Tezos — 自己修正暗号台帳
ホワイトペーパー
日本語訳 *

L.M Goodman

2014 年 9 月 2 日

元の論文と現在の実装との間の変更は赤で示されている。訳註: この和訳は最終変更が *2018* 年 *5* 月 *20* 日のドキュメントに基づいているため、実際の実装はここからさらに変更されている可能性がある。

> “我々の議論は単に堂々巡りをしているわけではないが、まあ、そのようなものだ。”
>
> — Willard van Orman Quine

概要

我々は汎用的で自己修正可能な暗号台帳 Tezos を提案する。Tezos はブロックチェーンに基づくどんな台帳方式も選択し、採用することができる。通常のブロックチェーン操作は純粋関数型モジュールとして実装され、ネットワーク操作を受け持つ殻 (シェル) に挿入される。ネットワーク層への適切なインターフェースを実装すれば、Bitcoin、Ethereu、Cryptonote など、全てが Tezos 内で表現できる。

最も重要な Tezos の特徴はメタ更新、つまり、プロトコルがそれ自身のコードを更新して進化することが可能である事だ。この実現のため、Tezos はステークホルダー (通貨保持者) によるプロトコル修正承認の手続き、これには**この投票手続き自体の更新も含まれる**、が定義された開始プロトコル[*1]を使ってスタートする。これは Peter Suber の Nomic[3] という、完全に自らのルールを変更可能なゲームに似ていなくもない。

この論文では、読者に Bitcoin プロトコルと基本的な暗号プリミティブの知識があることを仮定する。

---

* この文書の最新版は `https://gitlab.com/dailambda/tezos-papers/blob/ja/pdfs/position_paper_ja.pdf` にある。翻訳内容に関する問い合わせについては訳者の古瀬 淳 (`jun.furuse@tezos.or.jp`) まで。

[*1] 訳註: 原文は”seed protocol”。一般的ではないので、開始、とした。

目次

1　はじめに

この論文の前半では、抽象ブロックチェーンのコンセプトと自己修正暗号台帳の実装を議論する。後半では、Tezos の開始プロトコルについて解説する。

2　自己修正暗号台帳

ブロックチェーンのプロトコルは次の三つの独立したプロトコル群に分割することができる:

- ネットワーク・プロトコル: ブロックを発見し、トランザクションのブロードキャストを行う。
- トランザクション・プロトコル: トランザクションの検証条件を定義する。
- コンセンサス・プロトコル: 唯一の正当なチェーンを決定するための合意形成を行う。

Tezos は汎用的なネットワークシェル (殻) を持つ。このシェルはトランザクション・プロトコルとコンセンサス・プロトコルに対し透過的である。トランザクション・プロトコルとコンセンサス・プロトコルをまとめて、「ブロックチェーン・プロトコル」と呼ぶこととする。我々はまずブロックチェーン・プロトコルの数学的な表現を与え、それから Tezos での実装におけるデザインチョイスについて解説する。

2.1　ブロックチェーンの数学的表現

ブロックチェーンプロトコルは根本的にはグローバル状態に対する並行書換系のモナディックな実装であり、これはグローバル状態へ作用する演算子として「ブロック」を定義することで実現できる。ブロック集合の自由モノイドを初期 (genesis) 状態に対して作用させると、ツリー構造が得られる。与えられた順序における最大[*2]の葉をグローバルな正準状態と定義する。

これにより次のような抽象表現が得られる:

- $(\mathbf{S}, \leq)$ を可能な状態の全順序付き加算集合とする。
- $\oslash \notin \mathbf{S}$ を特殊な無効状態とする。
- $\mathbf{B} \subset \mathbf{S}^{\mathbf{S} \cup \{\oslash\}}$ をブロックの集合とする。有効 *(valid)* なブロックの集合を $\mathbf{B} \cap \mathbf{S}^{\mathbf{S}}$ とする。

$\mathbf{S}$ 上の全順序を $\forall s \in \mathbf{S}, \oslash < s$ となるよう拡張する。この順序はブロックツリーのどの葉が最も正準なものかを決定する。$\mathbf{B}$ のブロックは状態に作用する演算子として見なすことができる。

だいたい、どんなブロックチェーン・プロトコル[*3](Bitcoin, Litecoin, Peercoin, Ethereum, Cryptonote など) も次の組で完全に決定できる:

$$\left(\mathbf{S}, \leq, \oslash, \mathbf{B} \subset \mathbf{S}^{\mathbf{S} \cup \{\oslash\}}\right)$$

これらのブロックチェーンでは、ネットワーク・プロトコルは根本的に同一である。「マイニング」アルゴリズムは、与えられたブロック生成インセンティブから導き出されるネットワークの性質そのものである。

[*2] 訳註: 原文では minimum とあるが、続く S の順序の定義 $\forall s \in \mathbf{S}, \oslash < s$ から、もちろん間違いである。

[*3] GHOST はツリーの性質によって葉を順序づける (訳註:そのため、この範囲外となる) が、そのようなアプローチは理論的にも実用的にも問題がある。このような手法を取らず、主チェーンにマイニング証明を挿入することでエミュレートする方がほとんどの場合良い結果が得られる。

Tezos ではブロックがプロトコル自身に作用できるようにすることで、ブロックチェーン・プロトコルを自己参照的にしている。そのため、プロトコルの集合 P は再帰的に次のように表現される:

$$\mathcal{P} = \left\{ \left(\mathbf{S}, \leq, \varnothing, \mathbf{B} \subset \mathbf{S}^{(\mathbf{S} \times \mathcal{P}) \cup \{\varnothing\}} \right) \right\}$$

2.2 ネットワーク・シェル

上述の形式的数学的記述は**どのように**ブロックツリーを構成するかは説明しない。この役割を担うのがネットワーク・シェルだ。シェルはゴシップ・ネットワークとプロトコルとの間のインターフェースとして振る舞う。

ネットワーク・シェルはクライアントに知られている最も良いチェーンを維持するのが仕事だ。シェルは三種のオブジェクトを取り扱う。最初の二つはトランザクションとブロックであり、これらは検証された場合のみネットワークに伝達される。三つ目はプロトコルで、これは既存のプロトコルを改訂するために使われる OCaml のモジュールだ。これについては、後で詳しく述べる。ここではトランザクションとブロックについて主に説明する。

ネットワーク・シェルの最も難しい部分は、DoS 攻撃からノードを保護することだ。

2.2.1 クロック

各ブロックにはタイムスタンプがついており、ネットワーク・シェルからアクセスできる。現在より先のタイムスタンプのついているブロックは、その差がシステム時間から数分以内であればバッファリングされるが、それ以外の場合は破棄される。プロトコルのデザインはクライアント側のクロックのずれをある程度許容すると同時に、タイムスタンプが改竄される可能性があることも仮定しなければならない。

2.2.2 チェーン選択アルゴリズム

シェルは、ブロックのツリー全体ではなく、単一のチェーンだけを取り扱う。このチェーンは、クライアントが厳密により良いチェーンの存在を発見した場合にのみ置き換えられる。

ツリー全体を管理するのはネットワーク通信の観点からはより簡潔になるかもしれないが、攻撃者がスコアは低いが有効なフォークを大量に生成した場合、そのサービス拒否攻撃の影響を受けやすくなる。

それでも、あるノードがチェーンのスコアについて嘘をつくことは可能であり、その嘘はクライアントが多数のブロックを処理して初めて発見できるものである可能性はある。しかし、そのような嘘をつくノードは、それ以降無視することができる。

幸い、プロトコルは低い得点のチェーンは低いブロック作成率を持つという性質を持つことができる。したがって、クライアントは、公表されたスコアが嘘であると結論するためには、「弱い」フォークの少数のブロックについて考慮するだけすむだろう。

2.2.3 ネットワークレベルの防御

さらに、シェルは「防御的」である。シェルは様々な IP レンジに渡って多数のピアに接続しようと試みる。また、接続の切れたピアを検知し、悪意のあるノードを除外する。

サービス拒否攻撃から守るため、プロトコルは、ブロックとトランザクションのコンテクストに依存のサイズ上限をシェルに通知する。

2.3 関数型表現

2.3.1 チェーンの検証

以下の OCaml 型を使用することで、抽象ブロックチェーン構造のほぼ全ての一般性を効率的に実装することができる。まず、ブロックヘッダーは次のように定義される:

```
type raw_block_header = {
  pred: Block_hash.t;
  header: Bytes.t;
  operations: Operation_hash.t list;
  timestamp: float;
}
```

ここで、ヘッダー要素は、どのような内容も表現できるよう、あえてこれ以上強く型付けしない。しかしシェルの動作に必要なフィールド、例えば前ブロックのハッシュ、オペレーションのハッシュとタイムスタンプのリストなど、については型を与える。実際には、ブロックに取り込まれることになるオペレーションはネットワークレベルでブロックと一緒に伝達される。オペレーションそのものは (訳註:このレベルでは) 任意のバイナリブロブとして表現される:

```
type raw_operation = Bytes.t
```

状態は、**Context** モジュールを使って表現され、ディスク上の不変キーバリューストアとしてカプセル化される。キーバリューストアの構造は多目的に使え、広い範囲に渡る状態を効率よく表現することができるようになっている:

```
module Context = sig
   type t
   type key = string list

   val get: t -> key -> Bytes.t option Lwt.t
   val set: t -> key -> Bytes.t -> t Lwt.t
   val del: t -> key -> t Lwt.t
   (*...*)
end
```

ディスク操作遅延によるブロッキングを避けるために、関数群は非同期モナドライブラリである Lwt[4] を使用する。コンテクスト上の操作は副作用のない純粋関数型になっている: 例えば、**get** は (訳註: 値が見つからなかった時に) 例外を発生するのではなく、**option** モナドを返す。また、**set** と **del** は (訳註: 既存 Context を破壊的更新するのではなく、) 新しい **Context** を生成する。**Context** モジュールはメモリキャッシュとディスクストレージを組み合わせた効率の良い不変ストアを提供する。

任意のブロックチェーン・プロトコルのモジュール型は次のように定義できる:

```
type score = Bytes.t list
module type PROTOCOL = sig
   type operation
   val parse_block_header : raw_block_header -> block_header option
   val parse_operation :  Bytes.t -> operation option

   val apply :
     Context.t ->
     block_header option ->
     (Operation_hash.t * operation) list ->
```

```
    Context.t option Lwt.t

  val score : Context.t -> score Lwt.t
  (*...*)
end
```

ここでは、数学的モデルのように状態を直接比較するのではなく、**score** 関数を使って **Context** からスコアのバイト列を求める。バイト列には、最初に長さ、次に辞書順の順序が入っている。これは、ソフトウェアのバージョン管理で使用されているものに類似した非常に一般的な構造で、様々な順序を表現するのに非常に汎用性がある。

なぜプロトコルモジュールの中に状態の比較関数を定義しないのか? まず、そのような比較関数が全順序を満たすことを強制することが難しいからである。スコアへの写像であれば、この全順序性は常に満たされる(最後のブロックのハッシュを使えば複数の異なるコンテクストが同じスコアを持つことを避けることができる)。また、原則として我々は異なるプロトコル間の状態を比較する能力が必要だ。特定プロトコル修正ルールによって、このような比較が発生することは極力抑えられるはずだが、それでもなお、ネットワーク・シェルはこの性質を仮定できない[*4]からである。

parse_block_header と **parse_operation** 関数はシェルに公開され、プロトコル層へと完全に型付けされたオペレーションとブロックを渡すために使われる。また、これらの関数はオペレーションを伝達したり、ブロックをローカルブロックツリーデータベースに加えることを決定する前に、オペレーションやブロックが正しい形であることもチェックする。

apply 関数はプロトコルの中心である:

- ブロックヘッダとオペレーションのリストを渡された場合、コンテクストへの変更を計算し、変更を適用したコンテクストのコピーを返す。内部では、バージョン管理システムと同様、バージョンハンドルとして前ブロックのハッシュを使って、そこからの差異だけが保存される。
- オペレーションのリストだけが渡された場合 (訳註: ブロックヘッダが省略された場合)、できるだけ多くのオペレーションを適用しようとする。この機能はプロトコル自体には必要ないが、マイナーが正当なブロックを構成しようとする時に非常に便利である[*5]。

2.3.2 プロトコル改訂

Tezos の最も強力な機能は、自己修正可能なプロトコルを実装できることだ。これは、次の 2 つの手続き関数をプロトコルに公開することによって実現される:

- **set_test_protocol** 関数はテストネット環境で使われているプロトコルを新しい物 (通常、ステークホルダーの投票によって採用された物) に入れ替える。
- **promote_test_protocol** 関数は現在のプロトコルをテスト中のプロトコルに入れ替える。

これらの関数は関連するプロトコルを変更することでコンテクストを変更する。新しいプロトコルは、チェーンに次のブロックが適用される時に有効になる。

```
module Context = sig
  type t
```

[*4] 訳註: そのため、やはり異なるプロトコルを使った状態間の比較が可能である必要がある。

[*5] 訳註: Tezos ソースコードの docs/whitedoc/validation.rst の”Prevalidator” の節にもう少し説明がある。

```
  (*...*)
  val set_test_protocol: t -> Protocol_hash.t Lwt.t
  val promote_test_protocol: t -> Protocol_hash.t -> t Lwt.t
end
```

protocol_hash は、プロトコルを実装する**.ml** と**.mli** ソースファイルの tar アーカイブの **sha256** ハッシュだ。これらのファイルは各ノード上でコンパイルされる。プロトコルコードは小さい標準ライブラリにアクセスできるが、それはシステムコールを呼び出すことができないようサンドボックス化されている。

これらの関数はプロトコルの新しい **Context** を返す **apply** 関数を通して呼び出される。

プロトコルの更新は多くの条件によって引き起こされる。最も単純な場合、ステークホルダーの投票がプロトコル更新を引き起こすが、時が経つにつれ、より複雑なルールが徐々に投票により組み込まれていくだろう。例えば、もしステークホルダーが、それ以降の改訂案には一定の性質を満たしていることを機械で確認できる証明を添付しなければならない、というプロトコル改訂を通そうと思った場合だ。これは実質的に合憲性のアルゴリズム的チェックを導入する事となる。

2.3.3 RPC

GUI 構築の複雑性を軽減するため、プロトコルには JSON-RPC API が備わっている。API 自体はさまざまなプロシージャの型を示す JSON スキーマで記述される。典型的には、**get_balance** のような関数はこの RPC で実装される。

```
type service = {
  name : string list ;
  input : json_schema option ;
  output : json_schema option ;
  implementation : Context.t -> json -> json option Lwt.t
}
```

RPC の名前は文字列のリストになっていて、これによりプロシージャの名前空間を構成する。その入力と出力の型は JSON スキーマによって記述することができる。

RPC はあるコンテクスト、典型的には最も高いスコアを持つ葉の最近の親、に対して呼び出される。例えば、最も高いスコアを持つ葉の 6 代前のブロックのコンテクストを照会すると、6 回の確認作業が行われた台帳の状態を得ることになる。

UI 自体はある特定のプロトコルに対して特別に作成することもできるし、より汎用的に JSON 仕様から自動生成することもできるだろう。

3 開始プロトコル

ブロックチェーンが genesis ハッシュから開始されるように、Tezos は開始プロトコルからスタートする。このプロトコルは、プロトコル改訂により、ほぼいかなるブロックチェーンベースのアルゴリズムへにも変化することができる。

3.1　コイン経済

3.1.1　コイン

開始時には 10 000 000 000 (100 億) コインが存在し (初期のトークン供給度合いはクラウドファンディングで発行したトークン数となる。必ずしも「10 億」とはならない。これは単に仮の値であって、このサイズの変更は主要な性質に影響を与えない。)、それらは小数点以下二桁まで分割することができる (精度向上のため、実際には少数点 8 桁までの分割を許すことになるだろう)。我々はこの 1 コインを「tez」、最小単位を単にセントとと呼ぶことにする。また tez の通貨記号として ꜩ (\ua729, “Latin small letter tz”) を使うことにする。つまり、1 cent = ꜩ0.01 = 1/100 tez。

3.1.2　マイニングと署名報酬

■**原則**　いかなる非中央集権通貨においても、その安全性には参加者への金銭的報酬の動機付けを与えることが必要であると予想されている (我々は現在報酬スケージュールの確定作業中である)。トランザクションコストのみに依存するのは、ポジションペーパーに説明されているように、共有地の悲劇をまねく。Tezos では、担保と報酬の組み合わせを使用する。

担保はマイナーによって払い込まれる一年間 (その高い機会コストと 1 サイクル以上の拘束が安全上の利益がないことから、担保期間は 1 サイクル (約三日) に短縮された) の安全保証金となる。(裏書きをする署名者も担保を購入する必要がある。) 二重署名をしてしまった場合、この担保は没収される。

1 年後 (1 サイクル後)、マイナーたち (および裏書人) は預けた担保とともに機会コストを補う報酬を受け取る。システムの安全性は主に担保の価値によってもたらされる。報酬はその価値の数 % ほどである。

担保の目的は必要報酬額を減らすことだ。また、おそらくその損失回避効果はネットワークを利することとなるだろう。

■**詳細**　開始プロトコルでは、1 ブロックあたり ꜩ512 のマイニング報酬を提供し、それには ꜩ1536 の担保が必要となる。ブロックへの署名は $32\Delta T^{-1}$ tez の報酬を提供する。ここで ΔT は署名されるブロックと前ブロックとの時間差である。各ブロックには 16 名までの署名者がおり、署名には担保を必要としない。これらの数字は 10 億トークンの供給に基づいたもので、実際の供給量に沿ったものとなるだろう。シミュレーションにより、ブロックごとの署名数が増えるとフォークが非常に難かしくなるとわかったので、その数を増やすことがあるかもしれない。

したがって、毎分 1 ブロックのマイニングレートを仮定すると、一年後には初期通貨供給量の約 8% が安全担保として保管されることになる上述のパラメータ変更によりこの率は変化する。

この報酬スケジュールに従うと、名目インフレーション率は年間最大で 5.4% となる (全ブロック報酬は今でも年間で約 5% で始まるが、漸近的な総トークン数の上限を導入することができる。ガバナンスモデルがトークン保持者の利益と一致しているときには、総トークン数上限の導入は的外れだと我々は考えるが、ある人々には重要であるようなので、仕方なく考慮している)。名目インフレーションは中立的で、誰もそれで得をしたり、損をしたりということはない [*6]。

一年の期間はブロックのタイムスタンプによって決められるもので、ブロック数によってではないことに注

[*6] 対照的に、Bitcoin のマイニングインフレーションは全体として Bitcoin 保持者に損をさせ、中央銀行は預金者を犠牲にすることで金融業界に得をさせている。

意。マイナーのコミットメント (訳註:担保保証金など) の長さに関する不確実性を取り除くためのものである。

■展望　提案する報酬はマイナーに担保の 33% のリターンをもたらす (現在このパラメータを調整中でもうすぐ全ての参加者にとって意義のあるものとなるだろう)。このリターンは Tezos 開始初期には、マイナーや署名者たちが一年間もの間、ボラティリティーが高い可能性がある資産をロックするというリスクにあわせて、高くある必要がある (担保の保持期間は丸一年ではなく 1 サイクルのみに変更された)。

しかしながら、Tezos が成熟するにつれ、このリターンは一般の金利レートにまで次第に落ちていくだろう。名目インフレーションレートは 1% を切ることは安全に可能だが、あえてやる利点があるかどうかは不明である。

3.1.3　失われたコイン

マネタリーベースに関わる不確実性を減らすために、(タイムスタンプにより) 一年以上活動のないと判断されたアドレスは保持するコインごと破壊される (ホワイトペーパーにおける当初の提案とは異なり、活動のないアドレスは一年以上たっても資金を失うことはない。これらのアカウントは再び活動が認められるまでステーク権利を失うだけである。これはつまり、アドレスが非活動状態の時には、ブロック生成者として選ばれない (さもなければ合意アルゴリズムが遅くなってしまう)、また、再び活動するまでは投票も許されない (投票率の不確実性を避けるため)。。

3.1.4　修正ルール

プロトコル修正は $N = 2^{17} = 131\,072$ ブロックの選挙サイクル上で採用される。各ブロック間隔が 1 分間であるとして、これはカレンダー上では約 3 ヶ月である。この選挙サイクルは各々 $2^{15} = 32\,768$ ブロックからなる 4 つの期間に分けられる。このサイクルは初期の改善を奨励するため比較的短く設定されているが、将来のプロトコル改訂によりこのサイクルはより長くなるだろうと期待されている (最初の一年間は、プロトコルアップグレードを早急に繰り返すため、投票はもっとより頻繁に行われるであろう。システム安全性のため、投票手続きシステムに問題ないことを確認するまで、Tezos 財団は初めの 12 ヶ月間は拒否権を持つ。) 修正の採択には一定の議決定足数が必要である。この定足数は $Q = 80\%$ から始まるが、平均参加率に応じて動的に変更される。これは失われたコインに対処するためだけに必要である。

■第一期　プロトコル改訂は、新しいプロトコルを実装する `.ml` と `.mli` ファイルの tarball のハッシュという形で提案される。ステークホルダーはこれらの新プロトコル提案を好きなだけ承認することができる。これは「承認投票」として知られ、特に堅牢な投票手続きである。

■第二期　第一期でもっとも信任を集めた改訂案が投票にかけられる。ステークホルダーは賛成、反対、そして明示的な棄権のどれかに投票できる。この明示的な棄権票は定足数に関わる投票として数えられる。

■第三期　もし (明示的棄権票も入れて) 議決定足数が満たされ、そのうち (訳註: 明治的棄権票を除き)80% が賛成票の場合は、改訂案は可決されテストプロトコルとして採用される。そうでない場合は否決される。投票率が q だった場合、次回の議決定足率 Q は次のように更新される:

$$Q \leftarrow 0.8Q + 0.2q$$

この定足率の更新は、時間が経つにつれて失われたコインによって投票手続きが止まってしまうのを防ぐた

めにある。最低定足率は各前投票の定足率の指数平滑移動平均となる*7。

■第四期　改訂が (第二期で) 承認された場合、それは第三期のはじめからテストネットで動作している。ステークホルダーは第二回目の投票を行い、テストプロトコルからメインプロトコルへの移行を行うか、否か決定する。この投票にも上述の定足率と 80% の多数決ルールが適用される。

我々は意図的に保守的な改訂アプローチを選択した。しかし、ステークホルダーは自分の利益に沿うようこの改訂ルールを緩くしたり厳しくしたりすることが (訳註: 改訂ルールそのものの変更案を提案、投票することで) 可能だ。

3.2　Proof-of-stake メカニズム

3.2.1　概観

我々の proof-of-stake メカニズムは Slasher[1]、chain-of-activity[2]、proof-of-burn などいくつかのアイデアを取り入れたものだ。次にアルゴリズムの簡単な概観を説明し、その構成要素については後で詳しく解説する。

各ブロックは、ランダムなステークホルダー (マイナー) によってマイニングされ、ランダムに選ばれたステークホルダー (署名者) による前ブロックの複数の署名を含んでいる。マイニングと署名はどちらも小額の報酬を提供するが、同時に 1 年間 (保証金の変換は、はじめに提案された 1 年ではなく、1 サイクル後に行われる。1 サイクル以上に期間を伸ばし、多くの資本を移動不可にするコストに見合う分の、安全性の改善は実際にはなかったため。) の安全保証金を必要とする。この保証金は二重マイニングや二重署名が発見された場合は没収される。

プロトコルは各 2048 ブロックからなるサイクルに分解される。各サイクルの始めに、最後から 2 番目のサイクル中で選択されたブロックマイナー達が選んで投稿し、最後のサイクルで公開された数字群からランダムシードが導出される。このランダムシードを使用して、follow-the-coin 戦略を使って次サイクルのマイニング権と署名権のアドレスに割り当てを行う。図 1 を参照。

*7 訳註: 現在もこの定足数変化が採用されているが、アクティブでないアカウントに関する投票権の取り扱いが変わったため、この説明は現在は成り立たない

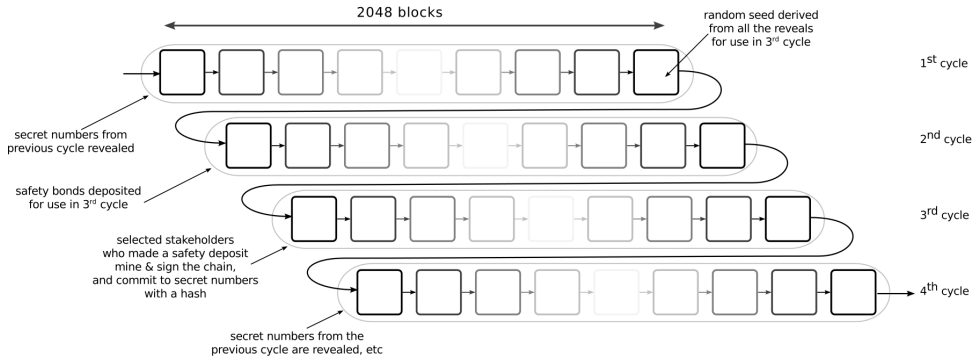

図 1　Four cycles of the proof-of-stake mechanism

3.2.2 クロック

プロトコルは、ブロック間に最小の時間間隔を設けている。原則として、各ブロックはステークホルダーならば誰でもマイニングすることができる。しかしながら、あるブロックに対して、各ステークホルダーはそれぞれランダムな最小遅延を受ける: 最優先権を得たステークホルダーは、前のブロックの 1 分後に次のブロックをマイニングすることができる。2 番目に高い優先度を得たステークホルダーは、前のブロックの 2 分後に次ブロックをマイニングすることができ、3 番目は 3 分後、と続く。

これにより、ステークホルダーの少数だけが関わるフォークはブロック生成レートが低くなることが保証される。もしそうでなければ、非常に高いスコアを持っていると主張するとても長いチェーンを検証させることで、CPU サービス拒否攻撃を行いノードを邪魔することが可能となってしまうだろう。

3.2.3 乱数シードの生成

マイニングされたすべてのブロックはハッシュ値を持っていて、それはマイナーによる乱数選択に寄与する。これらの値は次サイクルに公開する義務があり、履行されなければマイナーはマイニング保証金を没収される。この厳しいペナルティは、乱数シードのエントロピーを攻撃するために使用可能な数字を選択的に保留することを防ぐためである。

次サイクルの悪意あるマイナーは、この値の公開を検閲しようとする可能性があるが、複数の値が 1 つのブロックに公開される可能性があるため、これが成功する可能性は非常に低い。

1 サイクル中の全ての公開された値はハッシュリストに格納され、乱数シードは `script` キー導出関数によって root から計算される。キー導出の複雑性は、シード生成が典型的なデスクトップ PC でのブロック検証平均時間の 1 パーセントのオーダーの時間となるよう、調整されるべきである。

3.2.4 Follow-the-coin 手続き

ランダムにステークホルダーを選ぶため、次のコイン手続きを使う:

■原則　このアイデアは、Bitcoin では、follow-the-satoshi として知られている。この手順は、現時点までに鋳造された全ての satoshi にユニークなシリアル番号が割り振られている「かのように」動作する。Satoshi は暗黙のうちに作成時順に並べられ、そこからランダムな satoshi が一つ選択され、ブロックチェーン中を追跡される。もちろん、個々の cent(訳註: satoshi) を直接追跡するわけではない。代わりに、Bitcoin の入力が結合され複数の出力として費やされる際に何が起こるか (訳註: 結果として個々の satoshi の所有権が誰に移動するのか) を記述したルールが使用される。

最後に、アルゴリズムは各キーに関連付けられた区間を追跡する。各区間は satoshi の「範囲」を表す。残念ながら、時間が経つにつれて、データベースはどんどん断片化し、クライアント側で膨らんでいくことになる。

■コインロール　Tezos では、10 000tez 単位で構成された大きな「コインロール」を作ることで、上述のアルゴリズムを最適化する。総計で約 100 万本のロールが存在することになる。データベースは、すべてのロールが現在誰に所有されているのかを管理する。

各アドレスには特定のロールの集合といくらかの端数が結び付けられている。完全なロールの一部のコインを消費すると、ロールは壊され、そのシリアル番号はロールの LIFO キュー、一種の「不確定状態 (limbo)」に送られる。すべてのトランザクションは、壊れたロールの数を最小限に抑える方法で処理される。アドレスがロールを形成するのに十分なコインを保持すると、シリアル番号がこのキューから引き出され、ロールが再

び形成される。

LIFO 優先順位は、密かにフォーク上で作業している攻撃者が、アカウント間のロールの端数をシャッフルしてアカウントが保持しているコインを変更するのを防止する。

このアプローチのわずかな欠点は、ステークが最も近い整数のロール数に切り下げられることだ。しかし、これによって、元の follow-the-satoshi アプローチに比べると効率が大幅に向上する。

ロールが番号付けされているが、このアプローチは、Zerocash のような代替性維持プロトコルの使用を排除するものではない。そのようなプロトコルでも、同じ「limbo」キュー手法を使うことができる。

■動機付け　この手順は、残高による重み付けされたアドレスを単にランダムに選択することとは機能的に異なる。

実際、(訳註: 単に残高による重み付けによるアドレス選択であれば、悪意を持って) 密かに作成されるフォークでは、マイナーはランダムシードの生成をコントロールし、先回りして適切なアドレスを作っておくことで自分に署名と鋳造権を割り当てるように試みることができる。これはロールがランダムに選ばれる場合、よりとても難しくなる。なぜなら密かに作成されるフォークはあるロールの所有者を偽造できないので、署名と鋳造権を自分に割り当てるため、ランダムシードに適用するハッシュ関数を原像計算しなければならないからだ。

実際、長さ $N = 2048$ のサイクルでは、全ロールの f の割合を保持する者は、平均 fN 回マイニング権を得ることができ、受け取る実行割合 f_0 は

$$\sqrt{\frac{1}{N}}\sqrt{\frac{1-f}{f}}.$$

の標準偏差を持つ。

もし攻撃者が W の異なるランダムシードをブルートフォース検索を行なったとすると、攻撃者のアドバンテージは高々[*8]

$$\left(\sqrt{\frac{2\log(W)}{N}}\sqrt{\frac{1-f}{f}}\right)fN$$

ブロックとなる。

たとえば、攻撃者が全体の $f = 10\%$ のロールをコントロールしているとすると、1 サイクルあたり 205 ブロックを採掘できると見込める。密かに作ったフォークで攻撃者はシードをコントロールして、3 兆回以上のハッシュ計算を行うことで約 302 ブロックを自分に割り当てることができる、つまりそれは約 14.7% のブロックになる。次のことに注意しよう:

- シードを計算するためのハッシュは、計算量のかかるキー導出関数なので、総当たり方式の検索は現実的では無い。
- マイニングされたブロックから線形の利益を上げるためには、攻撃者の計算量は二次指数関数的に増える。

3.2.5　ブロックマイニング

乱数シードはロールを選ぶために繰り返し使われる。第一のロールはそのステークホルダーに 1 分後にブロックのマイニングを許す。次のロールは 2 分後 — などなど。

[*8] これは、W の正規分布変数の最大値の期待値に基づく標準である (xxx ???)

ステークホルダーがシードを見て次のサイクルで自分が高い優先度でブロックをマイニングできる権利があるとわかった場合、保証金を預けることができる。

どのステークホルダーもあるブロックをマイニングするための保証金を払わない困った状況を避けるため、16 分の遅れの後、ブロックは保証金なしにマイニングできるようになる。

担保は、ブロックをマイニングしなかった場合、どのチェーンでもすぐにその支払元に返還される。

3.2.6 ブロック署名

Proof of stake システムを動かす道具だてはほとんど揃った。チェーンの重みをブロック数として定義することもできるが、それはセルフィッシュマイニングへの扉を開いてしまうだろう。

そこで、署名方式を導入する。ブロックが鋳造される際に乱数シードを使って 16 の署名権を 16 のロールに割り当てる。

署名権を受け取ったステークホルダーたちは鋳造されようとしているブロックを見てブロックへの署名を投稿する。これらの署名は、ブロックチェーンへの親ブロックの追加を保全しようとするマイナーたちによって、次のブロックに取り込まれる。

署名者が得る署名報酬はブロックとその親との間の時間に逆比例したものとなる。

それ故、署名者は、ある時点で作られた最も良いブロックと彼らが本当に信じられるものに署名する強いインセンティブを持つ。また、署名報酬はそのブロックがブロックチェーンに取り込まれた場合にのみ払われるので、彼らはどのブロックにサインすべきかについて合意しようとする強いインセンティブを持つ。

もし最高優先度を持つブロックがマイニングされなかった場合、(おそらく担当マイナーがネットに接続していなかったため) マイナーの作業が遅れているだけかもしれないので、署名者たちはしばらく待つというインセンティブを持つかもしれない。しかしながら、他の署名者たちが現時点で最良の優先度を持つブロックをサインすると決定し、新しいブロックにこれらの署名が含まれれば、この待ち状態を解消することができる。したがって、マイナーたちは (訳註:遅れている最高優先度のブロックマイニングをただ待つという) この戦略を取ることはまずないだろう。

逆に、署名者たちがパニックに陥り、他の署名者も同じことをして新しいブロックがすぐに作られるだろうという恐れから最初に見たブロックにとにかくサインするという平衡状態を考えることもできるしかしこれは誰も得をしない非常に想定的な状況だ。このように行動するよう彼らのプログラムを変更するという場合を除いて、この平衡状態がよくあると考えるインセンティブは署名者たちには無い。この戦略を取ったとしても、オペレーションを妨害しようとする悪意あるステークホルダーは自分自身を傷つけるだけであろう。なぜなら、他人が同じ戦略を取ることはほとんどないからである。

3.2.7 チェーンの重み

署名の数をチェーンの重みとする。

3.2.8 告発

ブロックの二重鋳造や二重署名を避けるために、マイナーは自分のブロックに告発 (denunciation) を含ませることができる。

告発は、鋳造署名と、ブロックの高さを署名するブロック署名、の二つの署名を持つ。これにより、不正行為の証明を非常に簡潔にできる。

誰でも不正行為を告発することは理論上は可能だが、ブロックマイナーを差し置いて他の誰かに告発を許可

することにはあまり意味がない。実際、マイナーは他人が行なった不正行為の証明を単にコピーして自分の発見だとすることができるからである[*9]。

二重鋳造や二重署名が告発され、それが認められると、鋳造や署名のために預けていた担保は没収される。

3.3 スマートコントラクト

3.3.1 コントラクトの型

未使用アウトプット (訳註: Bitcoin の UTXO のこと) の代わりに、Tezos は状態のあるアカウントを使う。このアカウントが実行可能なコードを指定している場合、アカウントはより一般にコントラクトと呼ばれる。アカウントは一種の (実行可能コードが紐づいていない) コントラクトなので、一般化して両者を「コントラクト」と呼ぶこととする。

各コントラクトには「マネージャ」が割り振られており、アカウントの場合はそれは単にその持ち主である。コントラクトに spendable(消費可能) というフラグが立っていると、マネージャはコントラクトに紐づいた資金を消費することができる。さらに、各コントラクトは proof-of-stake プロトコルにおいてブロックを署名したりマイニングに使う公開鍵のハッシュを指定することができる。コントラクトの秘密鍵はマネージャがコントロールしている場合も、そうでない場合もある。

形式的には、コントラクトは次のように表される:

```
type contract = {
  counter: int;        (* 繰り返し攻撃を防ぐためのカウンタ *)
  manager: id;         (* コントラクトのマネージャの公開鍵ハッシュ *)
  balance: Int64.t;    (* 資金バランス *)
  signer: id option;   (* 署名者のid *)
  code: opcode list;   (* コントラクトコード(オプコードのリストとして) *)
  storage: data list;  (* コントラクトの状態記憶領域 *)
  spendable: bool;     (* マネージャによって金が使用可能かどうか *)
  delegatable: bool;   (* マネージャが署名キーを変更できるかどうか *)
}
```

コントラクトのハンドルは、その初期内容のハッシュとなる。既存のコントラクトと衝突するハッシュを持つコントラクトを生成するのは不正なオペレーションであり、正当なブロックには取り入れられない。

コントラクト記憶領域の型 **data** は次の共用体型で定義される:

```
type data =
  | STRING of string
  | INT of int
```

ここで `INT` は 64 ビット符号付整数で、文字列は最大 1024 バイトの配列である。記憶領域 **storage** の最大サイズは 16 384 バイトに制限されている (整数は 8 バイト、文字列はその長さとカウントする)。

3.3.2 オリジネーション (Origination)

オリジネーション (Origination) 操作は、コントラクトのコードと初期記憶領域内容を与えることで新しいコントラクトを生成する。もしそのハンドルが、すでに存在するコントラクトのハンドルであった場合、オリジネーションは拒否される (間違いか、悪意がなければ、このようなことがまず起こらない)。

[*9] ゼロ知識証明によって誰でも自分の名前で不正行為を告発することで利益を得ることが可能となるが、(全体として) 非常に意味があるかは特に明らかでは無い。

コントラクトが有効であるためには、最低 1 の残高がなければならない。もし残高がこの値を割ってしまうと、コントラクトは破棄されてしまう。

3.3.3 トランザクション

トランザクションはあるコントラクトから別のコントラクトに送られるメッセージである。このメッセージは次のように定義される:

```
type transaction = {
  amount: amount;          (* 転送量 *)
  parameters: data list;   (* スクリプトに渡されるパラメタ *)
  counter: int;            (* 繰り返し攻撃を避けるためのカウンタ(invoice id) *)
  destination: contract hash;
}
```

このようなトランザクションは、コントラクトからマネージャの秘密鍵で署名するか、コントラクト実行中のコードからプログラム的に送ることができる。トランザクションを受け取った時、転送量は宛先のコントラクトの残高に加えられ、宛先のコントラクトコードが実行される。このコードは渡されたパラメータの使用し、コントラクトの記憶領域の読み書きし、署名キーの変更し、他のコントラクトにトランザクションを送ることができる。

カウンタは繰り返し攻撃を防ぐために存在する。トランザクションはコントラクトとトランザクションのカウンタの値が同じでなければ正当では無い。トランザクションが適用されたら、(訳註:コントラクトの) のカウンタが一つ増やされる。これにより同じトランザクションが二度使われることが防がれる。

トランザクションはまたクライアントが正当だと考えている最近のブロックのハッシュを持っている。もし攻撃者が万が一フォークにより長いチェーンの再構成に成功した場合、攻撃者はこのトランザクションを取り入れることができないから、フォークが明らかに偽造したものであるとすぐにわかる。これは最終防御である。TAPOS[*10]は長いチェーン再構成を防ぐ良いシステムだが、短期間の二重使用を防ぐにはあまり良く無い。

(account_handle, counter) の組は、大雑把にいうと Bitcoin の未使用アウトプットと同じものである。

3.3.4 記憶領域使用料

記憶領域はネットワークにコストを強いるので、記憶領域が 1 バイト増えるごとに ꜩ 1 最低料金をかける。例えば、もしトランザクションの実行終了後、整数が一つ記憶領域に付け加えられ、10 文字のデータが記憶領域に既存の文字列に付け加えられた場合、ꜩ 18 がコントラクトの残高から引かれ、破壊される。

3.3.5 コード

言語はスタックベースで、高レベルのデータ型とプリミティブ関数を持ち、厳密な静的型検査を行う。このデザインは Forth、Scheme、ML、そして Cat から発想を得たものだ。命令セットの全仕様は [5] に与えられている。この仕様は完全な命令セット、型システムと言語の意味論を与える。これは正確なリファレンスマニュアルであって、簡単な入門では無い。

[*10] 訳註: この防御方法の名称。“transactions as proof of stake” の略

3.3.6 料金

このシステムは、ここまではEthereumのトランザクション処理方法に似ているが、Tezosでは手数料の扱い方が異なる。Ethereumでは、プログラムの実行時間対し線形に増加する料金を支払いさえすれば、任意の長い時間、プログラムを実行させることができる。残念ながら、これはある1人のマイナーがトランザクションを検証するインセンティブを提供するが、この取引を再検証する必要がある他のマイナー達にはそのようなインセンティブを提供しない。実際には、スマートコントラクトとして使用可能な興味深いプログラムのほとんどは非常に短いので、Tezosでは、プログラムを実行できるステップ数に上限を設定することで、問題の構造を簡潔にする。

あるプログラムに対しこの上限が厳しすぎると判った場合は、プログラム実行を複数に分割し、複数のトランザクションとして全体を実行することができる。Tezosのプロトコルは改訂可能であるから、将来この上限を変更することが可能だし、また、高度なプリミティブを新しいオプコードとして導入することもできる。

アカウントが許可するならば、変更を要求する署名付きメッセージを発行することで、署名キーを変更することができる。

4 結論

我々は魅力的な開始プロトコルを構築できたと思う。しかしTezosの真の可能性は、ステークホルダーに、彼ら自身にとって最も良いと思うプロトコルを決定する役割を与えることにある。

参考文献

[1] Vitalik Buterin. Slasher: A punitive proof-of-stake algorithm. `https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm/`, 2014.

[2] Ariel Gabizon Iddo Bentov and Alex Mizrahi. Cryptocurrencies without proof of work. `http://www.cs.technion.ac.il/~idddo/CoA.pdf`, 2014.

[3] Peter Suber. Nomic: A game of self-amendment. `http://legacy.earlham.edu/~peters/writing/nomic.htm`, 1982.

[4] Jérôme Vouillon. Lwt: a cooperative thread library. 2008.

[5] Tezos project. Formal specification of the Tezos smart contract language. `https://tezos.com/pages/tech.html`, 2014.