と短寿命型の接続数の上限を決めている。この上限を超えた接続リクエストは、即刻拒否される。この上限に達するのは、複数の短寿命型接続リクエストが、同じリモートノードから複数の作業について送られてきた場合に起こりうる。特に、ノードが起動した直後のサーバーからは、このような積極的な接続要求が送られてくる場合がある。接続制限エラーは一般に長くは続かず、時間とともに消失するのならば無視しても問題ない。

書込接続は、より能動的で、他のノードにデータを送付するのに使われる。ブロードキャスト操作は、すべての生きていてかつ受け入れが可能な書込接続に送りつけられる。その後、それぞれの書込接続は、個々に選別された後、リクエスト/レスポンスプロトコールへと送られる。受け手側の処理を簡略化するために、現在リクエスト/レスポンスプロトコールが継続中である接続相手には、ブロードキャストパケットは送られない。

サービス ID は、長寿命型の接続にだけ振り分けられる。この中で、同期サービスは、ピアノードに向けてデータを送りたい場合に使われる。同様に、API パーシャル（未完成トランザクション用？）サービスは、API ノードにデータを送り出すために使われる。読取サービスは、情報を受け取るための接続に使用される。API 書込サービスは、まだ実験段階であり、node:apiPort によって規定されているポート番号に、書込みを許可させるために使われる。

Identifier	Name	Direction
0x50415254	pt.writers	outgoing
0x52454144	readers	incoming
0x53594E43	sync	outgoing

図 26：サービス ID

「13.2：重み付けされたノード選択」で述べられるような、接続ノードの選択には、ノードの稼働時間とパートナーとして選択された頻度が、サービスごとにスコア化される。この評判情報は、すべてのサービスについて集計される。たとえば、あるノードが同期接続と API パーシャル接続を、（同時に）他のノードとしたとする。この場合でも、それぞれの接続は、異なる稼働時間を持つことになるだろう。なぜなら、スコアはサービスごとに集計されるからである。ちなみに、通信結果（Interaction result，通信の成功または失敗は、後で出てくる評判スコアを計算するのに使われる）は、どの接続形態についても、ノードごとに集計され、サービスには帰属しない。

後出の、サーバー選択基準に使われる評判システムに関する記述に、一部不明なところがありますが、読み進めていくにつれ明らかになっていくでしょう。

12.5 ピア来歴

ノードは、ブロックチェーンネットワークに参加しているすべてのノードに関する情報を集める。その評価は、ノードごとの来歴によって決まる。評価は高い方から低い方へ下記の順番で並ぶ：

1. ローカル（Local）ノード - ノードは node:localnode に記載されている。
2. 固定（Static）ノード - ノードはピア設定ファイルに記載されている（ビーコンノード？）。
3. 動的（Dynamic）ノード - ノードは、通信によって発見され、接続が可能な状態にある。
4. 受動的（Dynamic-Incoming）ノード - ノードは、接続してきたことがあるけれども、接続は確立していない。

まず固定ノードとなるためには、少なくともひとつのピア設定ファイルに記述されていなければならない。つまり、あるノードにとっての固定ノードは、別なノードにとっては動的ノードになっていることもある。ローカルノードを除いて、残りのノードは動的ノードとみなされる。動的ノードのうち、一部は受動的ノードとされる。受動的ノードは、そのノードからデータを受け取ったことだけがあり、こちらから送信したことがないノードのことである。この場合、接続可能なポート番号が未知なため、通信可能な状態にはなっていない。

ノードデータは、現在より高い評価となるような場合にのみ、書き換えられる。たとえば、固定ノードの来歴評価は、動的ノードとしての来歴評価によっては書き換えられないが、動的ノードの来歴評価は、動的または固定ノードとしての来歴評価によって書き換えることが可能である。

上記のようなことは、実際の接続が確立するプロトコールを考えると少々簡略化してある。ノードがリモートノードから接続を切断し、その後再接続する場合には、2 つの段階を経なければならない。動的ノードがリモートノードに再接続を試みた場合、別なノード公開鍵が必要である。その時、リモートノードはその動的ノードを（新規に接続してきた）受動的ノードとみなすため、評価がひとつ落ちる。その結果、リモートノードは動的ノードの情報をアップデートしようとしない。その代り、アップデートが進行中であるというフラグを付けるにとどまる。その後、リモートノード側から直接、動的ノードに接続を試みた時にようやくノード情報がアップデートされるのである。この時には、リモートノードはすでに確立した接続（受動的ノード接続）が存在するにもかかわらず、ノード情報をアップデートする。なぜなら、それ以前の段階でアップデート進行中であるとみなしているからである。このフラグが無いと、格下ノードとの通信中はノード情報のアップデートをしないため、再接続が起きる条件では不都合がある。

network:nodeEqualityStrategy に書かれたノード ID 決定方法が公開鍵（public-key）である場合、2 次的なノード ID は IP アドレスになる。アクティブな接続が存在しない時には、この ID は変更可能である。しかしこの方法では、ノードの評判情報は他の ID へは移管できない。

network:nodeEqualityStrategy がホスト（host）である場合、2 次的ノード ID はノード公開鍵になる。アクティブな接続が存在しない時、この ID は変更可能である。しかし、IP アドレスが変更された場合にも、このノード ID（＝公開鍵）は変化しないとみなすことができる。すなわち、ノード公開鍵と紐付いているすべての評判情報は、IP アドレスが変わっても引き継がれる。評判情報の一部が、（IP アドレスと結びついていて）ノード公開鍵と結びついていない場合は、その部分は破棄され、残りが引き継がれる。

TLS や X509 の導入により、ノード間通信のセキュリティーが上がっています。接続ごとに秘密・公開鍵が変化するため、一度接続が切れると、まったく別なノードのように見えることになります。IP アドレスも変化することを前提としており、ノードと紐付いたアカウント情報（公開鍵）を使って、接続相手を格付けるようなことをしているようです。

12.6 ノード探索

ノードは立ち上がってすぐに、まずはピア設定にリストされたすべての固定ノードに、短寿命の接続を試みる。この初期接続は、すべての固定ノードの IP アドレスを取得するのが目的である。ピア設定ファイルには hostname を記述すればよく、ノード管理を容易にしている。ノードが動いている限り、線形バックオフ（linear backoff）によって IP アドレス解析が継続される。

ノードは定期的に ID 情報をブロードキャストし、他のパートナーノードに自分の情報を繰り返し伝える。情報を受け取ったパートナーノードは、その情報を解析し、有効かどうかや互換性を検証する。情報が有効であるというのは、ノードによって送られたノード ID 公開鍵が、ルート X509 証明書に記載された公開鍵と一致しているということである。互換性とは、ブロードキャストをおこなったノードと、それを受け取ったノードが、同じブロックチェーンネットワークにあるかどうかである。もしも、hostname が提供されない場合は、ノードの IP アドレスが代わりに用いられる。これらの検証に合格すれば、ノードは新しいパートナーノード候補になり、次の同期作業に参加できるようになる。

また逆に、ノードは定期的に既知のすべてのピアノードに対して、パートナーノードリストを要求する。他のピアノードが使用中の固定あるいは動的なピアノードは、ノードのリクエストに対して返答することが期待できる。つまり、要求を送ったノードは、これらのノードすべてを、動的ノードとして取り扱うことができる。その後、ノードはこれらの動的ノードリスト全てに対して、ID 情報をリクエストする。この直接のコミュニケーションによって、悪意のあるノードが他のノードに関するニセの情報を広げるのを防ぎ、新しいノードと確実に接続できるようになる。ノードは、受け取った情報を処理して、有効性と互換性をチェックする。すべてのチェックに合格すれば、新しいノードはパートナー候補となり、次回の同期作業に使用されるようになる。

パートナーノードを見つけるために、ビーコンノード以外は他のノードが接続しているノードに直接アクセスしながら候補を探していくシステムです。これによって、信頼できるネットワ

ークが構築されていきます。京都風に言うと、一見さんお断りシステムですね。TLS と X509 を使って、接続プロトコールには堅牢な証明書発行と暗号化システムが組み込まれたため、IP 偽装などによって通信途中でノードをすり替えようと思っても難しいでしょう。

13 評判システム

「評判とは多くの功績を必要とし、たったひとつの悪評で失うものである」

- ベンジャミン・フランクリン

Symbol は、ピアツーピア（P2P)ネットワークを使っている。P2P ネットワークは頑強なシステムである。どれか 1 つのノードが失われたとしても、停まることはない。それでも、パブリックネットワークは、P2P システムならではの問題を抱えることになる。ネットワークには、誰もが匿名で参加することができる。つまり、反友好的なノードを送り込んで、ネットワークに虚偽の情報を拡散したり、稼働を妨害することができる。

どのようにして、そのような反友好的ノードを識別し、排除するかというのが問題になる。このために、多くのアプローチが考え出された。その中でもうまくいっているものの 1 つが、ノードの評判システムである。Symbol は、単純な評判システムを導入している。このシステムは、うまく運用されているノードへの接続に高いプライオリティーを与え、悪意あるノードの評判を下げることを目的としている。重要なのは、評判スコアそのものは、ブロックチェーンのコンセンサスには影響を与えず、ネットワークの接続構造を変化させるということだ。この章は、そのストラテジーについて概説する。

13.1 接続管理

個々のノードは node:maxConnections で指定された数まで接続を維持できる。この制限は、ネットワーク全体を構成する数百、数千のノードの数に比べると非常に少なく設定されている。少数のノードが閉じたグループを形成してしまうのを防ぐために、ノードは常に古い接続を切断し、別なノードとの新しい接続をつなごうとする。

どの接続を切断するか決定する時、ノードは接続の古さ（age）を利用する。接続の無駄を極力省くため、node:maxConnectionAge で設定された回数、接続を繰り返したノードから切断候補となる。そして、node:maxConnectionAge 回の接続が繰り返された後、接続は切断され、別なノードとの接続が開始される。このようにして、個々のノードがネットワーク内の多くの異なったノードと接続・切断を繰り返す。

前のコメントで、初回の接続は「一見さんお断り」の紹介システムであると書きました。しかし、それだけだとお互いに接続しやすいノードだけで固まって、狭くて閉じたネットワークになってしまいます。そこで、ある回数接続を続けた常連さんに出ていってもらって、新しいお客を入れる仕組みが組み込まれています。その時、どのような候補を選ぶかが次回からのテーマになります。このシステムはコンセンサスアルゴリズムには一切影響せず、ネットワークの堅牢性、正当性を高めます。

13.2 重み付きノード選択

ノードは基本的には、現在接続中のノードと、お互いにデータのやりとりをする。ノードは他のノードに、新しいトランザクションやブロックの要求を出す。あるいは、そのノードが過去に通信をおこなった実績のあるパートナーノードのリストを要求する。一方で、ノードは他のノードにこれらの情報を積極的に配信する。個々のノード間通信実績は、成功、普通、失敗のどれかにカテゴライズされる。たとえば、相手ノードが新しい、有効なデータを送ってきた場合には成功とみなされる。なぜなら、それは 2 者が同期するのに有益な通信だからである。もしも、相手ノードが新規データを持っていなかった場合は普通の評価になる。その他については、通信は失敗とみなされる。

それぞれのノードは、他のノードとの通信結果を常に監視している。この監視結果はそのノードでのみ使用され、他のノードと共有されることはない。ノードの通信実績は、パートナーノード選びに影響を与える。通信実績は、最長 1 週間保存されるが、ノードの再起動によってリセットされる。このように管理することで、ノードが短期間通信不能に陥った場合にも、他のノードの評価がすぐに落ちるのを防いでいる。

パートナーノードを選択する時、まず候補ノードを選び出すが、個々の候補ノードは、以下のような手順で 500 から 10000 ポイントの重みスコアを与えられる。

- 相手ノードとの通信に、3 回以下の成功または失敗実績しかない場合には、中央値である 5000 ポイントの重みが与えられる。これは、新規ノードが選択されやすくなる効果がある。
- それ以外の場合は、成功回数（s）と失敗回数（f）を使って、以下の式によって重みが計算される：

$$rawWeight = \max\left(500, \frac{s \cdot 10000}{s + 9 \cdot f}\right)$$

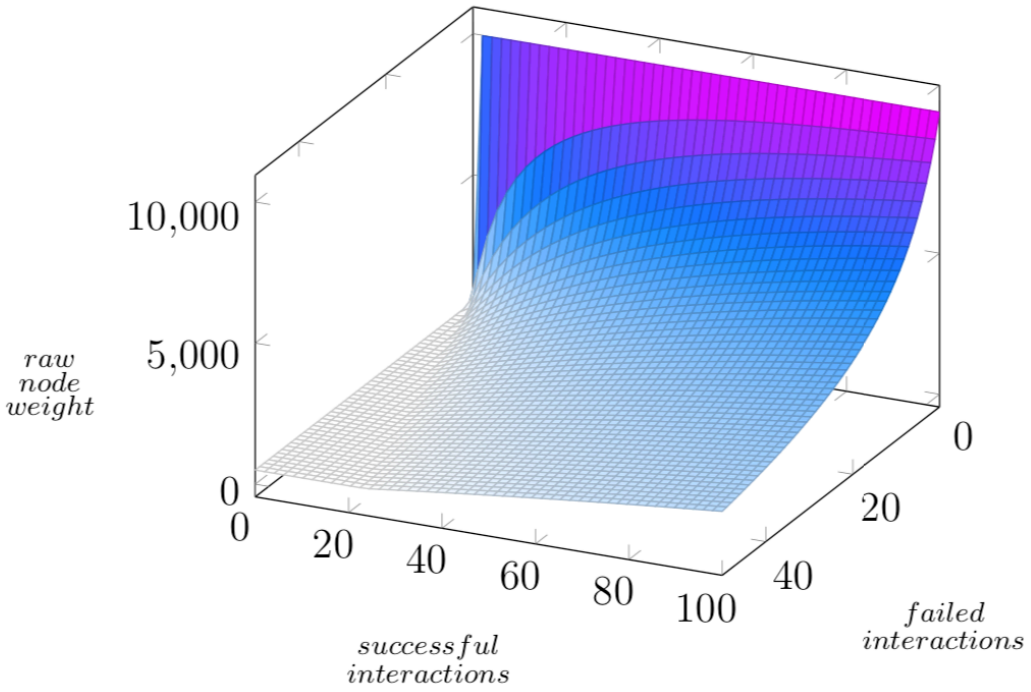


図 27: ノードの重み付けスコア

この計算式には、通信の失敗が失敗した場合に、急激に相手ノードの重み付けと選択確率を低くする効果がある。スコアの最小値（500 ポイント）が与えられているのは、通信の失敗を繰り返したノードにも多少の接続可能性を与えて、通信環境が改善した時にスコアを回復できるようにするためである。

計算された重みスコアに、ノードの種類によって決まる倍率をかけて、最終スコアを計算する。固定ノードの倍率は 2 である。動的ノードは 1 である。連続して接続に失敗したノード（13.3：ノード拒否を参照）については、倍率を 1 ずつ減ずる。これによって、ノードは連続して接続を失敗し続けた動的ノードには接続しなくなる。固定ノードについては、接続確率が半分に減らされる。

候補ノードからの除外は、接続の古さによって決められる。ノードとの接続が切断されると、次の新しいノードに接続して、接続数を維持する。その時に空いている接続枠があれば、下記の式によって新しい候補ノードに接続チャンスが与えられる：

$$P(\text{node is getting selected}) = \frac{node\ weight}{\sum_{\substack{candidates \\ nodes}} candidate\ node\ weight}$$

今回は、接続先ノードの選択基準についてでした。接続先ノードによって不利益と判断されるとスコアが大きく下がる仕様になっていますので、ノードを立てる際はネットワーク環境に注意を払った方が良いと思います。また、スコアは 1 週間でリセットされて、接続が失敗し続けて評判が下がってしまったノードでも環境さえ改善すれば 1 週間後には優良ノードに戻ることができるようです。

13.3 ノード拒否

パブリックネットワークでは、悪意のあるノードがネットワークの正常な運用を邪魔しようとしてくる可能性がある。そのため、リモートノードが悪意あるノードであると思われた場合には、接続を拒否する必要がある。

ノード拒否は、ノードレベルで適用され、ノードの持つリモートノード ID リストに反映される（12.6: ノード探索を参照）。挙動不審ノードはただちに、node:defaultBanDuration の期間、接続が拒否される。そして拒否が解除された後も、ローカルノードによってしばらくの時間（node:keepAliveDuration）、害をなす振る舞いをしたノードとして記憶され、その間に問題行動が繰り返された場合には、さらに長い期間（最大 node:maxBanDuration まで）拒否時間が延長される。接続拒否が継続している期間、拒否ノードとは一切の接続が成立しない。拒否が解除された後は、通常のパートナーノードと同様に取り扱われる。リモートノードの拒否にはいくつかのシナリオが考えられる。ペナルティーの大きさは、その原因によって変化する。

	Connection closed	Remote can reconnect	Remote can be selected	Remote can send data
Consecutive interaction failures	No	-	Yes Static No Dynamic	Yes
Invalid data	Yes - All[38]	No	No	No
Exceeded read rate	Yes - All	No	No	No
Unexpected data	Yes	Yes	Yes (after reconnect)	Yes (after reconnect)

図 28: ノード拒否のルール（図内脚注 38；All というのは、挙動不審ノードへの接続のうち、有害なものだけでなく他の接続もすべて、即時切断されるという意味）。この図の内容については、以下に説明が続きます。

接続が連続して失敗した場合（Consecutive interaction failures）

あるノードとの接続が、ネットワークやサーバーの不調によって、あまりにも長い期間失敗する場合は、しばらくの時間そのノードとの接続を保留して、状態が改善されるのを待ったほうが良い。何回くらいこのような接続失敗が続いたらパートナーノードを拒否扱いにするかは、設定ファイルで指定することができる。ノードが接続拒否されている時間は、ノード選択ラウンドごとに集計されており、これも設定で変更可能である。ノード拒否中は、接続ノードとしては選択されないが、新しいデータ送付は妨げない。つまり、この接続拒否は一時的なものである。

不正データ（Invalid data）

データが不正とみなされる場合はいくつか考えられる。たとえば、リモートノードがフォーク状態にあれば、ローカルノードのチェーンデータとは不整合を起こす可能性がある。1，2 ブロックの小規模なフォークは頻繁に発生する。その場合、送付されたデータは不整合であっても、悪意があるものとはみなされない。なぜなら、リモートノードの内部状態が不正でないことは明白だからである。一方、不正な署名がされたデータは、リモートノードになんらかの悪意があることを示している。なぜなら、署名の検証はノードの稼働状態とは無関係だからであ

る。他の検証が不正な場合についても同様なことが言える。これらすべてのシナリオで、リモートノードは全面的に拒否される。

データ転送速度超過（Exceeded read data）

過剰な量のデータを送りつけるスパム行為を防ぐため、個々のノードは通信ソケットのデータ転送量をモニターしている。これによって、問題のあるピアノードが予想以上のデータを送ってきた時に対処が可能になる。設定した時間内のデータ読み込み量が基準を超過した場合、ソケットを閉じてノードを拒否する。最大読み込みデータ量は設定ファイルに記述される。

予期しないデータ受信（Unexpected data）

ノード間通信をしている間、ローカルノードがリモートノードからデータを受け取る予定がない時間が生まれる。そのような期間に、リモートノードがさらにデータを送りつけてくる場合、プロトコール違反であり接続は切断される。このシナリオでは、接続はすぐに切断されるが、ノード拒否には至らない。

さまざまなシナリオでの、ノード拒否について書かれています。署名の改ざんや、スパムデータの送り込みには非常に厳しく、ネットワーク不調やデータ転送ミスには寛容です。たとえ一度拒否されても、一定期間経てば復活できるため、サーバー設定直後の不調などは、あまり気にしなくても大丈夫そうです。一時的に接続が拒否されたように見えても、時間が経てば回復するでしょう。

14. 合意形成

「わかるだろ？われわれは時々、自分たちが作り出した虚構の世界で暮らしているような気がするんだ。何が良くて、何が悪いのかあらかじめ決めてあって、自分たちが存在する意味を後付していく。そうしながら、自分たちのために作り出したものと、一生格闘し続けるんだ。問題なのは、みんなが違った価値観を持っているということ。だから、お互いに理解しあうのは難しいと感じるのさ。」

- オルガ・トカルチュク（ポーランドの小説家）

ビザンチン合意は、すべての分散型システムが直面する問題である。問題の核心は、個々に独立したプレイヤーが、不正なことをせずにどうやって協力し合うかということである。ビットコインの革新的発明は、Proof of Work (PoW)を使って、この問題を解決したことである。新しいブロックがビットコインのメインチェーンに送り込まれるたびに、すべてのマイナーたちが次のブロックを見つける競争を始める。すべてのマイナー（マイニングをおこなうプレーヤー）が、メインチェーンを伸ばすことに集中し、フォークすることを望まない。なぜなら、最大のハッシュパワーを蓄積したチェーンが勝ち残るからである。マイナーは、できるだけ速くハッシュを計算して、現在のネットワーク難度目標を満たす（ハッシュの先頭に 0 が並ぶ）ブロック候補を作ろうとする。マイナーがブロックをマイニングできる可能性は、ネットワーク全体のハッシュパワーに対するマイナーのハッシュ生成能力に比例する。その結果必然的に、計算力戦争を引き起こして、大量の電力が消費される。

Proof of Stake (PoS) [KN12][BCN13] ブロックチェーンは、ビットコインの後に作り出された、新しい合意形成メカニズムである。PoS ブロックチェーンは、大量の資源を消費しない、新しいビザンチン合意形成方法を示した。原理的には、これらのブロックチェーンは、一点を除いてビットコインとほぼ同じ振る舞いをする。ノードの相対的ハッシュレートを使う代わりに、ノードが持つ相対的ステーク（資産）量によって、ブロック形成のチャンスを得る。より多くの資産を持つアカウントが、より多くのブロックを作ることができるため、富めるものが更に富むことになる。

Symbol は、トークンホルダーよりもトークンユーザーを尊重する改良型 PoS アルゴリズムを使う。この新しい重み付け方法は、アカウントのインポータンスという全体的なスコアを計算しながら、ネットワークのパフォーマンスやスケーラビリティを犠牲にすることがない。

健全なエコシステムを構築するためには、いくつか解決しなければならない問題がある。他の状況が同じならば、より多くの資産を持つアカウントの方が、多くのトランザクションを生み出し、より多くのノードを動かしていればゲームを有利に進められて、その結果報酬も多くなる。まず、多くの資産バランスを持ったアカウントほど、ネットワークにおいて大きなステークを持ち、結果として全体としてのエコシステムを成功に導こうとするインセンティブが強く働く。アカウントが持つ通貨量が、ステークの指標となる。次に、アカウントがトランザクションを発生させて、ネットワークを活性化するようにすべきである。ネットワークの利用率は、そのアカウントが支払ったトランザクション手数料の総額で近似できる。3 つ目に、（ノードを運用する）アカウントを持つユーザーがネットワークをより強固で安全なものにしようと考えるよう導くことが必要である。これは、ノード保有者がブロックの受益者となる回数と相関する（脚注；もしすべてのアカウントがノードを運用したとすると、ステークと強く相関する単純な数値になる。しかし実際には、ノードを運用するアカウントを他のアカウントから差別化する意図がある）。ノードの保有者は、ノードごとに指定できる受益者に関する完全な支配権を持っており、寛大なノードオーナーならば、彼らに多くの利益を分配することもあるだろう（いわゆる還元ノードや寄付ノード）。

インポータンスは、network:importanceGrouping 設定のブロック数ごとに再計算される。ブロックごとでないのは、インポータンス計算の負荷が比較的大きいため、それがブロックチェーンに負担をかけないようにするためである。さらに、このような間欠的なインポータンスの再計算は、自動的に状態変化を熟成させる作用がある（細かなノイズを吸収して数値を安定させるという意味か？）。つまり、間欠的な再計算は、ブロックごとの再計算よりもメリットが多いというわけである。

アカウントが行儀良く行動するよう仕向けるためには、過去に多くの実績を上げたことの評価が重要で、しかしそれだけでは永続的な利益を得られないようにすべきである（ステークのみを評価するとアカウントの保有資産＝過去の実績のみで評価が固定します）。その点、トランザクションやノードスコアによるインポータンス上昇は、一過性である。これらの活動による効果は、network:importanceGrouping で定められたブロック数の 5 倍のブロックが生成される間続く（デフォルトでは 720 ブロック＝約 6 時間）。

PoS+の 3 つのポイントである、ステーク、トランザクション、ノードについて書かれています。Symbol の PoS+では、計算負荷を下げて PoI のスケーラビリティ問題を解決しつつ、トランザクションを発生させる行動やノードを運用することも、考慮されます。今のところその比重は大きくないですが、トランザクションが活発になれば変わってくるでしょう。

14.1 インポータンス比重計算アルゴリズム

network:minHarvesterBalance 以上の残高を持つアカウント（現状では Symbol メインネットで 10000XYM, dHealthNetwork で 2000DHP）は、すべてインポータンス計算に参加する高価値アカウントと呼ばれる。これらの高価値アカウントであることは、ブロック生成に関与できるアカウントの必須要件である（8.3：ブロック生成を参照）。言い換えると、直近のインポータンス計算の結果がゼロではないことは必要条件であるが、ブロック生成に関しての十分条件ではない。

アカウントのインポータンススコアは、後述の 3 つの要素に分解される：ステーク、トランザクション、ノードである。

アカウント A のステークスコア（S_A）は、高価値アカウント全体が保有する通貨量に対する、アカウント A が持つ通貨量の割合である。この割合は、全流通通貨量に対する、アカウント A が保有する通貨量の割合よりも大きな値になる（全流通通貨にはインポータンス計算に参加しない少額アカウントや、取引所アカウントなどが含まれているからだと思われます）。B_A をアカウント A が保有する通貨量とすると、アカウント A のステークスコアは次のように書くことができる。

$$S_A = \frac{B_A}{\sum_{a \in high\ value\ accounts} B_a} \tag{13}$$

次に、トランザクションスコア（T_A）は、一定期間内に、高価値アカウント全体が支払った手数料に対する、アカウント A の支払った手数料が占める割合である。FeesPaid(A)を、A が期間 P の間に支払った手数料とすると、トランザクションスコア T_A は以下の式で表される。

$$T_A = \frac{\text{FeesPaid}(A)}{\sum_{a \in high\ value\ accounts} \text{FeesPaid}(a)} \tag{14}$$

最後に、ノードスコア（N_A）は、アカウント A が（あるノードの受益者であった場合に、そのノードのハーベスト回数分の貢献があったと考える）、一定期間 P の間に受益者となった回数の、高価値アカウント全体が受益者となった総回数に対する割合である。BenficiaryCount(A)を、A が受益者になった回数とすると、N_A は以下のようになる。

$$N_A = \frac{\text{BeneficiaryCount}(A)}{\sum_{a \in high\ value\ accounts} \text{BeneficiaryCount}(a)} \tag{15}$$

これらのうち、トランザクションスコアとノードスコアを、アクティビティスコアと呼ぶことにする。なぜなら、このふたつは、アカウントの持つステークではなく、活動量によって動的に変化するスコアだからである。トランザクションスコアは（アクティビティスコアの）80%の重み付けをされ、ノードスコアは 20%である。さらに、これらを組み合わせたアクティビティスコアは、アカウントの残高（ステーク）によって貢献度が変化する。ステークが大きくなるほど、アクティビティの寄与は小さくなる（脚注；アクティビティスコアはこのダンピングを受けた後に network:importanceActivityPercentage によって決められた割合で、インポータンス計算に寄与する）。これによって、残高が少ないアカウントでも、残高が多いアカウントに対抗できるようになっている。言い換えれば、高ステークアカウントから、低ステークアカウントにインポータンスを傾斜配分することであり、PoS が持つ、富めるものがさらに富むという法則を多少なりとも緩和することができる。ステークに対して、活動量が与える影響力は、network:importanceActivityPercentage で設定できる。この値が 0 であれば、Symbol は純粋に PoS のブロックチェーンとして振る舞う。一方、この割合を高く設定しすぎると、大きなインポータンスを少ないステークで獲得できるため、51%攻撃を受けやすく、セキュリティーレベルが低下する。

ネットワーク効率を最適化するために、アクティビティースコアに関する情報は、直近のインポータンス計算において高価値（つまり 10000XYM または 2000DHP 保有）であったアカウントについてのみ集計される。インポータンスの再計算までは、新しいデータは作業ボックス（working bucket）に入れられる。インポータンス計算が更新されるたびに、作業ボックスはひとつずつシフトして、最も古いものが取り除かれ、新しいボックスが作られる。それぞれのボックスは、最大でも 5 回のインポータンス計算にしか用いられない。そのため、アクティビティースコアは速やかに更新される。

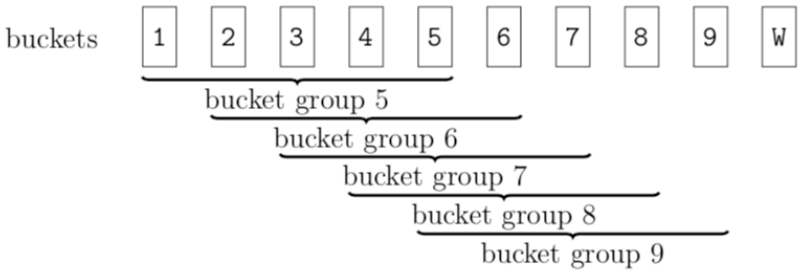


図 29: アクティビティーbuckets（本文ではボックスと意訳）

network:totalChainImportance 設定は、ネットワークに存在するすべてのアカウントに分配されたインポータンススコアの総量を定めている。つまり、あるアカウント A のインポータンス I'A は以下のように計算される（脚注；アクティビティースコア項目がゼロだった場合については別に定める。トランザクションあるいはノードスコアのいずれかがゼロの場合は残った方の貢献度が 100%になる。両方がゼロの場合は、ステークスコアの貢献度が 100%になる）：

$$\gamma = \text{importanceActivityPercentage}$$

$$ActivityScore'_A = \frac{\text{minHarvesterBalance}}{B_A} \cdot (0.8 \cdot T_A + 0.2 \cdot N_A)$$

$$ActivityScore_A = \frac{ActivityScore'_A}{\sum_{a \in high\ value\ accounts} ActivityScore'_a}$$

$$I'_A = \text{totalChainImportance} \cdot ((1-\gamma) \cdot S_A + \gamma \cdot ActivityScore_A)$$

解説；minHarvesterBalance/BA は、ハーベストするために必要な最小通貨保有量を現在のアカウントが保有する通貨量で割ったもので、例えば 10000XYM が最小保有量（minHarvestBalance）で、アカウント A が 100000XYM を保有していた場合は、minHarvesterBalance/BA は 0.1 になります。10000XYM の場合は 1.0 です。これがアクティビティースコアに掛けられるため、保有通貨が少ないほどアクティビティスコアは大きな値になります。これが少額アカウントブーストの正体です。現在はトランザクションが少なく、アクティビティスコアの関与率が低いため、あまり効果は感じられないかもしれません。

最終的なインポータンススコア IA は、ひとつ前の I'A と、新しい I'A のどちらか小さい方が選ばれる（別のアカウントに通貨を移動させてインポータンスを上げようとしても、もう一回分インポータンス計算を待たなければその効果は出ない）。これはステーク分割攻撃への対策で、不要なステークの移動を抑制する働きが期待される。また、スケール補正はおこなわれないため、高

価値アカウントのインポータンス I_A を合計した値は、network:totalChainImportance よりも小さくなるだろう。

ここの説明だけでは完全には理解できませんが、ハーベスト可能な最小 XYM 保有量が 10000XYM とすると、10000XYM より少しだけ多く保有するアカウント同士で XYM を投げあって、アクティビティースコアを上げるのが最も効率良いように思えます。また、保有する XYM を 10000XYM 保有の複数アカウントに分割した時のブースト効果については、次の項以降で詳しく述べられます。

14.2 シビル攻撃

P2P ネットワークへのシビル攻撃（重度の解離性同一性障害、いわゆる多重人格、を持つ女性の医学的治療を詳述した小説 Sybil（邦題；失われた私）にちなむ。脚注；15：ファイナライゼーションに追加情報あり）は、攻撃者が複数の ID を持ち、本来あるべきよりも大きな影響力を与えたり、なんらかの利益を得ようとする行為である。Symbol においては、インポータンスを不正に上げようとする試みが該当するだろう。インポータンススコアの各項目（ステーク、トランザクション、ノード）は、そのような攻撃に対する抵抗力を持つ必要がある。

「14.1：インポータンス比重計算アルゴリズム」で説明したように、アカウントのアクティビティスコアの底上げ率は、保有資産量によって減少する。そのため、アカウントの残高を複数のアカウントに分散することで、それらの平均の（アクティビティスコアの）減少率を小さくすることができる。アカウント分割の前後で、トランザクションなどの活動量が変化しないと仮定すると、分割によって合計のインポータンスは高くなる（脚注；これは 1 つのアカウントを分割した場合にのみあてはまる。多数のアカウントが細かく分割されると、インポータンス計算は全体に対する相対的なものなので、効果は減少する）。この効果はすでに織り込み済みで、利益を生み出す行動を引き出すためのものである。なぜなら、インポータンスを底上げするためには、（資産を分割するための）トランザクションを生成する活動が必要になるからである。トランザクションスコアを維持するために、複数のアカウントからトランザクションを生成して、手数料を支払おうとする行動につながる。ノードスコアを維持することは、さらに多くのノードを立ち上げて、ネットワークを拡大するモチベーションにつながる。どちらもアクティビティスコアを高く維持するためのモチベーションになるわけです。

μ を minHarvesterBalance（最小ハーベスト残高）とし、攻撃者がトータルで N・μ 通貨（原文では currency、つまり XYM のこと）を保有しているとする。この条件で、下記の 2 つの極端な例を比較してみる：

1. 攻撃者は、1 つのアカウントに N・μ XYM を保有する。
2. 攻撃者は N 個のアカウントに、それぞれ μ XYM を保有する。

ステークスコアの底上げ

上記の 2 つの場合、攻撃者が保有する残高の総額は同じである。よって、ステークスコアも同じとなり、アカウントを分割するメリットは無い。式で書くと以下のようになる。

$$B_A = \sum_{a \in \{1,\dots,N\}} B_a \tag{16}$$

ノードスコアの底上げ

Symbol では、ノードを運用者が、ノードがハーベストするブロックごとに手数料の一部を徴収する設定ができる。アカウントがその受益者として選ばれることで、ノードスコアはわずかに底上げされるだろう。

上記 1 と 2 のどちらの場合も、（ノードを 1 台しか運用せず、受益者を自分が持つアカウントのどれかに指定する限り）攻撃者が受益者となる回数は変わらない。つまり、ノードスコアは同じであり、アカウントを分割することによる不当な利益は存在しない。式で書くと：

$$\mathrm{BeneficiaryCount}(A) = \sum_{a \in \{1,\dots,N\}} \mathrm{BeneficiaryCount}(a) \tag{17}$$

攻撃者は、多くのノードを運用することで、より多くの委任ハーベスターを集めるかもしれない。しかし、これは悪い効果というより、むしろメリットである。これによって、ネットワークを構成するノードが増えれば、ネットワークはより強固になるからである。

攻撃者は、一台の物理サーバーに N 台の仮想ノードを設定することもできるだろう。それぞれの仮想ノードは、ネットワークからは普通のノードと同じように取り扱われる。そして、物理

サーバーは通常の N 倍の通信を受け入れることになる。これはすなわち、すべての仮想ノードが非常に強力な物理サーバー上に設置されているということであり、脆弱な物理サーバーに比べて、ネットワークが得る利益はむしろ大きい。

トランザクションスコアの底上げ

トランザクションスコアは、純粋に手数料に比例する。ひとつの高額保有アカウントが X という手数料を支払うのも、N 個の小さなアカウントが、それぞれ X/N の手数料を支払うのも同じことである。式にすると：

$$\mathrm{FeesPaid}(A) = \sum_{a \in \{1,\ldots,N\}} \mathrm{FeesPaid}(a) \tag{18}$$

トランザクションスコアを底上げする唯一の方法は手数料攻撃である。これについては、「14.4 : 手数料攻撃」で詳しく説明する。

Symbol では、アカウントを細かく分割して、インポータンスを上げる行為は、むしろネットワーク全体の利益につながるという考え方をします。その結果としてトランザクションやノードの数が増えて、ネットワークはより活性化されるという考え方は、長年 NEM で PoI を運用した経験から得られたものなのでしょう。

14.3 Nothing at Stake 攻撃

PoS コンセンサス・アルゴリズムに共通する脆弱性が、nothing at stake 攻撃問題（脚注；15：ファイナライゼーションを参照）である。この脆弱性を狙った攻撃は、ブロック生成にかかるコストが無視できるほど小さい場合に起こりうる。この攻撃のパターンには 2 種類ある。

最初のパターンは、攻撃者を除くすべてのハーベスターが、すべてのフォークしたチェーンでハーベストをおこなう場合である。説明を簡単にするために、フォークしたチェーンが 2 本だと仮定する。攻撃者は、一方のフォークチェーンで支払いを実行する。そして、すぐにもう一方のフォークチェーンでハーベストを開始する。攻撃者が十分な量のインポータンスを持っていれば、攻撃者が支払いをしたフォークチェーンよりも、ハーベストしたフォークチェーンの方が優位になり、より高いチェーンスコアを獲得するだろう（脚注；ここでは、攻撃者が一人

であるか、すべての攻撃者が結託して、支払いがおこなわれたチェーンでのハーベストを中止すると仮定している）。その結果、攻撃者の支払いは、メインのチェーンには書き込まれず、攻撃者に差し戻される（が、それでも商品は届く）。

この攻撃に対しては、Symbol には 3 種類の防衛策が存在する。1 つ目は、攻撃者は優位になるようなフォークチェーンを作り出すのに、限られた時間、すなわち network:maxRollbackBlocks（巻き戻し可能な最大ブロック数）しか与えられないことである。もし、商品の発送をこの巻き戻し可能最大ブロック数に達するまで待てば、攻撃は失敗する（支払いが無かったことがわかるため、商品発送はされない）。2 つ目は、nothing at stake 攻撃を成功させるためには、かなりの量のインポータンスを持っていなければならない（脚注；理論的には、攻撃者は network:minHarvesterBalance の残高を持っていれば攻撃は可能である。しかし、現実的には攻撃が成功するためには、巻き戻し不可能になる前に確実にハーベストできるだけのインポータンスが必要になる）。3 つ目は、この攻撃が成功した場合、通貨の価値が下がるということである。攻撃者以外のハーベスターが両方のフォークチェーンでハーベストすることが、この攻撃には必要だが、利益を最大にしたい他のハーベスターたちは、（通貨価値につながる評判を下げないために）攻撃者とは逆のチェーンでハーベストをおこなおうとするだろう。

もう一方の攻撃パターンは、ひとりの攻撃者がすべてのフォークチェーンでハーベストし、どのチェーンがメインになろうとも確実に手数料を獲得しようとする場合である。攻撃者は、ネメシスブロックの次の 2 番めのブロックから始まる、すべてのフォークチェーンでハーベストすることが可能である。そして、最も多くの手数料が獲得できるチェーンを探す。ブロックの承認（＝ハーベスターの選出）は、確率的におこなわれるため、時間さえかければ理論的には、攻撃者がすべてのブロックをハーベストするような「完璧な」チェーンを作り出すこともできる（確率論的には起こり得るというレベルの話ではないかと思います）。

ほとんどの、このような理論上の nothing at stake 攻撃は、理想化されたブロックチェーンを仮定しており、攻撃に対抗するプロトコールレベルの安全策については考慮していない。しかし、現実にはこのような攻撃は、攻撃者が持っている資産が充分でない場合は成立しない。先に述べたうち、2 種類の防御策がこの場合にもあてはまる。それに加えて、ブロック難度（8.1：ブロック難度）の変化量は最大でも 5%までと決まっている。そのために、攻撃者のチェーンが優位性を得るためには、ある程度時間がかかり、秘密裏にチェーンを伸ばそうと試みた場合も、その最初期に大きな遅延を生じることになる。このような大きな時間差が生じてしまうと、攻撃者のチェーンが高いチェーンスコア（ブロックスコアの合計）を達成するのが困難になる（8.2：ブロッ

クスコア）。

ステークの成熟率が小さく設定されていること（インポータンス計算が間欠的であることなど）も、この 2 番めのパターンの攻撃を困難にしている。ハーベストしようとする攻撃者のアカウントにとって、2 回の連続したインポータンス計算で、ゼロではないインポータンスを獲得しなければならないルールは、ジェネレーションハッシュの大量生産（脚注; ジェネレーションハッシュとは無関係に、ブロックのヒット値を無理やり下げる（8.3 : ブロック生成参照））攻撃を阻止する。このような強引なやり方のためには、攻撃者は network:importanceGrouping ブロック数のあいだに、（ジェネレーションハッシュの生産に用いた）特定のアカウントに残高を集めて、攻撃の準備を整えなければならない。しかし、その準備の間に承認されるすべてのブロックを知ることはできないため、そのような努力は徒労に終わるだろう。

NIS1 においてもそうであったように、Symbol で nothing at stake 攻撃を成立させるのは、ほぼ不可能であると思われます。0.9.6.3 で導入された VRF なども、意図的フォークを前提にしたこれらの攻撃を、さらに難しくするでしょう。

14.4 手数料攻撃

手数料攻撃とは、高額な手数料を支払うことによってトランザクションスコアを増やし、自分のインポータンスを上げようとする行為である。この攻撃によって、収支がプラスになった場合に、攻撃は成功したとみなされる。

この項では、パブリックネットワークの推奨設定を使用して、シミュレーションをおこなう。設定では、network:totalChainImportance を 90 億、network:importanceGrouping を 359 ブロック、minHarvesterBalance を 10000XYM とした。さらに、network:importanceActivityPercentage を 5% に設定すれば、トランザクションスコアの係数（計算式については、14.1：重み付けアルゴリズム参照）は、インポータンスの 4%となる。

ラージアカウント

まず多くの通貨を保有しているラージアカウントについて考える。特にアクティビティスコアをブーストしなくても、インポータンス再計算の期間（359 ブロック）に一回のハーベストを

おこなえるくらいの資産を持っているアカウントである。90 億の XYM のうち、20 億 XYM がハーベストに参加（残りは基金や取引所、参加しない少額アカウントが所有）しているとすると、このアカウントは少なくとも 557 万 XYM（ハーベスト参加 XYM の 1/359 であり、359 回の抽選で一回当たると期待される）を所有していることになる。

このアカウントが、インポータンス再計算までに自分がハーベストするブロックで、トランザクションに高額の手数料を設定すれば、（手数料はハーベストした自分に返還され）利益が見込める可能性が出てくる。つまり、この行動によって次のインポータンスが上昇すれば、（さらに次のインポータンス再計算までの間）それだけ多くのブロックをハーベストでき、手数料を多く得ることができるだろう。しかし、この行為にはリスクが伴う。（フォークによって）ブロックスコアがより高いブロックが並立すれば、自分が高額の手数料を支払ったブロックがそれに置き換えられてしまうこともある。狙ったブロックが巻き戻されてしまうと、高額の手数料の入ったトランザクションは、未承認トランザクションキャッシュに差し戻されて、次のブロックで他のハーベスターの手に渡ることになる。このシナリオは、高い手数料を支払った分、損失となる。

P を、（インポータンス再計算までに）フォークが起きる確率とする。F は、ブロックに含まれる高額な手数料、そして$\bar{F}$を、そのブロックに含まれる手数料の平均値とする。その場合の期待損益値 EV は、以下のとおりである：

$$\beta = 0.04 \cdot \frac{1}{557} \cdot \frac{F}{359 \cdot \bar{F} + F} \qquad \text{(importance boost)}$$

$$EV = \beta \cdot \frac{359}{P} \cdot \bar{F} - F \qquad \text{(expected value)}$$

（β はインポータンスに加算されるブースト値。トランザクション係数は 0.04。1/557 はステーク量による補正。フォークした場合には次のインポータンス計算後のハーベストで、割増された平均手数料が得られるが、F の方が大きければマイナスになり損失である。）

期待損益値は、P がごく小さい（フォークがほとんど起きない）場合、プラスに転じる。しかし、P または F が大きくなると、すぐにマイナス値（損をする）になる。推奨設定値でネットワークを運用する場合、攻撃者がプラスの期待損益値を得るためには、P は 0.0001 より小さくなければならない。これはすなわち、フォークが 10000 ブロックに 1 回以下しか起きない確率

である。分散型のコンセンサスアルゴリズムを採用する限り、この数値を達成するのはほぼ不可能だろう。1，2 ブロック分の小さなフォークは、頻繁に起きるからである。

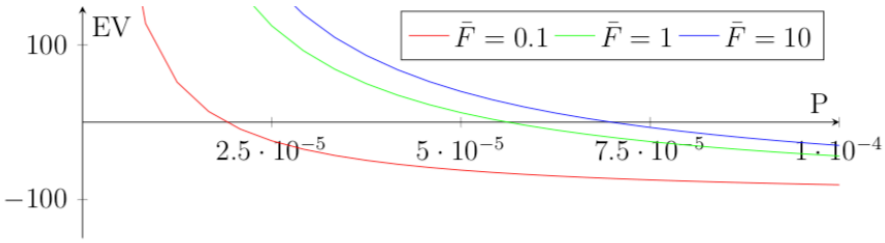


図 30: ラージアカウントによる手数料攻撃のシミュレーション（F = 100）

PoS 系のコンセンサス・アルゴリズムは、自分が次のハーベスターになるかどうかをある程度予測して、手数料アタックをかけることができます。これは意外でした。Symbol0.9.6.3 で導入された VRF によるハーベスター予測を困難にする手法は、このような攻撃もほぼ完全に防いでしまいます。図 30 は、557 万 XYM を保有するラージアカウントが、100XYM の手数料を支払ってトランザクションを発生し、そのブロックを自分でハーベストすることを狙った場合のシミュレーションです。ブロック平均手数料が 10XYM くらいまで上がったとしても、利益を出すのは非常に難しいみたいですね。1,2 ブロックの小さなフォークが頻繁に起きることも、このような攻撃を無効化しているというのも面白いです。次項は、もっと少ない資産しか持たないアカウントではどうかです。

スモールアカウント

次に、network:minHarvesterBalance 分（10000XYM）しか持っていないスモールアカウントについて考える。このアカウントはまず、2 回のインポータンス計算の間に大きな手数料を伴ったトランザクションを送るとする。これらの手数料は誰か他のハーベスターに渡る。なぜなら、該当するインポータンス再計算の期間にハーベストをおこなうには、アカウントの持つインポータンスが低すぎるからである。支払った高額の手数料は、次のインポータンス再計算で評価され、その先の再計算間隔（359 ブロック）の間、1 回のハーベストが期待できる程度までインポータンスを上げることができる。この時点から、スモールアカウントは、先のラージアカウントのように振る舞えるようになる。そして、ここからは再計算期間ごとに、自分がハーベ

ストするブロックに高額な手数料を送り込むこともできるようになる。そして、ブロックをハーベストできる確率が上がる事による報酬額が、支払ったコストを上回ることが期待できる。

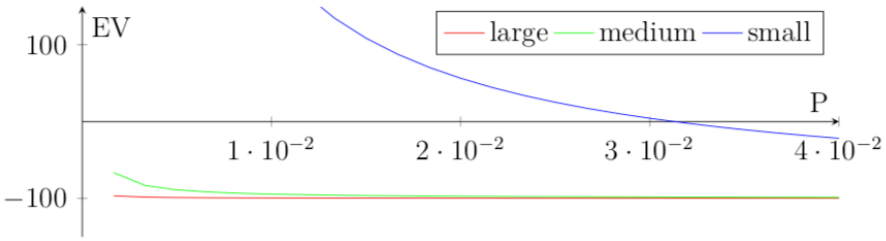


図 31: 残高（large, medium, small）よる手数料攻撃収支（F = 100, $\bar{F}$ = 1）

P を、損失（失敗）につながるようなフォークが起きる確率。F はブロックに送り込む高額手数料。$\bar{F}$をブロックの平均手数料額とする。損益期待値 EV は、インポータンスを上げるために使った手数料を除いて、下記のように計算できる（脚注；ラージアカウントの場合と似ているが、ステークによる補正率 1/557 のところが 1/1 になって消えている）：

$$\beta = 0.04 \cdot \frac{F}{359 \cdot \bar{F} + F} \qquad \text{(importance boost)}$$

$$EV = \beta \cdot \frac{359}{P} \cdot \bar{F} - F \qquad \text{(expected value)}$$

ラージアカウントとの違いは、P がより大きい（フォークしやすい）条件でも、収支がプラスになることである。つまり、手数料攻撃手法によって、ラージアカウントよりもスモールアカウントが優遇されることになる。ラージアカウントでは、所有する資産によって、アクティビティスコアの影響度が大きく減少するためである。具体的には、ラージアカウントのアクティビティスコアは 1/557 まで減少させられるのに対して、スモールアカウントにはそのような影響が無い。

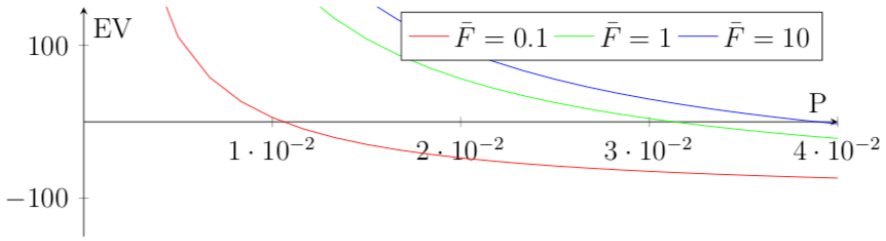


図 32: スモールアカウントにおける収支期待値のシミュレーション（F = 100）

平均手数料報酬$\bar{F}$が増加するほど、EV も増加する。P または F が増加すれば、収支がマイナスになるのはラージアカウントと同様であるが、推奨ネットワーク設定値においては、スモールアカウントの手数料攻撃が成功するためには、P は 0.05 以下であれば良い。これは 20 ブロックごとに 1 回のフォークが起きる率と一致する。現在の分散合意形成において、この数値は充分に達成可能である。

さらなる考察

この攻撃により、スモールアカウントはプラスの収支を期待できるわけだが、複数のアカウントが同時に同じことを試みるとその効率は悪化する。期待値がプラスであるため、利益を最大化したい者はほぼ確実にこの攻撃を試みるだろう。より多数のアカウントが参加することで、個々のアカウントが受けるインポータンスブーストは、すぐに低下してしまう。その結果 EV も減少することになる（脚注；より多くのアカウントが高額な手数料を支払うことで、平均手数料額（$\bar{F}$）が増加する。その結果、多くの攻撃者の参加によって個々のインポータンスブーストが減少しても、F が変化しない限り、EV は増加する。これはブロックの価値を上げる予測可能な現象である）。

さらに、この攻撃を実行可能なスモールアカウントの数には上限がある。この攻撃が成功するためには、インポータンス再計算がされる 359 ブロックの間に、最低 1 回はハーベストする必要がある。そして何よりもまず、トランザクションスコアを上げて、インポータンスをブーストしないとならない。しかし、インポータンスをブーストできるアカウントの数は、トランザクションスコアがインポータンスに影響を与える率や、再計算頻度が固定されているため、理論的上限がある。推奨ネットワーク設定のもとでは、この上限はだいたい 0.04 / (1/359) = 14.36 アカウントである。

N を攻撃を試みるスモールアカウントの数とし、P をフォーク確率、F をブロックに送り込む高額手数料、$\bar{F}$をブロックごとの平均手数料額とすると、収支期待値 EV は下記の計算式で表せる：

$$\beta = 0.04 \cdot \frac{F}{359 \cdot \bar{F} + N \cdot F} \qquad \text{(importance boost)}$$

$$EV = \beta \cdot \frac{359}{P} \cdot \left(\bar{F} + \frac{(N-1) \cdot F \cdot P}{359} \right) - F \qquad \text{(expected value)}$$

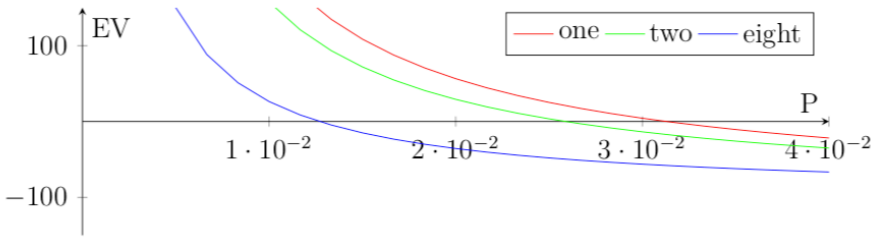


図 33: 攻撃者数（1, 2, 8 人）と、収支期待値（F = 100, $\bar{F}$ = 1）

Symbol では、10000XYM しか保有していないスモールアカウントが、高額な手数料を使ってインポータンスブーストを狙うことを認めています。なぜならそれによってネットワークが活性化し、ブロック報酬が全体的に上昇するからです。しかし、現在のトランザクション量では、平均手数料報酬が$\bar{F}$がほぼ 0 に近いので、これらの詳細なシミュレーションのような状況は、現実には起きていないと思われます。

15 ファイナライゼーション

「ようやく終わったという感じだ」

- パトリック・ネス（イギリスの作家）

CAP 定理（分散コンピュータシステムのマシン間の情報複製に関する定理）によると、ネットワークに障害や分断があれば、分散型システムは「一貫性」か「可用性」のどちらかを選ばねばならない。

一貫性を持つシステムとは、いかなる２つのノードに送ったリクエストに対しても、同じ値が返ってくるという意味である。もしも、その値がシステム全体の合意を得られていなければ、リクエストは失敗する（値は返ってこない）。このようにして、すべてのクライアントがシステムの持つ唯一値を共有することができる。BFT（ビザンチン障害耐性）システムの多くはこの一貫性を重視して、ネットワーク停止のリスクを容認しようとする。純粋な一貫性システムでは、すべてのステークホルダーが各ブロックについて投票して、2/3 の賛成票が得られたブロックだけが次のブロックとして承認される。しかし、この投票中に問題が起きた場合には、ブロック生成に遅れが生じることになる。

一方、可用性を持つシステムの場合は、2 つのノードに送られたリクエストに、直ちに値が返される。ただし、返ってきた値が同じであるという保証はなく、クライアントはシステムが複数の値を返してくることを許容しなければならない。PoW や、NXT 型 PoS を採用したシステムの多くは、まずこの可用性を重視して稼働し、最終的には合意を得るような設計をされた。その結果、分岐した（場合によっては解決不可能な問題を内包した）システム状態が拡散されるリスクが生じた。

Bitcoin のような可用性重視の PoW 型システムは、単純なフォーク解決策を持つ。すなわち、最も仕事をしたチェーンを選ぶというものである。ネットワークが 2 つに分裂し、しかも等しいハッシュ計算パワーを持っていた場合、分裂した状態のまま、お互いにそれと知ることもなく、状況は進んでいくだろう。もし、この 2 つのネットワークが再接続した際に、非常に大きな（そしておそらくは深い）代償を払って、フォーク解決をしなければならなくなる。

Symbol は常に一貫性よりも可用性を重視しているが、オリジナルのコンセンサスシステム（14：コンセンサスを参照）に加えて、ファイナリティを実現するガジェットを追加することができる。このガジェットは、ブロック生成やコンセンサス形成と独立して稼働して、BFT 投票機能を Symbol に追加する。Symbol は、決定論的ファイナライゼーション（ガジェットが稼働している場合）と、確率論的ファイナライゼーション（ガジェットが稼働していない場合）の両方を利用することができる。ガジェットは、node:maxRollbackBlocks が 0（ゼロ）にセットされた時、自動的に稼働するようになっている（脚注；この場合、ファイナライゼーション機能拡張も当然オンになっていなければならない）。

ガジェットを使用するアプローチは、Polkadot が使っている GRANDOPA [SK20]に習って設計された。一方 GRANDPA は CASPER [BG17]の影響を受けており、また CASPER は PBFT [CL99]をもとにしている（PBFT に関する日本語解説記事；https://hazm.at/mox/distributed-system/algorithm/transaction/pbft/index.html）。伝統的に、PBFT は 3 つのメッセージタイプを使う：前準備（pre-prepare）、準備（prepare）、決定（commit）である。前準備メッセージは、投票ラウンドをスタートする時に使われるが、Symbol では使用しない。代わりに、ネットワーク時間の経過（16：時刻同期を参照）によって、投票ラウンドが開始される。PBFT の準備と決定メッセージは、Symbol では投票準備（prevote）と決定準備（precommit）メッセージがほぼ担っている。

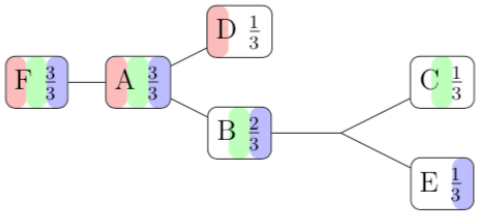
すでに予告されたとおりに、Symbol のファイナリティは Polkadot が開発した GRANDOPA システムを踏襲した、PBFT（実用的ビザンチン障害耐性）システムを導入したようです。Symbol の基本設計に変更はなく、プラグインや機能拡張としてコンセンサス形成やブロック形成を外から監視する付加システムのような扱いになっており、手軽にオンオフできます。ノード設定で、ブロックの巻き戻し数を 0 にセットすると自動稼働するようで、つまり普通に 400 とかにセットしておけばこれまで通りにファイナリティ無しでブロックチェーンを動かす事ができます。

15.1 概要の説明

ブロックのファイナライゼーションは複雑なプロセスで、2 種類のメッセージ：投票準備（prevote）と決定準備（precommit）を使用する。投票ラウンドの開始時、それぞれの投票人

（ノード）は、自身がつないでいるローカルチェーンに保存されたブロックの情報しか持っていない。他の投票者の情報を持つことは無いものとする。その結果、個々の投票者は、自分が所有するブロックのどれが特別過半数（2/3 以上）の支持を得てファイナライズされるかを、前もって知ることはできない。

ファイナライゼーションの一般的な手続きを図解するために、3 人の等価な投票人が参加しているネットワークを仮定する。あるハッシュ（ブロックのハッシュ値）に対する特別過半数は、3 人のうち少なくとも 2 人の賛同を得なければならない。F は、これまでで最新のファイナライズされたブロックである。



ここからは、3 人の投票人をそれぞれ赤、緑、青で表現する。

ファイナライゼーションは欲深なプロセスで、個々の投票ラウンドでひとつでも多くのブロックをファイナライズしようとする。最初の投票準備（prevote）ステージでは、個々の投票人は最新のファイナライズされたブロック（F）を起点として、ローカルチェーンに対応するハッシュチェーンを作って公開する。上の例では、それぞれの投票準備ハッシュチェーンは以下のようになる：図だとわかりにくいですが、赤ノードは F-A-D, 緑ノードは F-A-B-C, 青ノードは F-A-B-E のチェーンを承認しようとしています。下のリストの a, b, c は、原文では 1, 2, 3 となっていますがわかりにくいので改変しました。

a. 赤：H(F), H(A), H(D)
b. 緑：H(F), H(A), H(B), H(C)
c. 青：H(F), H(A), H(B), H(E)

この時、赤がネットワークの接続不具合によって、緑の投票準備情報しか得られず、青の投票準備情報が分からなかったとする。緑と青については、赤を含めたすべての投票準備情報が得

られたとする。この時点で、それぞれの投票人は以下のような投票準備（prevote）ハッシュチェーンを得ることになる。

1. 赤：
   (a) H(F), H(A), H(D)
   (b) H(F), H(A), H(B), H(C)
2. 緑：
   (a) H(F), H(A), H(D)
   (b) H(F), H(A), H(B), H(C)
   (c) H(F), H(A), H(B), H(E)
3. 青：
   (a) H(F), H(A), H(D)
   (b) H(F), H(A), H(B), H(C)
   (c) H(F), H(A), H(B), H(E)

それぞれの投票人は、（自身と他の投票人）すべての投票準備ハッシュチェーン情報を検査し、現在の投票ラウンドでファイナライズする**可能性のある**ブロックを選定する。赤は、A の特別過半数（2 票）のみを選定するが、緑と青は B の特別過半数（2 票）まで選定できる。（たとえば、赤ノードが持つ情報では H(F), H(A)は 2 票入っているが、H(B), H(C), H(D)は 1 票ずつしか入っていない。緑と青ノードは、H(F)と H(A)は 3 票、H(B)が 2 票、H(C), H(D), H(E)は 1 票ずつ入っている。H(F)はすでにファイナライズされているため除外される。）次の、決定準備（precommit）ステージで、それぞれの投票人は選定したベストブロックのハッシュを公表する：

1. 赤：H(A)
2. 緑：H(B)
3. 青：H(B)

今回は、緑がネットワーク接続障害によって、赤の情報は得られたが、青の情報が得られなかったとする。赤と青は今回は全部の情報が得られたとする。その場合、個々の投票人は以下のような決定準備情報を持つ：

1. 赤：H(A), H(A), H(A), H(B), H(B)
2. 緑：H(A), H(A), H(B)
3. 青：H(A), H(A), H(A), H(B), H(B)

投票人はそれぞれ、すべての決定準備ハッシュを検査して、ファイナライズする最終候補を決める。ここで重要なのは、ひとつのブロックに対する決定準備は、それに至るすべてのブロックへの決定準備情報を含んでいるということである（緑と青の決定準備ブロックは B であるけれど、その一つ前の A も決定準備ブロックとして取り扱われます。それを分かりやすくするために上のリストには票数分繰り返して赤字で補足しています）。緑は、A については特別過半数を得ている（しかし、B については 1 票しか得ていないので、棄却する）。赤と青は B の特別過半数も得ている。その結果、最終決定ステージ（commit stage）では、それぞれの投票人は以下のようなブロックをファイナライズする：

1. 赤：H(A), H(B)
2. 緑：H(A)
3. 青：H(A), H(B)

この後の節では、ファイナライゼーションのより詳細なアルゴリズムを解説する。

ファイナライズの概要説明ですが、説明がややこしいですね。赤のノードは、自分の持つブロックチェーンは D に分岐していますが、投票では B をファイナライズします。この時点で、自分が持つブロックチェーンを A まで巻き戻すことになるでしょう。ファイナライゼーションは、ブロックチェーンの承認とは独立に動く監視システムですから、あるノードの最新承認ブロックが、ファイナライゼーションプロセスによって追認されるまでは、巻き戻される可能性があるものとして取り扱わねばなりません。

15.2 投票ラウンド

ラウンド（finalization round）は、ファイナライズのプロセスを象徴するステップで、エポック（finalization epoch）とポイント（finalization point）からなる。

エポックは、1 グループのブロックを内包する。エポックに含まれるすべてのブロックは、一回の投票によってファイナライズされる。それは network:votingSetGrouping 個のブロックごとに再計算をされる。最初のエポックは、ネメシスブロックのみを含み、あらかじめファイナライズされている。それに続くエポックは、network:votingSetGrouping の整数倍のブロックまでを含まなければならない。エポックがファイナライズされたとみなされるのは、エポックに含まれるすべてのブロックがファイナライズされた時である。エポックの最後のブロックについて、ファイナライゼーション証明が与えられる。

ポイントとは、エポックに含まれる細かなステップのことであり、エポックがファイナライゼーションのどの段階にあるかを示す。こうすることで、同じブロックが複数の投票セットでファイナライズされることを防いでいる。エポックに含まれるすべてのブロックをファイナライズするのに必要な数のポイントが用意される。そして、各エポックの最後のポイントは必ず、そのエポックの最後のブロックをファイナライズする。ブロックがファイナライズされるのは、ポイントがブロック、または、そのブロック以降のブロックをファイナライズした時である。

ネットワークが分断されると、ポイントが新しいブロックをファイナライズできない可能性がある。この現象は、最後にファイナライズされたブロックまでしか、特別過半数の得票数を得られていない場合に起きる。これは、致命的なエラーではなく、ネットワークが分断化する事によって自然に発生する現象にすぎない。その後は、15.5 アルゴリズムに記載された方法に従って、ファイナライゼーションは次のポイントに進行する。

エポックとポイントの違いについての重要な点は、投票セットに対する関連性である。異なるエポックは、異なる（重ならない）投票セットを持つ。同じエポックに含まれるそれぞれのポイントは、（該当する）エポックに関連付けられた同じ投票セットを共有していなければならない。

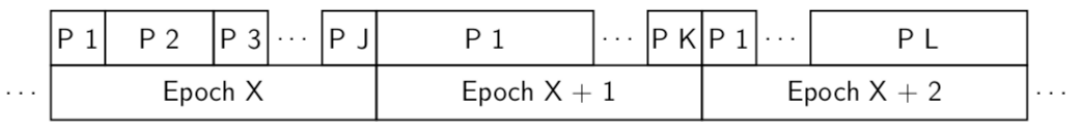


図 34：エポック（Epoch）とポイント（P）の関係

ファイナライゼーションのプロセス内では、エポックという複数のブロックから構成される単位が用意され、ポイントを使ってその状態を管理するようです。エポックは、設定で決められたブロック数ごとに実行され、１ラウンドの投票により１エポックが進行します。

15.3 投票者

下記の条件を満たしたアカウントは、エポックごとの投票に参加する資格がある。

1. 直近のエポックでファイナライズされたブロック高において、ハーベストバランス（残高）が network:minVoteBalance を下回らないこと。
2. 投票鍵（voting key）が StartEpoch≦エポック≦EndEpoch の間で登録されていること。

それぞれのエポックについて、上の条件を満たしたすべてのアカウントが投票セットとみなされる。ここで注意しなければならないのは、投票については（ハーベストとは異なり）インポータンスではなく、残高が使われる点である。これは、インポータンス操作が起きる可能性を排除する（脚注；ネットワーク分断が起きている時、それぞれの分断化されたネットワーク内でインポータンスが計算される。その結果、それぞれの分断ネットワークが異なるアクティビティスコアを算出する可能性がある。インポータンスはネットワーク全体に対する絶対値ではなく、参加しているアカウントの総計から計算される相対値であるため、理論的には複数の分断化されたネットワーク内で、それぞれのインポータンスの特別過半数を獲得した異なるブロックが承認される可能性が生じる）。

投票権のあるアカウントのすべてが、投票すると期待される。好意的な投票者は、有効な投票ラウンド全てに一回ずつ投票するだろう。ただし、すべての投票者が、高い可用性を持ったノードを運用していて、finalization:EnableVoting 設定をオンにしてあるという暗黙の了解のもとである。よって、投票者の挙動が、上の期待に沿わないものであった場合は、ビザンチン問題の発生源と見なされ、その後の投票でペナルティを与えられる可能性がある。投票権は、より多くの残高を持つ投票者により多くの権利が与えられるような重み付けがなされる。

15.4 メッセージ

投票準備（prevote）や決定準備（precommit）用のメッセージは、共通の構造を持っている。

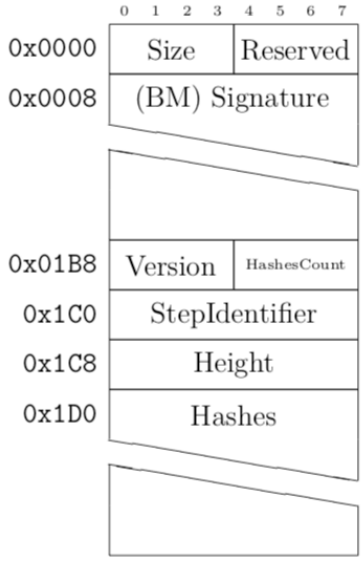


図 35：メッセージのレイアウト

Signature は、メッセージの BM ツリー署名を意味する（Bellare-Miner [BM99], 3.4：投票鍵リストを参照）。メッセージが関係するエポック内では、（ツリーの）ルート公開鍵が、投票鍵と一致していなければならない。（ツリーの）末端公開鍵は、エポックの鍵ツリー内の公開鍵と一致しなければならない。

version はメッセージのバージョンである。

Hashes は HashesCount の数のハッシュを含む。投票準備メッセージは少なくとも 1 つのハッシュを含むが、finalization:maxHashesPerPoint を超えることはない。決定準備メッセージに含まれるハッシュの数はひとつである。

StepIdentifier は、メッセージのファイナライゼーションラウンド数を示す。ポイントの最上位ビットは特別な意味を持ち、0 ならば投票準備メッセージであることを、1 ならば決定準備メッセージであることを示している。

Height は Hashes に含まれる最初のハッシュのブロック高である。

投票権にはインポータンスではなく残高に応じた重み付けがなされます。また、3.4 にある鍵リストを使い、投票に参加しているアカウントを効率よく管理しています。

15.5 アルゴリズム

まず、関数 g(...)を定義して、現在有効な投票力の特別過半数を獲得した最新のブロックをそれに含める。$V_{r,v}$ は、投票者 v のラウンド r での投票準備（BM ハッシュツリー？）とする。$C_{r,v}$ は、投票者 v のラウンド r での決定準備である。$E_{r,v}$ は、投票者 v のラウンド r でのファイナライズ候補とする。これは、単なる推測値であって、決定値ではないことに注意。$E_{r,v}$ は、$g(V_{r,v})$ を含むチェーンの最新ブロックであるはずであり、$C_{r,v}$ の特別過半数を得ていると考えられる。ラウンド終了条件は以下のいずれかになる：

1. $E_{r,v} < g(V_{r,v})$となる場合
2. $g(V_{r,v})$に含まれるブロックのどれもが $C_{r,v}$ の特別過半数を得ることができない場合

投票者 v は、前回のラウンドが終了条件を満たし、それ以前のすべてのラウンドに投票を終えたと確認された時に、次のラウンド r を開始できる。

15.5.1 投票準備

stepDuration の 1 倍の時間が経過した、あるいはラウンド r が終了条件を満たした時、v は投票準備メッセージを送信する。

投票者は、ファイナライズできそうな最良の候補ブロックを選び出す。そして、投票準備メッセージを用意する。このメッセージには、（ローカルノードで）最も最近ファイナライズされたブロックのハッシュ値でスタートし、候補ブロックに至る全てのブロックのハッシュ値を含む。投票準備メッセージ内のハッシュチェーンは、最大で finalization:maxHashesPerPoint 個のハッシュを含むことができる。このチェーンの最後のハッシュは、finalization:prevoteBlocksMultiple の整数倍のブロック高を持つはずである（脚注；実際には、finalization:maxHashesPerPoint は finalization:prevoteBlocksMultiple よりもかなり大きい値になる。また、network:votingSetGrouping はもまた、finalization:prevoteBlocksMultiple の整数倍にな

る）。こうすることで、ノードがより積極的に集めた同一のチェーン候補を、投票準備メッセージとして送り出す可能性が高まる。

未ファイナライズブロックのハッシュは、複数のエポックにまたがってはならない。この決まりにより、どのブロック高においても、ひとつの投票セットがひとつのブロックをファイナライズすることが保証される。そして、動的な投票セット運用が可能になる。

投票準備メッセージは、単独のハッシュの集まりではなく、ハッシュチェーンを含んでいる。Symbol のノードは、（分岐した）ブロックツリーではなく、単一のブロックチェーンを保存しているからである。Symbol は、ハッシュチェーンの塊から仮想的なブロックツリーを構築して、既知と未知の両方の分岐に投票できるように準備する。

コンセプト的には、投票者はひとつのブロック高に 1 度だけ投票する。実際は、投票者は自分の持つ「通貨残高と連動したスコア」を投票準備されたハッシュチェーンに投じる。

実際の投票がおこなわれる過程が見えてきました。投票は時間経過で開始され、あるブロック数（エポックサイズ）の整数倍ごとになされます。その際はブロックのハッシュが連なったハッシュチェーンというものが投票対象になるようです。投票者は、このハッシュチェーンに、自分が持つ通貨残高で重み付けされたスコアを投票します。

15.5.2 決定準備

次に、投票者は $g(V_{r,v}) \geq E_{r-1,v}$ になるまで待つ（ひとつ前にファイナライズされたブロックをノードの投票準備プロセスが追い越すまで）。その後、stepDuration の 2 倍（ラウンドのスタート時刻を基準）時間が経過するか、または、r が完了可能になった時、v は決定準備メッセージを送信する。

投票者は、ファイナライズ候補となり得る最良ブロックを決定する。そして、$g(V_{r,v})$に対応する単一のハッシュ値を含む決定準備メッセージを送信する。

理論的には、投票者はブロック高ごとに 1 回投票する。実際には、投票者は最も最近ファイナライズされたハッシュと決定準備されたハッシュの間のすべてのハッシュに、自身が持つ投票力を投入する（15.1 概要部分で説明しました）。

15.5.3 決定

非同期的に、投票者はラウンド r に対する投票準備と決定準備メッセージを集計する。実際には、これらのメッセージは現在進行中のラウンドか、過去のラウンドのどちらかのものと考えて良い。$g(C_{r,v})$が変化する時、そのブロックと、ローカルにファイナライズされたブロックからそのブロックに至る全てのブロックがファイナライズされる。

エポック e のファイナライゼーションラウンドとポイント p が与えられた時に、ほとんどの場合は、次のラウンドは(e, p+1)になるだろう。ひとつの例外は、エポック e の最後のブロックがファイナライズされた時である。その場合は、次のラウンドは(e+1, 0)になるはずである。(e, p+1)からでも(e+1, 0)からでも処理が開始されることに注意してほしい。そのような場合には、(e+1, 0)のほうが結果的に優先されて、(e, p+1)は完了しないということが起こりうる。

ファイナライズの最終過程についての説明ですが言葉だとすごくわかりにくいですね。特に g という関数が定義され、重要な役割を持っているのですが、ブロックハッシュがつながったハッシュツリーを指しているような気がします。この章の最後に実際例が掲載されているので、それを待ちましょう。

15.6 証明書

ネットワーク負荷を最小限に抑えるために、投票者以外のノードは投票準備や決定準備メッセージを送信しない。その代わりに、これらのノードは finalization:unfinalizedBlocksDuration に基づいて、繰り返しネットワークにプルリクエストを送信し、エポックが終了するごとにファイナライズ証明書をネットワークから取り寄せる。この設定値が 0 の場合は、証明書の要求は、個々のエポックの終了時（つまりファイナライズされた時）になる。それ以外の場合は、最新のファイナライズ未決定なブロックから、finalizaiton:unfinalizedBlocksDuration 分以上遡ったところにある最新のファイナライズブロックについて証明書が要求されるだろう。検証にあた

り、これらの非投票ノードは、proof エポック（最新のファイナライズブロックまでのこと？）を含む、それまでのエポックすべてを一度にファイナライズする。

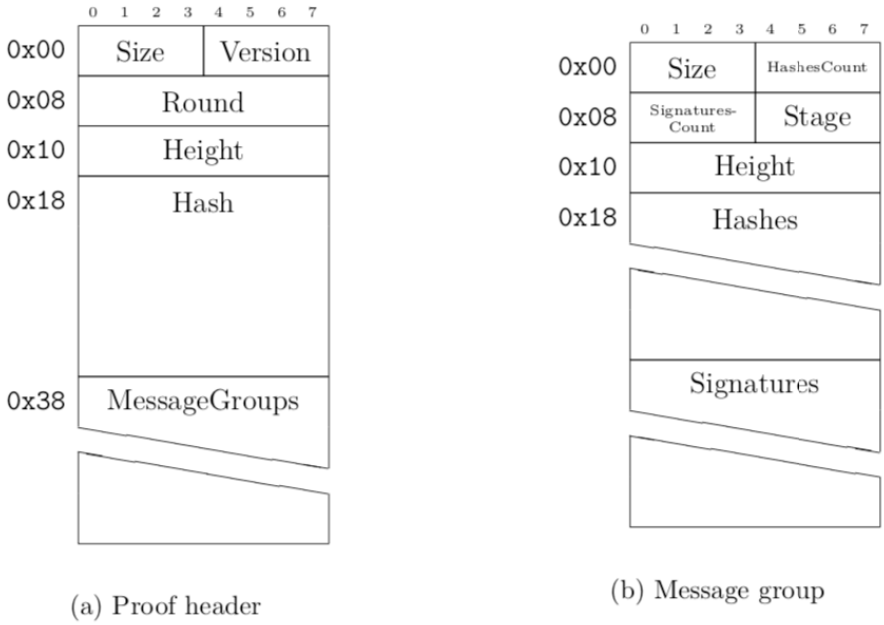


図 36：ファイナライズ証明書のレイアウト

ファイナライズ証明書は、ヘッダーとメッセージグループからなる。ヘッダーには、証明書発行までにファイナライズされた最新のブロックが示されている（脚注；技術的には、証明書は最新のブロックと、それにつながる過去のブロックすべてをファイナライズする。これによって単一チェーン形成にも寄与する）。この最新ブロックは、ブロック高とハッシュの両方で指定される。ヘッダーには、ファイナライズが実行されたラウンドも記載される。

メッセージグループは、ファイナライゼーション関連のメッセージの塊であり、暗号学的な証明書の検証を可能にしている。すべてのファイナライゼーションメッセージは、署名（Signature）のみ異なっており、すべてのメッセージが連結され、その後に署名が続く形で収められている。最低でも 2 つのメッセージグループが含まれていて—ひとつは投票準備メッセージ、もうひとつは決定準備メッセージ—投票の内容によってはもっと増える可能性がある。

ファイナライズ証明書の検証には、エポックに関連する投票セットの情報が必要である。重要なのは、正式な投票者とその投票力が既知でなければならないことである。検証のために、証

明書は正式な投票者からの有効なメッセージのみを含んでいなければならない。さらには、個々のメッセージはすべて、証明書のヘッダーで指定される最新ブロックに至る $g(C_{r,v})$と、$g(V_{r,v}) \geq g(C_{r,v})$を満たす $g(V_{r,v})$ を示していなければならない。

15.7 シビル攻撃

決定論的ファイナライゼーションが設定でオンになっている時、投票者によるシビル攻撃は、通貨残高を反映する重み付けによって防がれる。攻撃者が、より多くの投票力を得るための唯一の方法は、より多くの通貨を集めることでしかない。アカウントの持つ残高を、複数のアカウントに分け与えたとしても、トータルの投票力は変化しないからである。

15.8 NOTHING AT STAKE 攻撃

決定論的ファイナライゼーションが有効化されている時、nothing at stake 攻撃もまた防ぐことができる。つまり、取引時に店主は、ブロックがファイナライズされるのを待ってから商品やサービスを提供すれば良いからである。もし店主がそれまで待てない場合でも、さらなる防衛力が期待される。

その最も効果的と思われるものは、攻撃者がより良い独自チェーンを構築するための時間が限られていることである。ブロックチェーンの巻き戻しのために、攻撃者はブロックがファイナライズされる前に独自チェーンを作らなければならない。さらには、ブロック難度を充分下げるためには、そのチェーンがファイナライズされるよりも多くの時間を要するだろう。

ファイナリティ導入によるメリットは、様々な攻撃を受けにくくするところにもあります。チェーンの巻き戻しは困難を極め、実用上の安全性は非常に高いものです。

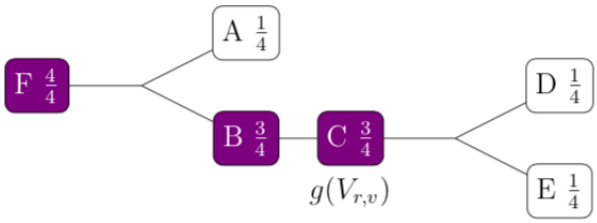
15.9 実例

ここから述べるすべての実例は、4 人の等価な投票人からなるネットワークを想定している。特別過半数は 4 人のうち 3 人が同じハッシュ値に合意することである（実際の Symbol では 2/3 以上）。F は常に最新のファイナライズされたブロックを指す。ブロック B に対する投票準備

メッセージは、F から B に至るすべてのハッシュを内包したハッシュチェーンを含んでいるとする。

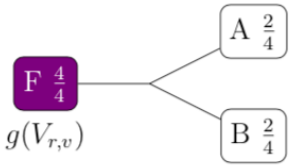
実例 1

投票準備メッセージによる投票が、ブロック A, C, D, E に（1/4 票ずつ）なされた場合。C は D と E に分岐しているため、C 直接には 1/4 票しか入っていないように見えても、（D と E に投票された分（それぞれ 1/4）を含んでいて）3/4 の特別過半数を満たしている（g(Vr,v)のところまでファイナライズ）。



実例 2

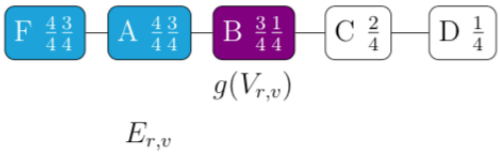
ブロックチェーンが、等価な 2 本に分岐している場合。投票準備メッセージは A, A, B, B になされたとする。F は A と B の分岐点に当たるため、F 直接への投票は無くても特別過半数を満たしている。F はすでにファイナライズされているため、この場合は、新しいブロックはファイナライズされない。



実例 3a

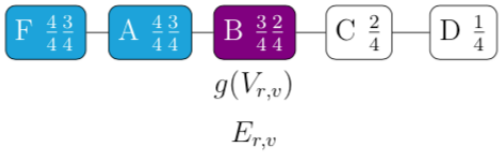
A, B, C, D のそれぞれについての投票準備メッセージが発行（4 人の投票人がそれぞれ、A, B, C, D に投票。投票内容は過去に遡って A, A, A, A, B, B, B, C, C, D になる。図の左側または単独の分数が投票割合をあらわす）され、B が特別過半数（3/4）を獲得している。２人の投票人は A, B, C への投票準備メッセージを見て、A の決定準備メッセージを送るとする（A, A, 図の右

側の数字が 2/4 になる）。（次に）1 人は、B, C, D への投票準備メッセージを見て、B（実際には A と B）の決定準備メッセージを送るとする。ここまでで A が 3/4、B が 1/4 の決定準備票（右側の分数）を得ている。最後の 1 人は、自分が持つ未知の決定準備投票（25%）が、その時点で 25%の決定準備票しか持たない B に与えられたとしても特別過半数を満たすことができないと判断するだろう。この場合の B は、15.5: アルゴリズムのケース 1（$E_{r,v} < g(V_{r,v})$）に該当する。



実例 3b

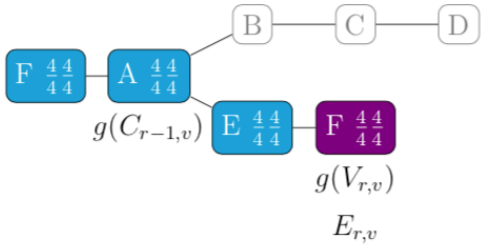
前の例に若干の変更を与えてみる。前回と同じ投票準備メッセージが異なる順番で受け取られた場合である。前と同様に、A, B, C, D のそれぞれについての投票準備メッセージが発行（投票内容は前回同様に A, A, A, A, B, B, B, C, C, D になる）され、B が特別過半数を獲得する。（前と異なり）2 人の投票人が B, C, D への投票準備メッセージを見て、B の決定準備メッセージを送るとする（A, A, B, B）。1 人は、A, B, C への投票準備メッセージを見て、A の決定準備メッセージを送るとする（A）。残りの 1 人は、未知の決定準備票（25%）が、その時点で 0%の投票準備しか持たない C の決定準備の特別過半数を満たすことができないと判断するだろう（この場合、B は 75%を獲得できる可能性がある）。この場合の B は、15.5: アルゴリズムのケース 2（$g(V_{r,v})$に含まれるどのブロックの組み合わせも $C_{r,v}$ の特別過半数を得ることができない場合）を満たす。



実例 4

先の例（3a と 3b）の、次の状態を考えてみる。A がラウンド r-1 で決定準備され、B は未定という状態である。ここで、投票者は次のラウンド r に進む。4 人の投票準備メッセージと決定

準備メッセージが F（最初の F ではなく、E の次のブロックの F）に送られるとする。その結果、ブロック B, C, D は、メインチェーンから取り除かれている。B は r-1 においては A のファイナライズ投票に使われたけれども（3b の場合？）、今回は使用されていない。r-1 時点での検証のために、A に続くブロックとして B の仮置きが必要だったからである。この B の情報は、投票準備メッセージにハッシュチェーンとしては記録されているが、決定準備メッセージからは省かれているはずである。これが、ファイナライゼーションの証明書作成に投票準備が必要な理由である。



実例を使っての説明です。図の中の分数（分子が得票数、分母が有効投票総数）は、2 つ並んでいる場合は、左側が投票準備、右側が決定準備での投票獲得率を示しています。ひとつだけのものは、投票準備の獲得率です。これまで説明されてきたアルゴリズムに従い、投票者は自分がどこまでを承認するか決めて投票するわけですが、自分が投票したブロックにつながるチェーン全体に投票することと同じであるところが重要です。例えば、実例 3 で B に投票するという事は、A と B に 1 票ずつ投票するということであり、A に投票するということは、A に 1 票投票するという感じです。3a と 3b の違いがどのようにして生じるかはよくわかりませんが、決定準備メッセージの送信の順番が微妙に前後することで、A までファイナライズ候補にするか B も含めるかが分かれそうです。このあたりのゆらぎが、実例 4 のようなマイクロフォークとなって、1, 2 ブロックの小さな巻き戻しを起こすのではないかと思います。

16 時刻同期

「時間も潮の満ち引きも、人を待ってはくれない」

- ジェフリー・チョーサー（イギリスの詩人）

他のほとんどのブロックチェーン同様に、Symbol はトランザクションやブロックに刻まれるタイムスタンプによって、時間的な連続性を保証する。理想的には、ネットワークにつながったすべてのノードが、同じ時計を持つべきである。現代のほとんどの OS は、時刻同期の仕組みを持っているが、1 分以上狂った時計を内蔵しているノードも少なくない。狂った時計を持ったノードは、有効なトランザクションやブロックを棄却してしまう可能性があり、ネットワーク全体との同期に失敗することにつながる。

それゆえに、ノードは他のすべてのノードと時刻を同期させる仕組みが必要であり、その実装には、主に 2 種類の方法がある：

1. Network Time Protocol (NTP)などの、既存の同期方法を使う（脚注；https://en.wikipedia.org/wiki/Network_Time_Protocol）。
2. 独自の時刻同期プロトコールを使う。

NTP など、既存のプロトコールを使うメリットは、実装が楽であり、ネットワークで共有される時刻が、常に実時間とも同期することである。しかし、ネットワーク外のサーバーに依存するという欠点がある。

カスタムプロトコールを利用すれば、P2P 通信によってこの問題を解決できるが、欠点もある。ネットワーク時刻が、実時間と同じであるという保証はできなくなる。いくつかの時刻同期プロトコールについては[Sci09]に述べられている。Symbol は、この文献の第 3 章にある方法を採用し、外部のリソースにはまったく依存しない方法を用いている。このプロトコールは、時刻同期（timesync）機能拡張が担っている。

16.1 サンプル収集

ネットワーク上の個々のノードは、オフセット（offset）という整数値を持ち、最初は 0 にセットされる。ノードの OS が持つシステム時刻を、オフセット値（負になることもある）で表されるミリ秒の分だけシフトしたものが、ノードのネットワーク時刻（network time，これもミリ秒単位）となる。

ノードが起動完了した後、ローカルノードはパートナーノードの選択に入り、時刻同期を始める。

ローカルノードは、選んだパートナーノードすべてに、現在のネットワーク時刻をリクエストする。ローカルノードは、リクエストを送付した時刻と、それに対する応答が返ってきた時刻を記憶する。それぞれのパートナーノードは、リクエストを受け取った時刻と、応答を送り出した時刻を、メッセージに入れて送り返す。

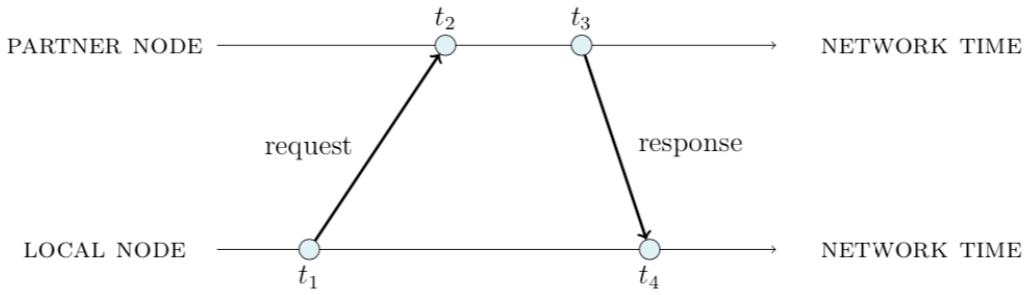


図 37: ローカルとパートナーノード間のコミュニケーション

パートナーノードは、自分が持つネットワーク時刻を使って、これらの送付用タイムスタンプを生成する。図 37 に、ノード間の通信の様子を示した。

これらのタイムスタンプを使って、ローカルノードは通信にかかった時間を計算する。

$$rtt = (t_4 - t_1) - (t_3 - t_2)$$

そして、2 つのノードが使っているネットワーク時刻間のオフセット（o）を計算する。

$$o = t_2 - t_1 - \frac{rtt}{2}$$

これをすべてのパートナーノードと同期がおこなわれるたびに繰り返して、ローカルノードはオフセット値のリストを作る。

インターネットの時刻同期プロトコールである NTP は、時々サーバーが落ちたりして使えなくなることもあります。NTP サーバーへの接続に失敗して、いつの間にか PC の時計が狂っていることはよくあることです。NIS1 や Symbol は、独自のネットワーク時計を持っていて、通信が生きている間は同期が崩れないところに特徴があります。オフセットは、２つのノード間の時刻のずれを反映します。

16.2 低品質な時刻データのフィルタリング

時刻データの品質を下げる要因はいくつか考えられる：

- 悪意あるノードによる不正確なタイムスタンプの送付。
- 悪意はないが、内蔵時計が大きく狂っているノードから送られてくる時刻データ。
- インターネットの問題によって、時刻データの往復時間が非対称になっているか、または、ノードの一方が非常に混雑してレスポンスが悪くなっている。これは、チャンネル非対称性と呼ばれ、避けるのは困難である。

低品質な時刻データを取り除くために、フィルターが用意されている。フィルタリングは 3 段階で実行される：

1. パートナーノードからのレスポンスが、決められた時間内に送られてこない場合（たとえば、$t_4 - t_1 > 1000ms$）、そのデータは破棄される。
2. 計算したオフセット値が制限時間内に収まっていない場合も、データは破棄される。許される制限時間の長さは、ノードの稼働時間とともに減少する。ノードが初めてネットワークに接続した時には、ネットワークで共有されている時刻と同期するために、大きなオフセット値が許される。しかし、時間が経過するにつれて、ノードが許されるオフセット値の範囲は狭められる。こうすることで、悪意あるノードが大きくずれたオフセット値を報告し続けるのを防ぐ。
3. これら以外の時刻データは、オフセット値順に並べられて、両端より α トリムフィルターがかけられる。言い換えると、両端からある程度の範囲までのデータが破棄される。

オフセット値に限界を定めることによって、ノードの時刻がネットワーク時間から大きくずれているノードのデータは破棄されます。ネットワークに接続した直後は、ある程度の範囲で受け入れることになってます。また、大きくずれた時刻データは、極端な値として他のノードによって棄却されるため、故意にネットワーク時刻をずらそうとする攻撃も成功しにくいようになっています。

16.3 有効オフセット値の計算

時刻同期時にノード間で交換されるオフセット値は、送信元ノードのブートアカウントが持つインポータンス値によって重み付けされる。パートナーとして選ばれるノードは、最低限度の量のインポータンス値を持たなければならない仕組みで、非常に低いインポータンスのノードは単純に考慮されない。これもまた、シビル攻撃を防ぐための仕組みである。

低インポータンス値しか持たないノードを大量に使って、有効範囲ギリギリのオフセット値を使い、ネットワーク時刻に影響を与えようとする攻撃者は、それらのノードを合計したインポータンス値を持った、たったひとつのノードと同じ程度にしか影響力を持つことができない。つまり大局的には、そのような多数のノードで攻撃したとしても、その影響力は、1 台の（高いインポータンスを持つ）ノードとみなしてよい。

また、サンプルとして用いたオフセット値の数や、パートナーノードの合計インポータンス値も、考慮される。これらの数値から求めたスケール係数がかけられたうえで、有効オフセット値が計算される。

I_j を j 番目のオフセット値 o_j を報告したノードのインポータンス値とする。n を、最新のインポータンス計算で有効だったノードの数、s をオフセット計算に使うデータの数とする。

その場合のスケール係数は、

$$scale = \min\left(\frac{1}{\sum_j I_j}, \frac{1}{\frac{s}{n}}\right)$$

となる。そして、この係数を使って実効オフセット値 o は、以下のように表せる。

$$o = scale \sum_j I_j \, o_j$$

ここで、大きなインポータンスを持つアカウントの影響力が制限されている点が重要である。つまりスケール係数を計算する際、$\frac{n}{s}$ が上限となる。そこで、大きなインポータンスを持つアカウントは、この制限がかからないようにするためにインポータンスを複数のアカウントに分割することで、全体として大きな影響力を持つことはできる。しかし、そのようにして増やしたノードのすべてが、他のノードによって時刻同期パートナーに選ばれるとは限らないし、その確率は高くはない。そのため、分割したノード全体の影響力の合計は、期待値よりも低くなる。

時刻同期にもインポータンスが関係していて、シビル攻撃を防いでいます。また、大きなインポータンスを持つアカウントが運用するノードが、必要以上に影響力を持ちすぎない仕組みも組み込まれています。大きなインポータンスを持つノードを使って、他のパートナーノードの時刻をずらそうとしても、n/s というインポータンスによって影響されない数値で影響力の上限が決められているためです。

16.4 カップリングと閾値

ネットワークに参加したばかりの新規ノードは、稼働中のネットワーク時刻に対するオフセットを、速やかに計算し同期をおこなわなければならない。逆に、ネットワーク内で長期間稼働しているノード（old nodes）は、悪意のあるノードや、新規ノードの接続による影響を受けないために、オフセット値を簡単には変化させないほうが良い。

どちらにしても、ノードが外部から取得した実効オフセットデータに、徐々に適応していくことになる。ノードは前項で計算した実効オフセット値にカップリング係数をかけたものを、最終のオフセット値として用いる。

個々のノードは、過去の時刻同期をおこなった回数を記録している。これをノード歴（node age）と呼ぶ。

カップリング係数の計算式は：

$$c = \max\left(\mathrm{e}^{-0.3a},\ 0.1\right) \text{ where } a = \max(nodeAge - 5,\ 0)$$

これにより、1 回目から 5 回目までの時刻同期におけるカップリング係数は 1 になり、それ以降は指数関数的に 0.1 まで減少する。

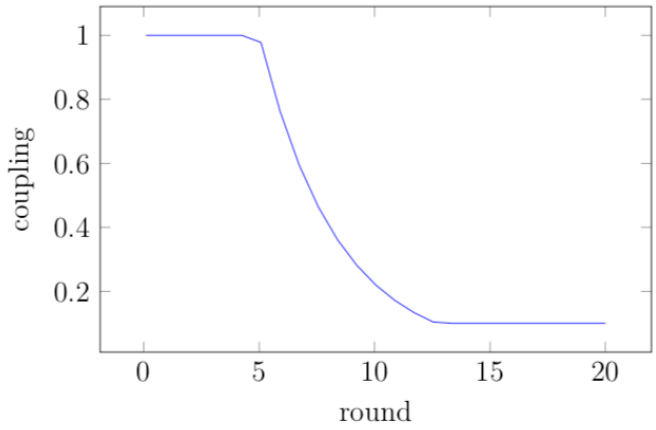


図 38: カップリング係数

最後に、ノードは計算した最終オフセット値を、内部で保持しているオフセット値と足し合わせる。ただし、オフセット値の変化の絶対値が、ある閾値（現在のところ 75ms）を超える場合のみである。閾値を設けて（感度を下げて）いるのには、ノード間のチャンネル非対称性によって、ネットワーク時刻がゆっくりと振動するのを防ぐ効果がある。

ネットワークに接続されたノードは、最初は素早く、時間が経つにつれゆっくりと、ネットワークで共有されているネットワーク時刻に適応していきます。時刻同期を 10 数回繰り返したノードは最小の 0.1 になり安定した old node とみなされるようです。

17 メッセージ

「良いメッセージは、伝えてくれる人に必ず出会えるの」

- アメリア・バー（イギリスの小説家）

ブロックチェーンのクライアントアプリケーション（ウォレットなど）は、常にブロックチェーンから情報を取り寄せて、ユーザーに提示する。優秀なクライアントアプリであれば、最新のブロックチェーンデータを表示して、情報に変化があれば直ちにそれを反映するだろう。普通のクライアントは、繰り返し REST サーバーに問い合わせをしたり、ローカルに作ったデータベースを利用しようとする。しかし、これらの方法はネットワークの帯域や、その他のリソースを無駄に消費するため、効率的とは言えない。そこで、Symbol では、単一のメッセージキューを使って、ブロックチェーンのデータ変更を自動的にクライアントに配信する仕組みを取り入れている。

17.1 メッセージチャンネルとトピック

クライアントが利用できる Symbol のメッセージキューは、複数のチャンネルを持っている。そして、それぞれのチャンネルは、独自のトピックに対応している。トピックは常にトピックマーカーと呼ばれる数値で始まり、どのような種類のトピックであるかがすぐに分かるようになっている。より選別を容易にするために、トピックマーカーの後ろにアドレスが続くものもある。クライアントの多くは、ブロックチェーンの状態変化のすべてを知りたいわけではない。そこで、クライアントは提供されるトピックのうちの一部にサブスクライブ（登録）すれば良いようになっている。図 39 は、全トピックマーカーのリストである。

Topic marker name	Topic marker
Block	0x9FF2D8E480CA6A49
Drop blocks	0x5C20D68AEE25B0B0
Finalized block	0x4D4832A031CE7954
Transaction	0x61
Unconfirmed transaction add	0x75
Unconfirmed transaction remove	0x72
Partial transaction add	0x70
Partial transaction remove	0x71
Transaction status	0x73
Cosignature	0x63

図 39：トピックマーカー

17.2 接続とサブスクリプション

メッセージ機能は、zeromq という機能拡張で提供される。ノードがメッセージ機能をサポートしたい場合は、この機能拡張をブローカープロセスの中で有効化する必要がある。この機能拡張は、ブロックやトランザクションに関するイベントに、サブスクライバーを登録し（2.2 : Symbol 機能拡張を参照）、メッセージキューマップを構築する。有効化された機能拡張を読み込んだブローカーは、messaging:subscriberPort を通して、新しいクライアントの登録要求を監視する。そして、ノードに接続したクライアントは、ひとつまたは複数のトピックについてのメッセージキューにサブスクライブすることができる。

古いプログラムだと、新しいトランザクションが無いかを定期的にノードに問い合わせる方法を取っています。毎日問い合わせ電話をかけて確認するようなものですね。しかし、ノードが新しいトランザクションを受け取った時に、クライアントにメッセージを送る仕組みが Symbol には組み込まれています。何かあった時だけ、向こうから電話がかかってくるような感じです。

17.3 ブロックメッセージ

ブロックメッセージの先頭にあるトピック欄に書き込まれるのは、トピックマーカーのみである。ブロックメッセージ内のレイアウトは図 40 に示した。下記の 2 種類のブロックメッセージを使うことができる：

- ブロック：新しいブロックがチェーンに付け加えられた。
- ドロップ：示したブロック高（height）以降のブロックが破棄された（巻き戻されて消えた）。
- ファイナライズブロック：ブロックがファイナライズされた。決定論的ファイナライゼーションが有効化された場合のみ送信される。

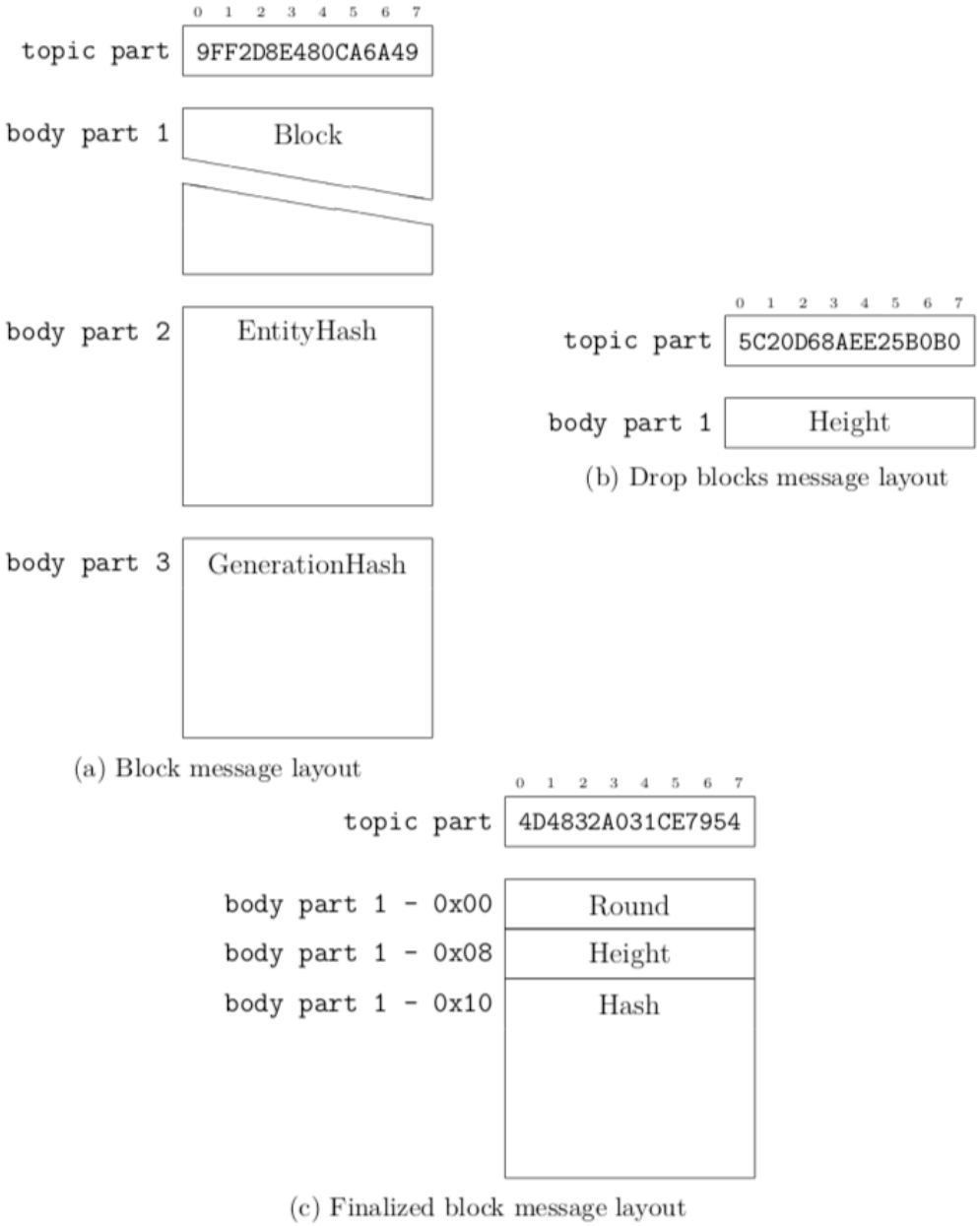


図 40：ブロックメッセージのレイアウト（a; ブロックメッセージ, b; ドロップブロックメッセージ, c; ファイナライズブロックメッセージ）

メッセージは、サブスクライブしたクライアントアプリに対して送られるため、ウォレットやエクスプローラーは、このメッセージを通じて新しいブロックの生成や、チェーンの巻き戻し、ファイナライズについての最新情報を得ることができます。ブロックメッセージの中にはブロック情報がまるごと入っているので、必要な情報はすべてメッセージに含まれていて、追加で問い合わせなくても良いようになっています。

17.4 トランザクションメッセージ

トランザクションメッセージの最初の部分には、トピックマーカーと、オプションで未解析アドレス（公開鍵とまだ結びついていないアドレスのことか？）フィルターが配置されている。未解析アドレスが提示されている時は、メッセージの本体には、そのアドレスに関連するものしか含まれていない。たとえば、送付トランザクションに関するメッセージのこの部分には、そのトランザクションの送り手か受け手のどちらかの未解析アドレスが提示されているはずである。ここに未解析アドレスが提示されていない場合は、メッセージはすべてのトランザクションに関連するものである。トランザクションメッセージのレイアウトについては、図 41, 図 42, 図 43 に示すとおり。下記のトランザクションメッセージが存在する。

- トランザクション：トランザクションが承認された。
- 未承認トランザクション追加：ひとつの未承認トランザクションが、未承認トランザクションキャッシュに追加された。
- 未承認トランザクション削除：ひとつの未承認トランザクションが、キャッシュから削除された。
- 未完成トランザクション追加：ひとつの未完成トランザクションが、未完成トランザクションキャッシュに追加された。
- 未完成トランザクション削除：ひとつの未完成トランザクションが、キャッシュから削除された。
- トランザクション状態（status）：トランザクション状態が別な状態に変化した場合の、その変化後の状態。

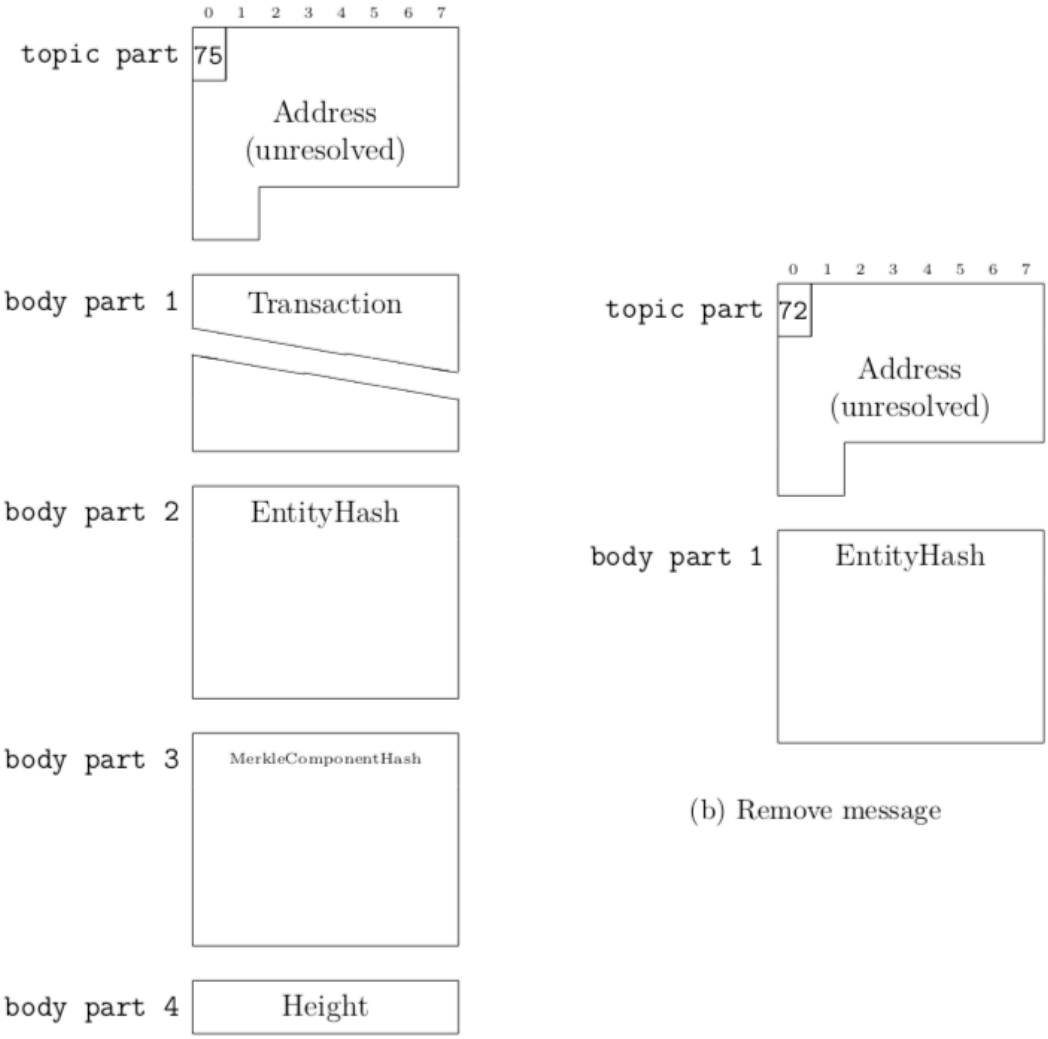


図 41：未承認トランザクション関連のメッセージレイアウト

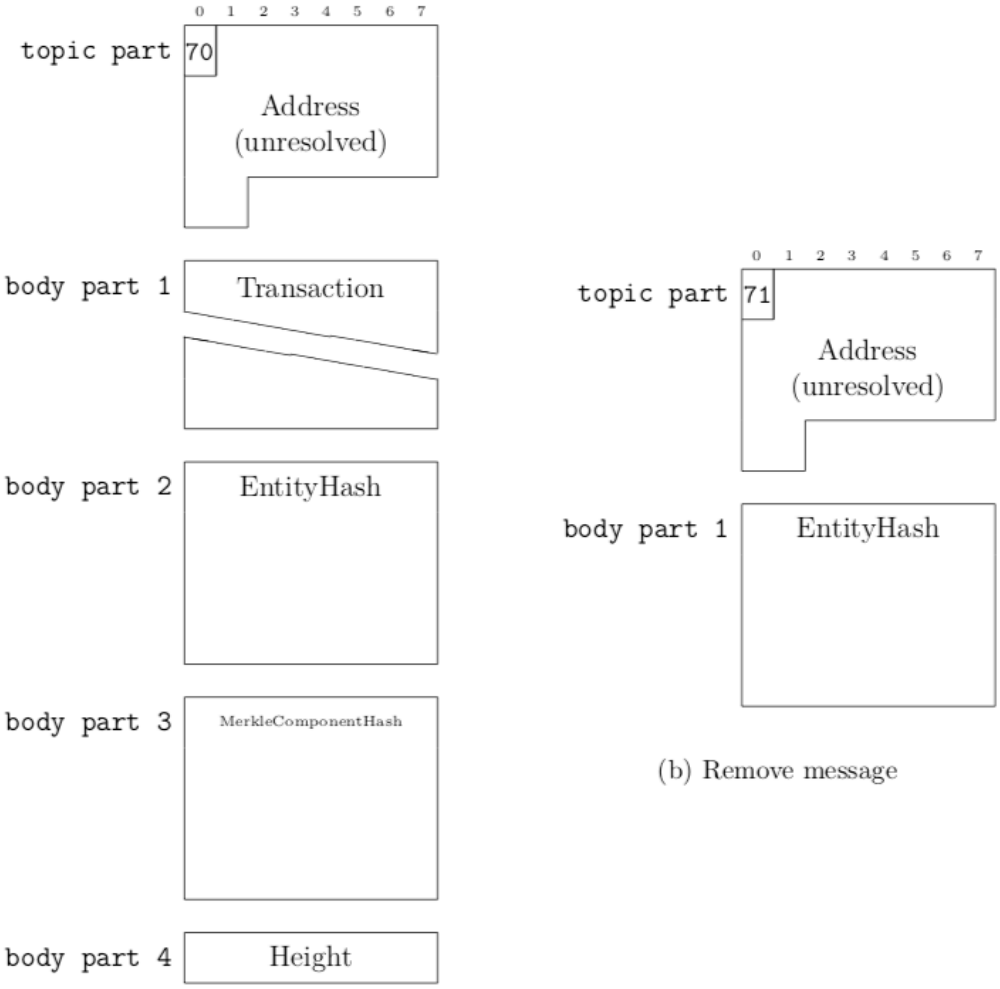


図 42：未完成トランザクション関連のメッセージレイアウト

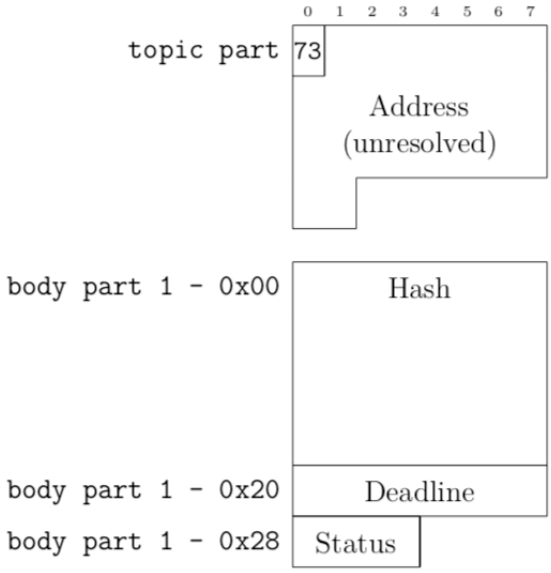


図 43：トランザクション状態（status）メッセージ

トランザクション関連のメッセージには、アドレスが無いものもあるようですが、図ではすべてアドレスが含まれていますね。アドレスは unresolved（未解析）とされています。「2.1 トランザクションプラグイン」では、エイリアス解決に resolve という言葉を当てています。図では、アドレスの長さが 25 バイトくらい（現在は 24 バイトに減少した）で表現されていますので、「5.1.2 アドレスエイリアス」で定義された、8 バイトのネームスペース ID を含む 24 バイトのアドレス、または、生のバイナリーアドレスが入りそうです。後者の場合は、unresolved は、まだ一度もトランザクションに使われておらず、公開鍵と結び付けられていない可能性のあるという意味にも取ることができます。アドレスが関連するトランザクション本体と、トランザクションハッシュ。そして、マークルハッシュが含まれています。最初にアドレスが来るので、それを見てウォレットなどは自分が関連するトランザクションかどうかを簡単に判断できるようになっています。

17.4.1 共同署名メッセージ

共同署名メッセージは、トピックマーカーと、オプションとして未解決アドレスフィルターが付けられる。このメッセージは、未完成トランザクションキャッシュに格納されているアグリゲートトランザクションに、新しい署名が追加された時、クライアントアプリに送られる。もし、未解決アドレスフィルターがあるなら、そのアドレスが、該当するアグリゲートトランザ

クションに含まれているということである。それが無い場合は、その他の変更に関するものである。共同署名メッセージのレイアウト構造を図 44 に示した。

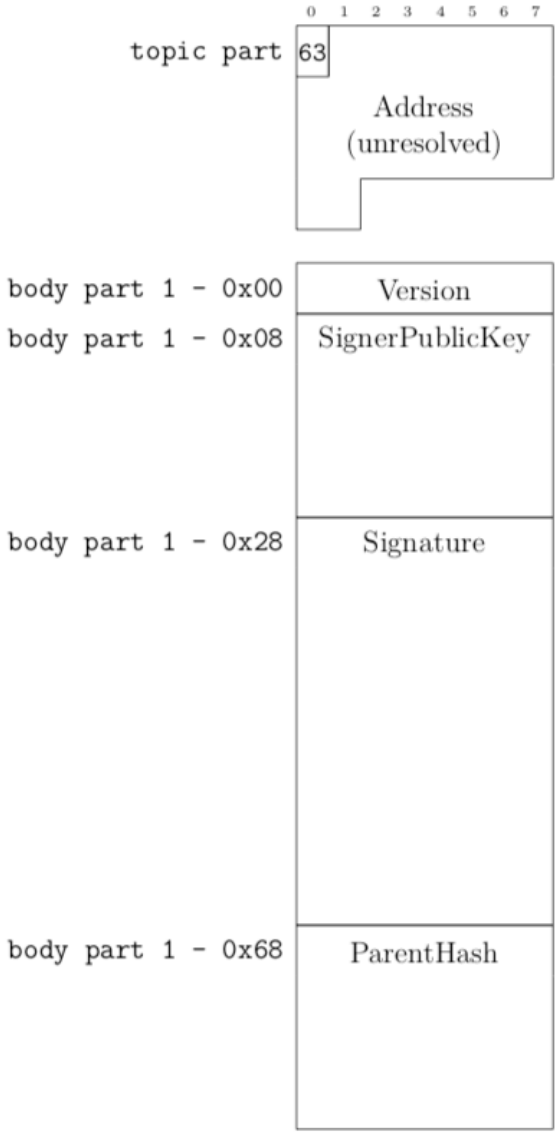


図 44：共同署名メッセージ

アグリゲートトランザクションに共同署名者が署名した時に発行されるメッセージについてです。やはり未解決アドレスの有無で 2 パターンあるとなっています。有りの場合は分かりますが、無しというのは、ちょっと分からないです。0.9.6.3 でバージョン情報が追加されました。

訳者あとがき

NEM ブロックチェーンは NIS1(NEM Infrastructure Server v.1)からスタートしました。PoW や PoS というスタンダードな合意形成アルゴリズムではなく、PoI という新しいアルゴリズムを採用した最初のブロックチェーンです。PoI はアカウントのネットワーク内重要度（インポータンス）を数値化するのに、全アカウントの接続性（グラフ）を計算するという、大胆な方法を採用しました。しかし残念なことに、PoI はスケールしない。つまり、参加するアカウントとトランザクション量が増えると計算量が増大して、ブロックチェーンの運用を圧迫するという致命的欠陥がありました。

この問題を解決し、劇的な高速化を図るためにカタパルトというプロジェクトが立ち上がりました。そして、カタパルトもまた不運な運命をたどります。時はブロックチェーンの氷河期です。バブル市場の縮小にともなって、未熟なブロックチェーンが次々と倒れ、消えていきました。しかし、その市場評価額を大きく下げながらも、NEM を支えるコミュニティーと、テックビューロの庇護の元、カタパルトはかろうじて生き延びます。他のブロックチェーンが軒並み階層性を深めて、複雑で難解になっていくのを横目に見ながら、カタパルトはスクラップアンドビルドによってブロックチェーンを高速化しつつ、スマートコントラクトに匹敵する複雑なタスクをこなす新しいタイプのブロックチェーンとして育まれます。

やがて、カタパルトはその射出台という名前の通り、その役割を終えます。Symbol の誕生です。Symbol は、ブロックチェーンの最大の弱点であるファイナリティ問題をも解決しました。常に進化し続けること。それが NEM = New Economy Movement の本質だと、主張するかのようにです。

NEM ブロックチェーンと出会えたことは、初学者であった私にとっての僥倖でした。オリジナルであることにこだわらず、技術を実用性というものさしで評価し、有用なものは積極的に取り込んでいく。つまり、NEM に取り込まれている技術を学べば、ブロックチェーンの核心が見えてきます。本稿が、みなさんのブロックチェーンについての知識を深め、いろいろな技術やサービスを生み出していく一助になることを願っています。

参考文献

[BCN13] BCNext. *Whitepaper:Nxt*. Web document. 2013. url: https://nxtwiki.org/wiki/Whitepaper:Nxt#Blocks.

[Ber+11] Daniel J. Bernstein et al. “High-Speed High-Security Signatures”. In: *Crypto-graphic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*. 2011, pp. 124–142. doi: 10.1007/978-3-642-23951-9_9. url: http://dx.doi. org/10.1007/978-3-642-23951-9_9.

[BG17] Vitalik Buterin and Virgil Griffith. *Casper the Friendly Finality Gadget*. 2017. eprint: 1710.09437. url: http://arxiv.org/abs/1710.09437.

[BM99] Mihir Bellare and Sara K. Miner. “A Forward-Secure Digital Signature Scheme”. In: *Advances in Cryptology - CRYPTO ’99, 19th Annual International Cryp- tology Conference, Santa Barbara, California, USA, August 15-19, 1999, Pro- ceedings*. Springer, 1999, pp. 431–448. doi: 10.1007/3-540-48405-1_28.

[CL99] Miguel Castro and Barbara Liskov. “Practical Byzantine Fault Tolerance”. In: *Proceedings of the Third Symposium on Operating Systems Design and Implementation*. USENIX Association, 1999, pp. 173–186. isbn: 1880446391. doi: 10.5555/296806.296824.

[Gol+20] Sharon Goldberg et al. *Verifiable Random Functions (VRFs)*. Internet-Draft draft-irtf-cfrg-vrf-07. Work in Progress. Internet Engineering Task Force, June 2020. 39 pp. url: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vrf-07.

[KN12] Sunny King and Scott Nadal. *PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake*. Web document. Aug. 2012. url: https://decred.org/research/king2012.pdf.

[Mer88] R. C. Merkle. “A Digital Signature Based on a Conventional Encryption Function”. In: *Advances in Cryptology — CRYPTO ’87*. 1988, pp. 369–378.

[Mor68] Donald R. Morrison. “PATRICIA - Practical Algorithm to Retrieve Information Coded in Alphanumeric”. In: *Journal of the ACM, 15(4)*. 1968, pp. 514–534.

[Nak09] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. 2009. url: http://www.bitcoin.org/bitcoin.pdf.

[Sci09] Sirio Scipioni. “Algorithms and Services for Peer-to-Peer Internal Clock Syn- chronization”. PhD thesis. Universit‘a degli Studi di Roma „La Sapienza”, 2009.

[SK20] Alistair Stewart and Eleftherios Kokoris-Kogia. *GRANDPA: a Byzantine Finality Gadget*. July 2020. arXiv: 2007.01560 [cs.DC].

お詫び：索引は訳語の統一が十分にできておらず、今回は作成できませんでした。PDF ビューワーの検索機能等を利用してくださると助かります。