

# Tezos — a self-amending crypto-ledger

## White paper alpha

L.M Goodman

August 12, 2014

*“Our argument is not flatly  
circular, but something like it.”*

— Willard van Orman Quine

### Abstract

We present Tezos, a generic and self-amending crypto-ledger. Tezos can instantiate any blockchain based ledger. The operations of a regular blockchain are implemented as a purely functional module abstracted into a shell responsible for network operations. Bitcoin, Ethereum, Cryptonote, etc. can all be represented within Tezos by implementing the proper interface to the network layer.

Most importantly, Tezos supports meta upgrades: the protocols can evolve by amending their own code. To achieve this, Tezos begins with a seed protocol defining a procedure for stakeholders to approve amendments to the protocol, *including* amendments to the voting procedure itself. This is not unlike philosopher Peter Suber’s Nomic[1], a game built around a fully introspective set of rules.

In addition, Tezos’s seed protocol is based on a pure proof-of-stake system and supports Turing complete smart contracts. Tezos is implemented in OCaml, a powerful functional programming language offering speed, an unambiguous syntax and semantic, and an ecosystem making Tezos a good candidate for formal proofs of correctness.

Familiarity with the Bitcoin protocol and basic cryptographic primitives are assumed in the rest of this paper.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Self-amending cryptolledger</b>	<b>3</b>
2.1	Mathematical representation . . . . .	3
2.2	The network shell . . . . .	4
2.2.1	Clock . . . . .	4
2.2.2	Chain selection algorithm . . . . .	4
2.2.3	Network level defense . . . . .	4
2.3	Functional representation . . . . .	5
2.3.1	Validating the chain . . . . .	5
2.3.2	Amending the protocol . . . . .	6
2.3.3	RPC . . . . .	7
<b>3</b>	<b>Seed protocol</b>	<b>7</b>
3.1	Economy . . . . .	8
3.1.1	Coins . . . . .	8
3.1.2	Amendment rules . . . . .	8
3.2	Proof-of-stake mechanism . . . . .	9
3.2.1	Overview . . . . .	9
3.2.2	Clock . . . . .	9
3.2.3	Generating the random seed . . . . .	9
3.2.4	Follow-the-coin procedure . . . . .	10
3.2.5	Minting blocks . . . . .	11
3.2.6	Signing blocks . . . . .	11
3.3	Weight of the chain . . . . .	12
3.3.1	Denunciations . . . . .	12
3.4	Smart contracts . . . . .	12
<b>4</b>	<b>Conclusion</b>	<b>14</b>

# 1 Introduction

In the first part of this paper, we will discuss the concept of abstract blockchains and the implementation of a self-amending crypto-ledger. In the second part, we will describe our proposed seed protocol.

## 2 Self-amending cryptoledger

A blockchain protocol can be decomposed into three distinct protocols:

- The network protocol discovers blocks and broadcasts transactions.
- The transaction protocol specifies what makes a transaction valid.
- The consensus protocol forms consensus around a unique chain.

Tezos implements a generic network shell. This shell is agnostic to the transaction protocol and to the consensus protocol. We refer to the transaction protocol and the consensus protocol together as a “blockchain protocol”. We will first give a mathematical representation of a blockchain protocol and then describe some of the implementation choices in Tezos.

### 2.1 Mathematical representation

A blockchain protocol is fundamentally a monadic implementation of concurrent mutations of a global state. This is achieved by defining “blocks” as operators acting on this global state. The free monoid of blocks acting on the genesis state forms a tree structure. A global, canonical, state is defined as the minimal leaf for a specified ordering.

This suggests the following abstract representation:

- Let  $(\mathbf{S}, \leq)$  be a totally ordered, countable, set of possible states.
- Let  $\emptyset \notin \mathbf{S}$  represent a special, invalid, state.
- Let  $\mathbf{B} \subset \mathbf{S}^{\mathbf{S} \cup \{\emptyset\}}$  be the set of blocks. The set of *valid* blocks is  $\mathbf{B} \cap \mathbf{S}^{\mathbf{S}}$ .

The total order on  $\mathbf{S}$  is extended so that  $\forall s \in \mathbf{S}, \emptyset < s$ . This order determines which leaf in the block tree is considered to be the canonical one. Blocks in  $\mathbf{B}$  are seen as operators acting on the state.

All in all, any blockchain protocol<sup>1</sup> (be it Bitcoin, Litecoin, Peercoin, Ethereum, Cryptonote, etc) can be fully determined by the tuple:

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

---

<sup>1</sup>GHOST is an approach which orders the leafs based on properties of the tree. Such an approach is problematic for both theoretical and practical reasons. It is almost always better to emulate by inserting proofs of mining in the main chain

The networking protocol is fundamentally identical for these blockchains. “Mining” algorithms are but an emergent property of the network, given the incentives for block creation.

We make a blockchain protocol introspective by letting blocks act on the protocol itself. We can express the set of protocols recursively as

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

## 2.2 The network shell

This formal mathematical description doesn’t tell us *how* to build the block tree. This is the role of the network shell, which acts as an interface between a gossip network and the protocol.

The network shell works by maintaining the best chain known to the client. It is aware of three type of objects. The first two are transactions and blocks, which are only propagated through the network if deemed valid. The third are protocols, OCaml modules used to amend the existing protocol. They will be described in more details later on. For now we will focus on transaction and blocks.

The most arduous part of the network shell is in protecting nodes against denial-of-service attacks.

### 2.2.1 Clock

Every block carries a timestamp visible to the network shell. Blocks that appear to come from the future are buffered if their timestamps are within a few minutes of the system time and rejected otherwise. The protocol design must tolerate reasonable clock drifts in the clients and be resilient to falsified timestamps.

### 2.2.2 Chain selection algorithm

The shell maintains a single chain rather than a full tree of blocks. This chain is only overwritten if the client becomes aware of a strictly better chain.

Maintaining a tree would be more parsimonious in terms of network communications but would be susceptible to denial-of-service attacks where an attacker produces a large number of low-scoring but valid forks.

Yet, it remains possible for a node to lie about the score of a given chain, a lie that the client may only uncover after having processed a potentially large number of blocks. However, such a node can be subsequently ignored.

Fortunately, a protocol can have the property that low scoring scores exhibit a low rate of block creation. Thus, the client would only consider a few blocks of a “weak” fork before concluding that the announced score was a lie.

### 2.2.3 Network level defense

The shell is defensive. It attempts to connect to many peers across various IP ranges. It detects disconnected peers and bans malicious peers.

To protect against certain denial of service attacks, the protocol provides the shell with context dependent bounds on the size of blocks and transactions.

## 2.3 Functional representation

### 2.3.1 Validating the chain

We can efficiently capture almost all the genericity of our abstract blockchain structure with the following OCaml types. To begin with, a block header is defined as:

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

We are purposefully not typing the header field more strongly so it can represent arbitrary content. However, we do type the fields necessary for the operation of the shell. These include the hash of the preceding block, a list of operation hashes and a timestamp. In practice, the operations included in a block are transmitted along with the blocks at the network level. Operations themselves are represented as arbitrary blobs.

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

The state is represented with the help of a **Context** module which encapsulates a disk-based immutable key-value store. The structure of a key-value store is versatile and allows us to efficiently represent a wide variety of states.

```
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
```

To avoid blocking on disk operations, the functions use the asynchronous monad `Lwt`[2]. Note that the operations on the context are purely functional: **get** uses the **option** monad rather than throwing an exception while **set** and **del** both return a new **Context**. The **Context** module uses a combination of memory caching and disk storage to efficiently provide the appearance of an immutable store.

We can now define the module type of an arbitrary blockchain protocol:

```
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
end
```

```

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

```

We no longer compare states directly as in the mathematical model, instead we project the **Context** onto a list of bytes using the **score** function. List of bytes are ordered first by length, then by lexicographic order. This is a fairly generic structure, similar to the one used in software versioning, which is quite versatile in representing various orderings.

Why not define a comparison function within the protocol modules? First off it would be hard to enforce the requirement that such a function represent a *total* order. The score projection always verifies this (ties can be broken based on the hash of the last block). Second, in principle we need the ability to compare states across distinct protocols. Specific protocol amendment rules are likely to make this extremely unlikely to ever happen, but the network shell does not know that.

The operations **parse\_block\_header** and **parse\_operation** are exposed to the shell and allow it to pass fully typed operations and blocks to the protocol but also to check whether these operations and blocks are well-formed, before deciding to relay operations or to add blocks to the local block tree database.

The apply function is the heart of the protocol:

- When it is passed a block header and the associated list of operations, it computes the changes made to the context and returns a modified copy. Internally, only the difference is stored, as in a versioning system, using the block's hash as a version handle.
- When it is only passed a list of operations, it greedily attempts to apply as many operations as possible. This function is not necessary for the protocol itself but is of great use to miners attempting to form valid blocks.

### 2.3.2 Amending the protocol

Tezos's most powerful feature is its ability to implement protocol capable of self-amendment. This is achieved by exposing two procedures functions to the protocol:

- **set\_test\_protocol** which replaces the protocol used in the testnet with a new protocol (typically one that has been adopted through a stakeholder voter).
- **promote\_test\_protocol** which replaces the current protocol with the protocol currently being tested

These functions transform a `Context` by changing the associated protocol. The new protocol takes effect when the following block is applied to the chain.

```
module Context = sig
  type t
  (*...*)
  val set_test_protocol: t -> Protocol_hash.t Lwt.t
  val promote_test_protocol: t -> Protocol_hash.t -> t Lwt.t
end
```

The `protocol_hash` is the `sha256` hash of a tarball of `.ml` and `.mli` files. These files are compiled on the fly. They have access to a small standard library but are sandboxed and may not make any system call.

These functions are called through the `apply` function of the protocol which returns the new `Context`.

Many conditions can trigger a change of protocol. In its simplest version, a stakeholder vote triggers a change of protocol. More complicated rules can be progressively voted in. For instance, if the stakeholder desire they may pass an amendment that will require further amendments to provide a computer checkable proof that the new amendment respects certain properties. This is effectively an algorithm check of “constitutionality”.

### 2.3.3 RPC

In order to make the GUI building job’s easier, the protocol exposes a JSON-RPC API. The API itself is described by a json schema indicating the types of the various procedures. Typically, functions such as `get_balance` can be implemented in the RPC.

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

The name is a list of string to allow namespaces in the procedures. Input and output are optionally described by a json schema. Note that the call is made on a given context which is typically a recent ancestor of the highest scoring leaf (depending on the number of confirmations sought).

## 3 Seed protocol

Much like blockchains start from a genesis hash, Tezos starts with a seed protocol. This protocol can be amended to reflect virtually any blockchain based algorithm.

## 3.1 Economy

### 3.1.1 Coins

There are initially 10,000,000,000 (ten billion) coins, divisible up to two decimal places. We suggest that a single coin be referred to as a “Tez” and that the smallest unit simply as a cent. We also suggest to use the symbol  $\mathfrak{B}$  (“Latin capital letter tz”) to represent a Tez.

Therefore 1 cent =  $0.01\mathfrak{B}$  = one hundredth of a Tez.

### 3.1.2 Amendment rules

Amendments are adopted over election cycles lasting  $N = 2^{17} = 131,072$  blocks each. Given the a one minute block interval, this is about three calendar months. The election cycle is itself divided in four quarters of 32,768 blocks. This cycle is relatively short to encourage early improvements, but it is expected that further amendments will increase the length of the cycle. Adoption requires a certain quorum to be met. This quorum starts at  $Q = 80\%$  but dynamically adapts to reflect the average participation. This is necessary if only to deal with lost coins.

**First quarter** Protocol amendments are suggested by submitting the hash of a tarball of `.ml` and `.mli` files representing a new protocol. Stakeholders may approve of any number of these protocols. This is known as “approval voting”, a particularly robust voting procedure.

**Second quarter** The amendment receiving the most approval in the first quarter is now subject to a vote. Stakeholders may cast a vote for, against or can choose to explicitly abstain. Abstentions count towards the quorum.

**Third quarter** If the quorum is met (including explicit abstentions), and the amendment received 80% of yays, the amendment is approved and replaces the test protocol. Otherwise, it is rejected. Assuming the quorum reached was  $q$ , the minimum quorum  $Q$  is updated as such:

$$Q \leftarrow (1 - \alpha)Q + \alpha q, \alpha = 0.2$$

The goal of this update is to avoid lost coins causing the voting procedure to become stuck over time. The minimum quorum is an exponential moving average of the quorum reached over each previous election.

**Fourth quarter** Assuming the amendment was approved, it will have been running in the testnet since the beginning of the third quarter. The stakeholders vote a second time to confirm they wish to promote the test protocol to the main protocol. This also requires the quorum to be met and an 80% supermajority.

We deliberately chose a conservative approach to amendments. However, stakeholders are free to adopt amendments loosening or tightening this policy should they deem it beneficial



## 3.2 Proof-of-stake mechanism

### 3.2.1 Overview

Our proof-of-stake mechanism is a mix of several ideas, including slasher, chain-of-activity and proof-of-burn.

The following is a brief overview of the algorithm, the components of which are explained in more details below.

Mining rights are randomly allocated to stakeholders. In addition, each block is signed by multiple randomly selected stakeholders. Mining and signing both offer a small reward but also require making a one year safety deposit ( $2^{19}$  blocks) to significantly cut the risk of short term forks.

The protocol unfolds in cycles of 2048 blocks. At the beginning of each cycle, a random seed is derived from random numbers that block miners committed to in the penultimate cycle, and revealed in the last. Using this random seed, a follow the coin strategy is used to allocate mining rights and signing rights to specific addresses for the next cycle. See figure 1.

### 3.2.2 Clock

The system paces itself and attempts to produce a block every minute. This is achieved by imposing that successive block timestamps be at least one minute apart. A block creator could of course postdate a block, but there is no incentive to do so.

### 3.2.3 Generating the random seed

Every block mined carries a hash commitment to a random number chosen by the miner. These numbers must be revealed in the next cycle under penalty of forfeiting the safety bond. This harsh penalty is meant to prevent selective withholding of the numbers to attack the entropy of the seed.

Malicious miners in the next cycle could attempt to censor such reveals, however since multiple numbers may be revealed in a single block, they are very unlikely to succeed.

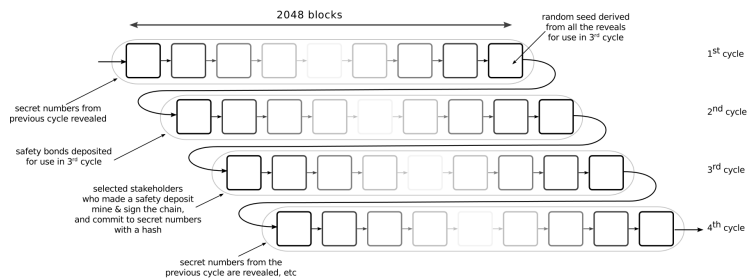


Figure 1: Four cycles of the proof-of-stake mechanism

All the revealed numbers in a cycle are combined in a hash list and the seed is derived from the root using the `scrypt` key derivation function. The key derivation should be tuned so that deriving the seed takes at most a few percents of the average validation time for a block on a typical desktop PC.

### 3.2.4 Follow-the-coin procedure

In order to randomly select a stakeholder, we use a follow the coin procedure.

**Principle** The idea is known in bitcoin as follow-the-satoshi. The procedure works “as-if” every satoshi ever minted had a unique serial number. Satoshis are implicitly ordered by creation time, a random satoshi is drawn and tracked through the blockchain. Of course, individual cents are not tracked directly. Instead, rules are applied to describe what happens when inputs are combined and spent over multiple output.

In the end, the algorithm keeps track of a set of intervals associated with each key. Each interval represents a “range” of satoshis. Unfortunately, over time, the database becomes more and more fragmented, increasing bloat on the client side.

**Optimization** We optimize the previous algorithm by constructing large “coins” made up of many units. The size of the “coin” is such that there are only on the order of a million coins. A database maps every coin to its current owner. Of course, we need to explain what happens when less than a coin is being spent.

We implicitly create a FIFO pile of coins in “limbo”. Each address holds a certain set of specific coins and some change. When an address attempts to make a transfer, it does so in a way that breaks as few coins as possible. However, if a coin is to be broken, that coin is placed in limbo. Whenever an address has enough change to form a coin, a coin is pulled from limbo and assigned to that address.

The FIFO priority ensures that an attacker working on a secret fork cannot change the coins he holds by shuffling change between accounts.

A slight drawback of this approach is that stake is rounded down to the nearest integer number of coins. However, this provides a massive improvement in efficiency over the follow-the-satoshi approach.

Note that the approach consisting in putting coins in and out of “limbo” is directly applicable to history-free blockchains like Zerocash.

**Motivation** This procedure is functionally distinct than merely drawing a random address.

Indeed, in a secretive fork, a miner could attempt to control the generation of the random seed and to assign itself signing and minting rights by creating the appropriate addresses ahead of time. This is much harder to achieve if coins are randomly selected, as the secretive fork cannot fake ownership of certain coins and must thus try to preimage the hash function applied to the seed to assign itself signing and minting rights.

Indeed, in a cycle of length  $N = 2048$ , someone holding a fraction  $f$  of the coins will receive on average  $fN$  mining rights, with a standard deviation of  $\sqrt{Nf(1-f)}$ , thus the effective fraction received,  $\hat{f}$  has a standard deviation of  $\sqrt{\frac{1}{N}}\sqrt{\frac{1-f}{f}}$ . If an attacker can control a number  $b$  bits of the random seed using brute force search, then his expected advantage is at most

$$\sqrt{\frac{2 \log(2)b}{N}} \sqrt{\frac{1-f}{f}}$$

For instance, an attacker controlling 10% of the coins should expect to mine about 204.8 blocks per cycle. In a secret fork where he controlled the seed, assuming he computed over a trillion hash, he could assign itself about 302.3 blocks, or about 14.7% of the blocks. Note that:

- The hash from which the seed is derived is an expensive key derivation function, rendering brute-force search impractical
- To make linear gains in blocks mined, the attacked needs to expend a quadratically exponential effort

### 3.2.5 Minting blocks

The random seed is used to repeatedly select a coin. The first coin selected is given a minting priority of  $2^{32}$ , the second  $2^{31}$  – and so on, halving the priority at every step until it reaches 1, at which point it remains constant.

When a stakeholder realizes he can mint a high priority block, he may purchase a safety bond in the next cycle.

Any address can mint the next block in the chain, but as we'll see, in practice, only the highest priority blocks have any chance to be included in the blockchain.

### 3.2.6 Signing blocks

As it is, we almost have a working proof of stake system. We could define a chain's weight to be the sum of block priorities; however, this opens the door to a form of selfish mining.

We thus introduce a signing scheme. While a block is being minted, the random seed is used to randomly assign 32 signing rights to 32 coins.

The stakeholders who received signing rights observe the blocks being minted and then submit signatures of that blocks. Those signatures are then included in the next block, by minters attempting to secure their parent's inclusion in the blockchain.

The signing reward received by signers is proportional to the block's priority. Signer thus have a strong incentive to sign what they genuinely believe to be the best block produced at one point. They also have a strong incentive to agree on which block they will sign as signing rewards are only paid if the block ends up included in the blockchain.

If the highest priority block isn't mined (perhaps because the minter isn't on line), there could be an incentive for signers to wait for a while, just in case the minter is late. However, other signers may then decide to sign the best priority block, and a new block could include those signatures, leaving out the holdouts. Thus, minters are unlikely to follow this strategy.

Conversely, we could imagine an equilibrium where signers panic and start signing the first block they see, for fear that other signers will do so and that a new block will be built immediately. This is however a very contrived situation which benefits no one. There is no incentive for signers to think this equilibrium is likely, let alone to modify the code of their program to act this way. A malicious delegate attempting to disrupt the operations would only hurt itself by attempting to follow this strategy, as others would be unlikely to follow suit.

### 3.3 Weight of the chain

The weight is the number of signatures.

#### 3.3.1 Denunciations

In order to avoid the double minting of a block or the double signing of a block, a minter may include in his block a denunciation.

This denunciation takes the form of two signatures. Each minting signature or block signature signs the height of the block, making the proof of malfeasance quite concise.

While we could allow anyone to denounce malfeasance, there is really no point to allow anyone else beyond the block miner. Indeed, a minter can simply copy any proof of malfeasance and pass it off as its own discovery.<sup>2</sup>

Once a party has been found guilty of double minting or double signing, the safety bond is forfeited.

### 3.4 Smart contracts

In lieu of unspent outputs, Tezos uses accounts, similar to Ethereum's. Unlike unspent outputs, accounts/contracts are stateful. In practice, an account is represented as:

```
type contract = {
  counter: int; (* counter to prevent repeat attacks *)
  manager: id; (* hash of the contract's manager public key *)
  balance: Int64.t; (* balance held *)
  delegate: id option; (* id of the delegate *)
  code: opcode list; (* contract code as a list of opcodes *)
  storage: data list; (* storage of the contract *)
  spendable: bool; (* may the money be spent by the manager? *)
  delegatable: bool; (* may the delegate be changed by the manager? *)
}
```

---

<sup>2</sup>A zero-knowledge proof would allow anyone to benefit from denouncing malfeasances, but it's not particularly clear this carries much benefit.

The handle of an account is the hash of its initial content: hash of public key, initial delegate, initial contract, initial stored data and account options.

A transaction is a message sent from one account to another account, this messages is represented as:

```
type transaction = {
  amount: amount; (* amount being sent *)
  parameters: data list; (* parameters passed to the script *)
  (* counter (invoice id) to avoid repeat attacks *)
  counter: int;
  destination: contract hash;
}
```

Such a transaction can be sent from an account if signed using the account's key or can be sent programmatically by code executing in the account. When the transaction is received, the amount is added to the destination account's balance and the destination account's code is executed. This code can make use of the parameters passed to it, it can read and write the account's storage, change the delegate handle and send transactions to other accounts.

The role of the counter is to prevent replay attacks. A transaction is only valid if the account's counter is equal to the transaction's counter. Once a' transaction is applied, the counter increases by one, preventing the transaction from being reused.

The transaction also includes the block hash of a recent block that the client considers valid. If an attacker ever succeeds in forcing a long reorganization with a' fork, he will be unable to include such transactions, making the fork obviously fake. This is a last line of defense, TAPOS is a great system to prevent long reorganizations but not a very good system to prevent short term double spending.

A special transaction may be used to originate a new contract, it will specify the code of the contract and the initial content of the account's storage. If the handle is already the handle of an existing contract, the origination is rejected (there is no reason for this to ever happen, unless by mistake or malice). A contract needs a minimum balance to stay active, if the balance falls below this number, the account is automatically destroyed.

The pair (account\_handle, counter) is roughly the equivalent of an unspent output in Bitcoin.

So far, this system is similar to the way Ethereum handles transaction. However, we differ in the way we handle fees. Ethereum allows arbitrarily long programs to execute by requiring a fee that increases linearly with the program's executing time. Unfortunately, while this does provide an incentive for one miner to verify the transaction, it does not provide such an incentive to other miners, who must also verify this transaction. In practice, most of the interesting programs that can be used for smart contracts are very short. Thus, we simplify the construction by imposing a hard cap on the number of steps we allow the programs to run for.

If the hard cap proves too tight for some programs, they can break the execution in multiple steps and use multiple transactions to execute fully. Since Tezos

is amendable, this cap can be changed in the future, or advanced primitives can be introduced as new opcodes.

If the account permits, the delegate's handle may be changed by issuing a signed message requesting the change.

## 4 Conclusion

We feel we've built an appealing seed protocol. However, Tezos's true potential lies in putting the stakeholders in charge of deciding on a protocol that they feel best serves them.

## References

- [1] Peter Suber. Nomic: A game of self-amendment. <http://legacy.earlham.edu/~peters/writing/nomic.htm>, 1982.
- [2] Jérôme Vouillon. Lwt: a cooperative thread library. 2008.