Algorand Protocol Description

Argimiro, CoinFabrik

July 7, 2023

Contents

1 Introduction

We aim to describe how the Algorand protocol works. This document is meant to reflect how the Algorand mainnet is behaving today. We relied on Algorand's documents ([GHM+17], [Mic16] and Algorand's official specifications) but consulted the node's code and probed the network when information was unclear or unavailable.

Algorand is a proof-of-stake blockchain cryptocurrency protocol using a decentralized Byzantine Agreement protocol that leverages pure proof of stake for consensus. The protocol maintains a ledger that is modified via consensus. In particular, this ledger encodes how many ALGOs (the native token) holds each account. Blocks encode the status of the ledger. Starting on the genesis block (round 0) that encodes the first state of the network, on each round the participation nodes vote on what will the next block be. Their voting power is proportional to the stake (in ALGOs) held by the accounts assoicated to the node.

At the genesis of the Algorand blockchain, 10Bn ALGO was minted. As of September 2022, circulating supply is approximately 6.9b ALGO, distributed through different forms of ecosystem support and community incentives. The remaining Algos are held by the Foundation in secure wallets assigned to Community and Governance Rewards (%53), Ecosystem Support (%36), and Foundation endowment (%11).

In the following document we present a detailed pseudocode interpretation of the program run by an Algorand node managing at least one online (i.e., participating) account. We attempt to explain the different states the node can be in, as well as all kinds of events that cause transitions

and network output. This work aims to provide a basis for an Algorand blockchain simulator currently in development.

2 Network

A network is formed with two kind of nodes: relay and participation nodes. Participation nodes can connect only to relay nodes. They are non-archival in nature, meaning they don't keep the whole ledger in memory at all times. At the time of writing, participation nodes keep the latest 1000 blocks. Any queries that require blocks prior to those, need to be made through one of the relay nodes they are connected to. These nodes are responsible of running the consensus algorithm that will be outlined in this document. Relay nodes on the other hand, are archival (meaning they have immediate access to the whole ledger), but do not participate in consensus. They function as network level facilitators, collecting messages sent by participation nodes or other relay nodes and distributing them across the network.

Currently, any individual or entity is able to run a participation node, but relay nodes are permissioned.

When a participation node is booted up, it establishes a P2P connection with 4 relay nodes on average, using a phonebook. Relay nodes manage a significantly higher number of connections, as they are responsible of keeping the network's high throughput.

Accounts are managed by participation nodes. Accounts should be registered in one participation node only, however this is not enforced. In order for an account to participate in consensus, it needs to switch its status to "online". That is achieved by sending a special "keyreg" transaction, which registers a participation key for the account and creates the two-level ephemeral key tree. Its important to note that the registered account will only start participating in consensus δ_r rounds after the key registration transaction is approved. Any account participating in consensus does so with their full balance. There is no risk associated to participation, nor there are any rewards for running a node.

TODO: maybe include some voting stats?

3 Node as a State Machine

A network participation node can be modeled as a state machine. In this section we'll define all primitives and data structures necessary to define a node's given state; the rest of the work will be spent on describing the rules and interrelations that make nodes transition from and into different states.

Let round and period number, r and p, be unsigned 64 bit integers. Let the step number, s, be an unsigned 8 bit integer. For convenience and readability, we define the following enumeration for step number:

- $s = 0 \equiv proposal$
- $s = 1 \equiv soft$
- $s = 2 \equiv cert$
- $s \in [3, 252] \equiv next_{s-3}$
- $s = 253 \equiv late$
- $s = 254 \equiv redo$
- $s = 255 \equiv down$

A **ledger** is a sequence of states which comprise the common information established by some instantiation of the Algorand protocol. It is identified by a string called the genesis identifier, as

well as a genesis hash that cryptographically commits to the starting state of the ledger. Besides the already defined round number r, which indexes into the Ledger's sequence of states, a Ledger's state is defined by the following components:

- Genesis identifier and genesis hash, unambiguously defining the ledger to which the state belongs
- Protocol version and update state
- Timestamp (in miliseconds since genesis block), identifying when the state was first proposed
- A 64 byte seed, source of randmoness for the consensus algorithm
- The current reward state, which describes the policy at which incentives are distributed to participants
- The current box state, which holds mappings from (app id, name) tuples to box contents of arbitrary bytes.

A Ledger's state is completed by a Balance table, BT, a mapping of addresses to account records. An account record is comprised of a 64 bit unsigned integer raw balance, a 3-state flag status (can be "online", "offline", or "non-participating"), its registered voting keys (more info on the keys subsection) and two 64 bit unsigned integers r_start and r_last, which represent the validity interval (in rounds) of the participation key (which acts as the root voting key for the merkle tree). A node may query the current state of a particular account by indexing the balance table by address, or it might query a previous round's account state by adding a round indexing. Note that past account states are always computable by iterating over the Ledger sequence of blocks and reverting transaction updates. Internally however, certain past records might be cached and maintained to improve performance.

For convenience we assume the existance of a locally maintained online account list, where a node has easy iteration access to all updated balance records of the accounts it manages that are flagged as online and have a valid participation key registered on this node. This array is named A from now on, and each entry a has an account address $a.I$, and all of the information in its mapped balance record, $BT[a.I]$, exposed and accesible.

Furthermore, a node posesses a Transaction pool, TP, which is a set of live unconfirmed transactions, txn_k, (where k is an account's address, the creator and sender of the transaction), either sent by accounts that are managed by the node itself or obtained from the network (broadcast by other nodes). For the purpose of this report, transactions are an otherwise opaque object, with the attribution of modifying account records.

A **block** is a data structure which specifies the transition between states. The data in a block is divided between the block header and its block body. A block header holds the following data:

- The block's round, which matches the round of the state it is transitioning into. (The block with round 0 is special in that this block specifies not a transition but rather the entire initial state, which is called the genesis state. This block is correspondingly called the genesis block).
- The block's genesis identifier and genesis hash, constant and decided on Ledger creation.
- The block's upgrade vote, which results in the new upgrade state. The block also duplicates the upgrade state of the state it transitions into.
- The block's timestamp, which matches the timestamp of the state it transitions into.
- The block's seed, which matches the seed of the state it transitions into.
- The block's reward updates, which results in the new reward state. The block also duplicates the reward state of the state it transitions into.

- A cryptographic commitment to the block's transaction sequence (a vector commitment).
- A cryptographic commitment, using SHA256 hash function, to the block's transaction sequence (useful for compatibility with systems where SHA512/256 is not supported)
- The block's previous hash, which is the cryptographic hash of the previous block in the sequence. (The previous hash of the genesis block is 0).
- The block's transaction counter, which is the total number of transactions issued prior to this block. This count starts from the first block with a protocol version that supported the transaction counter.
- The block's expired participation accounts, which contains an optional slice of public keys of accounts. These accounts are expected to have their participation key expire by the end of the round (or was expired before the current round). The block's expired participation accounts slice is valid as long as the participation keys of all the accounts in the slice are expired by the end of the round or were expired before, the accounts themselves would have been online at the end of the round if they were not included in the slice, and the number of elements in the slice is less or equal to 32. A block proposer may not include all such accounts in the slice and may even omit the slice completely

While a block's body is the block's transaction sequence, which describes the sequence of updates to the account state and box state

Applying a valid block to a state produces a new state by updating each of its components. A Ledger's evolution in time can then be specified as an ordered sequence of blocks.

A proposal-value is a tuple $v = (I, p, Hash(e), Hash(Encoding(e)))$ where I is an address (the "original proposer"), p is a period (the "original period"), and Hash is some cryptographic hash function (implemented as SHA512/256). The special proposal where all fields are the zero-string is called the bottom proposal $\perp$. It also includes the authentication information for the original proposer, that is, a signature and VRF proof of the proposal-value.

For convenience, we define the function $Proposal(v)$ for a given proposal value v, such that $Proposal(v) = e$ iff e is the proposal that, when hashed, corresponds to the proposal value v. The $Proposal()$ function has an inverse, $Proposal_{value}(e)$, such that $Proposal_{value}(e) = v$ if $Proposal(v) = e$. $Proposal()$ is not a biyective function (as hashes can theoretically collide), however we assume the inverse exists and has a defined output whenever the relevant proposal has been observed by the node (and is such that $e \in P$).

We define an auxiliary structure, *credentials*, useful in voting with the following fields:

- sh is the sortition hash (64 byte string)
- π is the sortition proof (32 byte string)
- j is a 64 bit unsigned integer that represents the vote's weight

An object of this type is output by the Sortition() procedure.

A vote vt, constructed as $Vote(I, r, p, s, v, cred)$ is a tuple with the following members:

- I is a valid Algorand address (in 32 byte format)
- r is the vote's round (64 bit integer)
- p is the vote's period (64 bit integer)
- s is the vote's step (8 bit integer)
- v is the vote's proposal value
- $cred$, of type $credentials$, the committee credentials of the voter

In reality all votes are broadcast wrapped in a structure called $UnauthenticatedVote$, where a $credentials$ field is added, containing only the voter's sortition proof and address. The rest of relevant data is reconstructed as part of the verification process, where a node authenticates the received vote and obtains the vote weight and sortition hash. For the sake of simplicity we abstract away this step and assume availability of all $credentials$ fields.

Consider now a pair of votes, vt_0 and vt_1, constructed like $vt_0 = Vote(I, r, p, s, v_0, cred)$ and $vt_1 = Vote(I, r, p, s, v_1, cred)$, where all values are equivalent save for the proposal value, that is, $v_0 \neq v_1$. We call this pattern an **equivocationvote**. An equivocation vote is valid if both of its constituent votes are valid. It's important to keep in mind that nodes are forbidden from equivocating on any steps $s \neq soft$.

Let set b of votes, $vt_0, vt_1 \in b \implies vt_0.r = vt_1.r \wedge vt_0.p = vt_1.p \wedge vt_0.s = vt_1.s$ and for each pair of distinct elements, either $vt_0.I \neq vt_1.I \wedge vt_0.v = vt_1.v$ or the pair vt_0, vt_1 is an equivocation vote. Also, the number of votes in b is such that the sum of weights for each vote in b is greater or equal to the committee threshold of the votes in the set. A set b with these characteristics is called a **Bundle**, constructed $b = Bundle(r, p, s, v)$.

We define a player's state, S, as a tuple of values, along with a ledger state L and a balance table BT $S = (r, p, s, V, P, \bar{v}, \bar{s}, L, BT)$ where

- r is the current round
- p is the current period
- s is the current step
- V is all valid observed votes and equivocation votes in the current execution state ((r, p, s) tuple)
- P is all valid observed proposals in the current execution state ((r, p, s) tuple)
- $barv$ is the pinned proposal-value
- $bars$ is the last finished step
- L is the Ledger state. Note that, thanks to the genesis hash and previous hash components, a ledger state unambiguously defines the whole history of state changes.
- BT is the current balance table as described above.
- TP is the node's transaction pool

The first 7 values are the consensus state parameters, whereas L and BT are ledger state parameters. Note that functionally, a node plays for all online accounts it manages, so all individual accounts playing in a specific node will be at the same state at all times.

Given all of the above data types and constructs, a node can be modelled as a finite state machine. A state is given by a 3-tuple of integers, (r, p, s), two sets of objects P and V which we will call observed *proposal* and *vote* values, a Ledger L and a balance table BT.

For convenience, we define two special values: $\mu(r, p)$ and $\sigma(r, p)$. $\mu(r, p)$ is the highest priority observed proposal-value in the current (r, p) context (lowest hashed according to the function outlined in the BlockProposal algorithm), or $\perp$ if no valid proposal vote has been observed by the node. $\sigma(r, p)$ is the sole proposal-value for which a soft bundle has been observed (again, in the current (r, p) context), or $\perp$ if no valid soft bundle has been observed by the node.

4 Main algorithm

Algorithm 1 Main node algorithm

```
1: function EventHandler(Event ev)
2:     if ev is TimeoutEvent then
3:         time ← ev.time
4:         if time = 0 then
5:             BlockProposal()
6:         else if time = FilterTimeout(p) then
7:             SoftVote()
8:         else if time = max{4λ, Λ} then
9:             Recovery(next_0)
10:        else if time = max{4λ, Λ} + 2^(st−3)λ + r, st ∈ [4, 252], r ∈ U[0, 2^(st−3)λ] then
11:            Recovery(next_(st−3))
12:        else if time = kλ_f + r, k ∈ ℤ ∧ k > 0, r ∈ U[0, λ_f] then
13:            FastRecovery()
14:    else //ev is MessageEvent
15:        msg ← ev.msg
16:        if msg is Proposal p then
17:            HandleProposal(p)
18:        else if msg is Vote v then
19:            HandleVote(v)
20:        else if msg is Bundle b then
21:            HandleBundle(b)
```

On a higher level, we can think of a step as a defined part of the consensus algorithm. The first three steps (*proposal*, $soft$ and *cert*) are the fundamental parts, and will be the only steps run in normal, "healthy" functioning conditions. The following steps are recovery procedures in case there's no observable consensus before their trigger times. $next_{s-3}$ with $s \in [3, 252]$ are recovery steps and the last three (*late*, *redo* and *down*) are special "fast" recovery steps. A period is an execution of a subset of steps, ran in order until one of them achieves a bundle for a specific value. A round always starts with a *proposal* step and finishes with a *cert* step (when a block is certified and commited to the ledger). However, multiple periods might be run inside a round until a certification bundle ($Bundle(r, p, s, v)$ where $s = cert$) is observable by the network.

Events are the only way in which the node state machine is able to both internally transition and produce output. In case an event is not identified as misconstrued or malicious in nature, it will certainly produce a state change, and it will almost certainly cause a receiving node to produce and then broadcast or relay an output, to be consumed by its peers in the network. There are two kinds of events: Timeout events, which are produced once the internal clock of a node reaches a certain time since the start of the current period; and Message events, which is output produced by nodes in response to some stimulus (including the receiving node itself). Internally, we consider the relevant data of an event to be:

- A floating point number representing time in seconds, from the start of the current period, in which the event has been triggered.
- An event type, from an enumeration of two options (either $TIMEOUT$ or $MESSAGE$)
- An attached data type, an enumeration of four options: $NONE$ (chosen in case of an event of main type $TIMEOUT$), $VOTE$, $PROPOSAL\_PAYLOAD$ and $BUNDLE$. It indicates the type of data attached.

- Attached data, plain bytes to be cast and interpreted according to the attached data type, or empty in case of a timeout event.

TimeoutEvents are events that are triggered after a certain time has elapsed after the start of a new period.

- $soft$ Timeout (aka. Filtering): The filter timeout is run after a timeout of $FilterTimeout(p)$ is observed (where p is the currently running period). Note that it only depends on the period as far as if it's the first period in a round or a subsequent one. Will perform a filtering action, finding the highest priority proposal observed to produce a soft vote, as detailed in the soft vote algorithm.

- $next_0$ Timeout: it's the first recovery step, only executed if no consensus for a specific value was observed, and no *cert* bundle is constructible with observed votes. It plays after observing a timeout of $\max\{4\lambda, \Lambda\}$. In it, the node will next vote a value and attempt to reach a consensus for a $next_0$ bundle, that would in turn kickstart a new period.

- $next_{st}$ Timeout: this family of timeouts runs whenever the elapsed time since the start of the period reaches $\max\{4\lambda, \Lambda\} + 2^{st-3}\lambda + r$, where $st[4, \in 252]$ and $r \in [0, 2^{st-3}\lambda]$, a random delta sampled uniformly at random that represents network variability. The algorithm run is exactly the same as in the $next_0$ step.

- Fast recovery Timeout (*late*, *redo* and *down* steps): On observing a timeout of $k\lambda_f + r$ with r a uniform random sample in $[0, \lambda_f]$ and k a positive integer, the fast recovery algorithm is executed. It works in a very similar way to $next_k$ timeouts, with some subtle differences (besides trigger time). For a detailed description refer to its own subsection.

MessageEvents Are events triggered after observing a certain message carrying data. There are 3 kinds of messages: votes, proposal payloads, and bundles, and each carry the corresponding construct (coinciding with their attached data type field).

4.1 Block Proposal

Algorithm 2 Block proposal

```
1:  function BlockProposal
2:      ResynchronizationAttempt()
3:      e ← AssembleBlock()
4:      for a ∈ A do
5:          cred ← Sortition(a_sk, proposal)
6:          if cred.j > 0 then
7:              if p = 0 ∨ ∃s'/Bundle(r, p − 1, s', ⊥) ⊂ V then
8:                  v ← Proposal_value(e)
9:                  Broadcast(Vote(I, r, p, proposal, v, cred))
10:                 Broadcast(e)
11:             else
12:                 v ← v_0/∃s, Bundle(r, p − 1, s, v_0) ⊂ V
13:                 Broadcast(Vote(I, r, p, proposal, v, cred))
14:                 if Proposal(v) ∈ P then
15:                     Broadcast(Proposal(v))
```

Description:
Block proposal is the first step, and the starting stage of the algorithm at every cycle. First, on line 2, the node attempts a resynchronization (described in 5). This only has any effect on periods $p > 0$. Afterwards, the node loops through all of its managed online accounts, Functionally "playing" for each of the accounts. This is a pattern that we'll see in every other main algorithm subroutine that performs any form of broadcasting. Any account that is selected as a possible proposer Should broadcast a vote and its corresponding proposal, where the Proposal value v inside the vote matches the proposal e, which is the block assembled by the node.

4.2 Soft Vote

Algorithm 3 Soft Vote

```
1:  function SoftVote
2:      lowestObservedHash ← ∞
3:      v ← ⊥
4:      for vote ∈ V|vote.step = proposal do
5:          priorityHash ← min_{i∈[0,vote.cred.j)}{H(vote.cred.sh||i)}
6:          if priorityHash < lowestObservedHash then
7:              lowestObservedHash ← priorityHash
8:              v ← vote.v
9:      if lowestObservedHash < ∞ then
10:         for Account a ∈ A do
11:             cred ← Sortition(a, soft)
12:             if cred.j > 0 then
13:                 Broadcast(Vote(r, p, soft, v, cred))
14:                 if Proposal(v) ∈ P then
15:                     Broadcast(Proposal(v))
```

Description:
The soft vote stage (also known as "filtering") is run after a timeout of $FilterTimeout(p)$ (where p is the node's currently executing period) is observed by the node. That is to say, filtering is triggered after either a X or X timeout is observed according to whether $p = 0$ or $p > 0$ respectively. Let $V*$ be all proposal votes received. By a priority hash function, this stage performas a filtering action, keeping the lowest hashed value observed. The priority function (line X) can be read as... Then, it proceeds to broadcast this value, running sortition for each managed online account. If the associated proposal, $Proposal(v)$, is in P, then it is subsequently broadcast too.

4.3 HandleProposal

Algorithm 4 HandleProposal

```
1:  function HandleProposal(Proposal e)
2:      v ← Proposal_value(e)
3:      if σ(r + 1, 0) = v then
4:          Relay(e)
5:          return //do not observe, as it's for a future round (we're behind)
6:      if ¬VerifyProposal(e) ∨ e ∈ P then
7:          return //ignore proposal
8:      if v ∉ {σ(r, p), v̄, μ(r, p)} then
9:          return //ignore proposal
10:     Relay(e)
11:     P ← P ∪ e
12:     if IsCommitable(v) ∧ s ≤ cert then
13:         for a ∈ A do
14:             cred ← Sortition(a_sk, cert)
15:             if cred.j > 0 then
16:                 Broadcast(Vote(a.I, r, p, cert, v, cred))
```

Description:
The proposal handler is triggered when a node receives a message containing a full proposal. It starts by performing a series of checks, after which it will either ignore the received proposal, discarding it and emitting no output; or relay, observe and produce an output according to the proposal's characteristics and the current context. The first check (lines 3:5) is a special case, where if the proposal is from the next round's first period, the node relays it and then ignores it for the purpose of the current round. Whenever the node catches up (observes a round change), and only if necessary, it will request this proposal back from the network. Lines 6:7 check if the proposal is invalid, or if it has been observed already. Any one of those conditions are sufficient to discard and ignore the proposal. Finally, on lines 8:9 it checks if the associated proposal value is either of the special values for the current round and period (σ, μ, or the pinned proposal value $\bar{v}$). Any proposal whose proposal value does not match one of these is ignored. Once the checks have been surpassed, the node is ready to relay the proposal and observe it by adding it to the observed proposals set, P (lines 10:11). Once relayed and observed, the proposal is then processed for further output. Here, and only if the proposal value has become commitable and the current executing node's step is lower or equal to a certification step (it's not yet in a recovery step), the node plays for each account performing sortition to select committee members for the certification step. For each selected member, a corresponding certify vote for the current proposal value is cast.

4.4 HandleVote

Algorithm 5 HandleVote

```
1: function HandleVote(Vote vt)
2:     if ¬VerifyVote(vt) then
3:         PenalizePeer(SENDER_PEER(vt))  //optional
4:         return  //ignore invalid vote
5:     if vt.step = 0 ∧ (vt ∈ V ∨ IsEquivocation(vt)) then
6:         return  //ignore vote, equivocation not allowed in proposal votes
7:     if vt.step > 0 ∧ IsSecondEquivocation(vt) then
8:         return  //ignore vote if it's a second equivocation
9:     if vt.round < r then
10:        return  //ignore vote of past round
11:    if vt.round = r + 1 ∧ (vt.period > 0 ∨ vt.step ∈ (cert, late)) then
12:        return  //ignore vote of next round if non-zero period or next_k step
13:    if vt.round = r ∧ (vt.period ∉ [p − 1, p + 1]∨
14: (vt.period = p + 1 ∧ vt.step ∈ (next_0, late))∨
15: (vt.period = p ∧ vt.step ∈ (next_0, late) ∧ vt.step ∉ [s − 1, s + 1])∨
16: (vt.period = p − 1 ∧ vt.step ∈ (next_0, late) ∧ vt.step ∉ [s̄ − 1, s̄ + 1])) then
17:        return  //ignore vote
18:    V ← V ∪ vt  //observe vote
19:    Relay(vt)
20:    if vt.step = proposal then
21:        if Proposal(vt.v) ∈ P then
22:            Broadcast(Proposal(vt.v))
23:    else if vt.step = soft then
24:        if ∃v|Bundle(vt.round, vt.period, soft, v) ⊂ V then
25:            for a ∈ A do
26:                cred ← Sortition(a_sk, cert)
27:                if cred.j > 0 then
28:                    Broadcast(Vote(a.I, r, p, cert, v, cred))
29:    else if vt.step = cert then
30:        if ∃v|Bundle(vt.round, vt.period, cert, v) ⊂ V then
31:            if Proposal(v) ∉ P then
32:                RequestProposal(v) //waits until proposal is obtained
33:            Commit(v)
34:            r_old ← r
35:            StartNewRound(vt.round + 1)
36:            GarbageCollect(r_old, p)
37:    else if vt.step > cert then
38:        if ∃v|Bundle(vt.round, vt.period, vt.step, v) ⊂ V then
39:            p_old ← p
40:            StartNewPeriod(vt.period)
41:            GarbageCollect(r, p_old)
```

Description:
The vote handler is triggered when a node receives a message containing a vote for a given proposal value, round, period and step. It first performs a series of checks, and if the received vote passes all of them, then it's broadcast by all accounts selected as the appropriate committee members. On line 2, it checks if the vote is valid by itself. If invalid, the node can optionally penalize the sender of the vote (by disconnecting or blacklisting it, for example). Equivocation votes on a

proposal step are not allowed, so a check for this condition is performed on line 5. Furthermore, second equivocations are never allowed (line 7). Any votes for rounds prior to the current round are discarded (line 9). On the special case that we received a vote for a round immediately after the current round, we observe it only if its a first period, proposal, soft, cert, late, down or redo vote (discarding votes for further periods or votes for a $next_k$ step). Finally, the checks on lines 13 to 16 check that, if the vote's round is the currently executing round, and one of:

- vote's period is not a distance of one or less away from the node's current period,
- vote's period is the next period, but its step is $next_k$ with $k \geq 1$,
- vote's period is the current period, its step is $next_k$ with $k \geq 1$, and its step is not within a distance of one away from the currently observed node's step, or
- vote's period is one behind the current period, its step is $next_k$ with $k \geq 1$, and its step is not within a distance of one away from the node's last finished step,

then the vote is ignored and discarded.
Once finished with the series of validation checks, the vote is observed on line 18, relayed on line 19, and then processed. The node will determine the desired output according to its current context and the vote's step. If the vote's step is *proposal*, the corresponding proposal for the proposal-value v, $Proposal(v)$ is broadcast if it has been observed (that is, the player performs a reproposal payload broadcast). If the vote's step is $soft$ (lines 19 through 24), and a $soft$ Bundle has been observed with the addition of the vote on line 19, the *Sortition* subprocedure is run for every account managed by the node (line 23). Afterwards, for each account selected by the lottery, a *cert* vote is cast as output (line 25). If the vote's step is *cert* (lines 26 through 32), and observing the vote causes the node to observe a *cert* Bundle for a proposal-value v, then it checks if the full proposal associated to the critical value has been observed (line 28). Note that simultaneous observation of a *cert* Bundle for a value v and of a proposal $e = Proposal(v)$ implies commitability of the associated entry. Had the full proposal not been observed at this point, the node may stall and request the full proposal from the network. Once the desired block is commitable, on lines 30:32 the node proceeds to commit, start a new round, and garbage collect all transient data from the round it just finished. Finally, if the vote is that of a recovery step (lines 33:36), and a Bundle has been observed for a given proposal-value v, then a new period is started and the currently executing period-specific data garbage collected.

4.5 HandleBundle

Algorithm 6 HandleBundle

```
1: function HandleBundle(Bundle b)
2:     if ¬VerifyBundle(b) then
3:         PenalizePeer(SENDER_PEER(b))  //optional
4:         return
5:     if b.r = r ∧ b.p + 1 ≥ p then
6:         for vt ∈ b do
7:             HandleVote(vt)
```

Description:
The bundle handler is invoked whenever a bundle message is received. If the received bundle is invalid (line 2), its immediately discarded. Optionally, the node may penalize the sending peer (for example, disconnecting from or blacklisting it). On line 5 a check is performed. If the bundle's round is the node's current round and it's at most one period behind of the node's current period, then the bundle is processed, which is simply calling the vote handler for each vote constituting it (lines 6:7). If the check on line 5 is not passed by b, no output is produced and the bundle is ignored

and discarded. Note that handling each vote separately, if a bundle $b\prime = Bundle(b.r, b.p, b.s, v\prime)$ is observed (where $v\prime$ is not necessarily equal to $\_.v$, consider b may contain equivocation votes), then it will be relayed as each vote was relayed individually, and any output or state changes it produces will be made. All leftover votes in b will be processed according to the new state the node is in, e.g. being discarded if $b.r < r$.

4.6 Recovery Attempt

Algorithm 7 Recovery

```
1:  function Recovery(uint64 k)
2:      ResynchronizationAttempt()
3:      s ← next_k
4:      for Account a ∈ A do
5:          cred ← Sortition(a_sk, next_k)
6:          if cred.j > 0 then
7:              if ∃v|IsCommitable(v) then
8:                  Broadcast(Vote(a.I, r, p, next_k, v, cred))
9:              else if ∄s_0 > cert|Bundle(r, p − 1, s_0, ⊥) ⊆ V ∧
                        ∃s_1 > cert|Bundle(r, p − 1, s_1, v̄) ⊆ V then
10:                 Broadcast(Vote(a.I, r, p, next_k, v̄, cred))
11:             else
12:                 Broadcast(Vote(a.I, r, p, next_k, ⊥, cred))
```

Description:
The recovery algorithm that is executed periodically every X seconds, whenever a *cert* bundle has not been observed before FilterTimeout(p) for a given period p.
On line 2 it starts by making a resynchronization attempt. Then on line 3 the node's step is updated.
Afterwards, the node plays for each managed account. For each account that is selected to be a part of the voting committee for the current step $next_k$, one of three different outputs is produced.
If there is a proposal-value v that is commitable in the current context, a $next_k$ vote for v is broadcast by the player.
If no proposal-value is commitable, no recovery step Bundle for the empty proposal-value ($\perp$) was observed in the previous period, and a recovery step Bundle for the pinned value was observed in the previous period (note that this implies $\bar{v} \neq \perp$), then a $next_k$ vote for $\bar{v}$ is broadcast by the player.
Finally, if none of the above conditions were met, a $next_k$ vote for $\perp$ is broadcast. A player is forbidden from equivocating in *next* votes.

4.7 Fast Recovery Attempt

Algorithm 8 FastRecovery

```
1: function FastRecovery
2:     ResynchronizationAttempt()
3:     for Account a ∈ A do
4:         if IsCommitable(v) then
5:             cred ← Sortition(a_sk, late)
6:             if cred.j > 0 then
7:                 Broadcast(Vote(a.I, r, p, late, v, cred))
8:         else if ∄s_0 > cert|Bundle(r, p − 1, s_0, ⊥) ⊆ V ∧
                       ∃s_1 > cert|Bundle(r, p − 1, s_1, v̄) ⊆ V then
9:             cred ← Sortition(a_sk, redo)
10:            if cred.j > 0 then
11:                Broadcast(Vote(r, p, redo, v̄, cred))
12:        else
13:            cred ← Sortition(a_sk, down)
14:            if cred.j > 0 then
15:                Broadcast(Vote(r, p, down, ⊥, cred))
```

Description:
The fast recovery algorithm is executed periodically every integer multiple of λ_f seconds (plus variance). Functionally, it's very close to the regular recovery algorithm (outlined in the previous section), performing the same checks and similar outputs. The sole difference is that it emmits votes for any of three different steps (*late*, *redo* and *down*) according to sortition results for every account. It's also important to point out that nodes are forbidden to equivocate for *late*, *redo* and *down* votes.

5 Subroutines

Algorithm 9 FilterTimeout

```
1: function FilterTimeout(uint64 p)
2:     if p = 0 then
3:         return 2λ_0
4:     else
5:         return 2λ
```

Arguments:

- *uint*64 p is a period number

Description:
Provides the timeout constant for the filtering (a.k.a. "soft" or "soft vote") stage. This timeout depends on the period value; the first period has a special, faster timeout. If no consensus was achieved, this timeout constant is relaxed in all subsequent periods.

Returns:

- The time constant used to trigger the filtering stage (according to whether the given period p is the first period of the current round or not)

Algorithm 10 General Purpose Hashing Function

```
1: function H(Hashable in)
2:     if SHA512/256 is supported then
3:         returnSHA512/256(in)
4:     else
5:         returnSHA256(in)
```

Arguments:

- $Hashable\ in$ = some hashable data (plain bytes)

Description:
General purpose hashing function. If $SHA512/256$ is supported by the underlying system running the node, it's used. Otherwise falls back to $SHA256$.

Returns:

- The result of hashing the input in with the selected algorithm based on underlying system support

Algorithm 11 AssembleBlock

```
1: function AssembleBlock
2:     Blockb
3:     b.body ← TP[−s]
```

Description:
Gets a set of transactions out of the transaction pool (prioritizing highest transaction fee if there were any). Then, assembles a full ledger entry, setting all of the appropriate fields in a block.

Algorithm 12 IsCommitable

```
1: function IsCommitable(Proposal_value v)
2:     return Proposal(v) ∈ P ∧ Bundle(r, p, soft, v) ⊂ V
```

Arguments:

- $Proposal_{value}\ v$, a value to check for commitability

Description:
Checks that the value v is commitable in the current node's context. To be commitable, the two conditions outlined in line 2 have to be met. That is, the corresponding proposal that the value refers to has to be available (have been observed in the current round), and there must be a valid bundle of soft votes for v observed during the current round and period.

Returns:

- A boolean value indicating commitability of the argument v

Algorithm 13 Commit

```
1: function Commit(Proposal_value v)
2:     L ← L||Proposal(v)
3:     UpdateBT(Proposal(v))
```

Arguments:

- $Proposal_{value}$ v, a proposal-value to be commited

Description:
Commits the corresponding proposal for the value passed by parameter into the ledger. As a precondition, the value is commitable (which implies validity and availability of the full ledger entry and seed). Afterwards, updates the balance table with all state changes called for in the commited entry.

Algorithm 14 VerifyVote

```
1:  function VerifyVote(Vote vt)
2:      valid ← vt.round ≤ r + 2
3:      if vt.step = 0 then
4:          valid ← valid ∧ vt.v.p_orig ≤ vt.period
5:          if vt.period = vt.v.p_orig then
6:              valid ← valid ∧ vt.v.I_orig = vt.I
7:      if vt.step ∈ {propose, soft, cert, late, redo} then
8:          valid ← valid ∧ vt.v ≠ ⊥
9:      else if vt.step = down then
10:         valid ← valid ∧ vt.v = ⊥
          //TODO: verificacion de firmas, VRFs y rounds de validez
11:     return valid
```

Description:

Algorithm 15 Proposal

```
1:  function Proposal(Proposal_value v)
2:      if ∃e ∈ P|Proposal_value(e) = v then
```

Description:
Gets the proposal associated to a given value, if it has been observed. Otherwise returns undefined. Here we ignore the theoretical posibility of a collision

Algorithm 16 Proposal

```
1:  function Proposal_value(Proposal e)
        //pack and hash
2:      proposal_value v ← (I, p, Digest(e), Hash(Encoding(e)))
3:      return v
```

Description:

Algorithm 17 VerifyProposal

```
1:  function VerifyProposal(Proposal pp)
2:      valid ← ValidEntry(pp.e, L)
3:      valid ← valid ∧ VerifySignature(pp.y)
```

Description:

Algorithm 18 VerifyBundle

1: **function** VerifyBundle($Bundle\ b$)
  //all individual votes are valid
2: $valid \leftarrow (\forall vt \in b)(VerifyVote(vt))$
  //no two votes are the same
3: $valid \leftarrow valid \wedge (\forall i, k \in \mathbb{Z})(vt_i \in b \wedge vt_k \in b \wedge vt_i = vt_k \implies i = k)$
  //round, period and step must all match
4: $valid \leftarrow valid \wedge (\forall vt \in b)(vt.r = b.r \wedge vt.p = b.p \wedge vt.s = b.s)$
  //all votes should either be for the same value or be equivocation votes
5: $valid \leftarrow valid \wedge (\forall vt \in b)(vt.v = b.v \vee (b.s = soft \wedge IsEquivocation(vt.v, b)))$
  //summation of weights should surpass the relevant threshold
6: $valid \leftarrow valid \wedge \sum_{vt \in b}(vt.w) \geq CommitteeThreshold(b.s)$
7: return $valid$

Description:

Algorithm 19 StartNewRound

1: **function** StartNewRound($uint64\ newRound$)
2: $\bar{s} \leftarrow s$
3: $\bar{v} \leftarrow \perp$
4: $r \leftarrow newRound$
5: $p \leftarrow 0$
6: $s \leftarrow proposal$

Description:
Procedure used to set all state variables necessary to start a new round. Last finished step is set to the step where the previous round culminated. Pinned proposal-value is set to the empty proposal-value as the round is just starting. The current round number gets updated to the freshly started round. Period and step number are set to 0 and *proposal* respectively.

Algorithm 20 StartNewPeriod

1: **function** StartNewPeriod($uint64\ newPeriod$)
2: $\bar{s} \leftarrow s$
3: $s \leftarrow proposal$
4: **if** $\exists v, s' \mid v \neq \perp \wedge (s' = soft \vee s' > cert) \wedge Bundle(r, newPeriod - 1, s, v) \subset V$ **then**
5: $\bar{v} \leftarrow v$
6: **else if** $\sigma(r, p) \neq \perp$ **then**
7: $\bar{v} \leftarrow \sigma(r, p)$
8: $p \leftarrow newPeriod$

Description:
Procedure used to set all state variables necessary to start a new period. Note that we start a new period on observing a recovery bundle for a proposal-value, whether it be an actual value or the special empty value $\perp$. On lines 2 and 3, the node sets the last finished step to the currently executing step when a new period's start was observed, and the current step to *proposal*. Then it checks for the existance of a non-*cert* step bundle in the period immediately before the new one (line 4), for a proposal-value that's anything but the empty value $\perp$ (note that if the node had observed a *cert* bundle in the previous period, it would not be starting a new period and it would be instead attempting to commit the relevant entry and subsequently start a new round). If such bundle for a proposal-value v exists, the pinned value is updated to v. Otherwise, and assuming implicitly in this case that the bundle that caused the period switch is of value $\perp$, a check for the

special $sigma(r, p)$ value is performed on line 6, where p is the period that was being executed by the node up until a new period was observed. If $sigma(r, p)$ is a valid non-empty proposal-value, the pinned value $\bar{v}$ is set to this (line 7). Finally, if none of the above conditions were met, the pinned value remains unchanged going into the new period. Finally, we update p to match the period to start.

Algorithm 21 GarbageCollect

1: **function** GarbageCollect
2: $V_{(r,p-1)} \leftarrow \{vt \in V | vt.r < r \vee (vt.r = r \wedge vt.p + 1 < p)\}$
3: $P_{(r,p-1)} \leftarrow \{pp \in P | pp.r < r \vee (pp.r = r \wedge pp.p + 1 < p)\}$
4: $V \leftarrow V \setminus V_{(r,p-1)}$
5: $P \leftarrow P \setminus P_{(r,p-1)}$

Description:
Garbage collection algorithm for the freshly started period or round. The procedure discards all votes in V and proposals in P where the round of emmision is less than the new round, or the round of emmision is equal to the new round and the period of emmision is below the period directly before the current one.

Algorithm 22 RequestProposal

1: **function** RequestProposal($Proposal_{value}$ v) //ver como hacer el request...como lo resuelve el nodo?

Description:

Algorithm 23 PenalizePeer

1: **function** PenalizePeer($PEER\_NETWORK\_ID$) //ver como resuelve el nodo la desconexion o blacklist de un peer

Description:

Algorithm 24 ResynchronizationAttempt

1: **function** ResynchronizationAttempt
2: $Val = \perp$
3: **if** $\exists v | Bundle(r, p, soft, v) \subset V$ **then**
4: Broadcast($Bundle(r, p, soft, v)$)
5: $val = v$
6: **else if** $\exists s_0 > cert | Bundle(r, p-1, s_0, \perp) \subset V$ **then**
7: Broadcast($Bundle(r, p-1, s_0, \perp)$)
8: **else if** $\exists s_0, v | s_0 > cert \wedge v \neq \perp \wedge Bundle(r, p-1, s_0, v) \subset V$ **then**
9: Broadcast($Bundle(r, p, s_0, v)$)
10: $val = v$
11: **if** $val \neq \perp \wedge Proposal(v) \in P$ **then**
12: Broadcast($Proposal(v)$)

Description:
A resynchronization attempt, performed at the start of all recovery algorithms. If a soft bundle has been observed for a proposal-value v, then the bundle is broadcast. Otherwise, if a recovery step bundle for an empty proposal-value $\perp$ was observed in the previous period, It's broadcast. Else,

if there is a recovery step s_0 and a non-empty proposal-value v for which a bundle was observed in the previous period, it's broadcast. Finally, if any Bundles were broadcast for a proposal-value v, the corresponding proposal $Proposal(v)$ is broadcast if it has been observed.

Algorithm 25 ComputeSeed

```
1: function ComputeSeed(b)
2:     if B ≠ empty_block then
3:         return VRF_{get_{SK_a}(ctx,r)}(ctx.LastBlock.seed||r)
4:     else
5:         return Hash(ctx.LastBlock.seed||r)
```

Arguments:

- b = the block whose seed is being computed

Description:
This subroutine computes the required sortition seed for the given round number, which goes in the proposed block's metadata. If the block is empty, the seed is a hash of the previous block's seed. The $get_{SK_a}(ctx, r)$ helper function gets the relevant account's secret ephemeral keys (according to the signing scheme described in specs, the keys 160 rounds prior to r). This roughly corresponds to the secret key from a round b time before block $r - 1 - (r \ mod \ R)$, where R is the sortition seed's renewal rate, r is the current round's number, and b is the upper bound for the maximum ammount of time that the network might be compromised.

Returns:

- the computed seed for the given block, ledger context and round

Algorithm 26 getSortitionSeed

```
1: function getSortitionSeed(ctx, r, a_pk)
       return ctx.block[r − 1 − (r mod R)].seed
```

Arguments:

- ctx = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- $round$ = current round number
- a_{pk} = the account's public key for the look up table

Description:
This helper function gets the relevant sortition seed for the current round r, according to the seed lookback parameter R. Conceptually, it corresponds with the seed computed R rounds prior to r, refreshed every R rounds.
Returns:

- a sortition seed to be used in the round r

Algorithm 27 getSortitionWeight

```
1: function getSortition_w(ctx, round, a_pk)
       return ctx.balanceTable[r − (R + SL)][a_pk]
```

Arguments:

- ctx = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- r = current round number
- a_{pk} = the account's public key for the look up table

Description:
This helper function retrieves the stake $R + SL$ rounds prior to r, for an account with public key a_{pk}
Returns:

- the relevant account's stake

Algorithm 28 getSortitionTotalStake

function $getSortition_W(ctx, r)$
    return $\sum_{a_{pk} \in ctx.balanceTable[r-(R+SL)]} balanceTable[r-(R+SL)][a_{pk}]$

Arguments:

- ctx = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- r = current round number

Description:
This helper function returns the sum of all stake for $R + SL$ rounds prior to r.
Returns:

- the total stake at play in the relevant round (according to lookback parameters)

Algorithm 29 Sortition

function Sortition($Account\ a, uint64\ step$)
    $\langle hash, \pi \rangle \leftarrow \mathsf{VRF}(seed||step)$
    $p \leftarrow \frac{t}{W}$
    $j \leftarrow 0$
    while $\frac{hash}{2^{hashlen}} \notin [\sum_{k=0}^{j} \mathsf{B}(k; w, p), \sum_{k=0}^{j+1} \mathsf{B}(k; w, p))$ **do**
        $j \leftarrow j + 1$
    return $\langle hash, \pi, j \rangle$

Arguments:

- sk = a user's secret key (an ephemeral key for the given round, according to key specs)
- $seed$ = the sortition seed to be used
- τ = the expected committee size for the given role
- $role$ = a flag specifying the role for the sortition to take place (e.g. block proposer)
- w = the user's weight (i.e., its relevant stake)
- W = the total relevant stake for the given round

Description:
The Sortition procedure is one of the most important subroutines in the main algorithm, as it is used in multiple stages and contexts. Generally, it manages to pseudo-randomly but verifiably (through the use of a Verifiable Random Function) select a user with probability proportional to

their weight (i.e., stake) by returning a j parameter, which indicates the number of times that specific user was chosen. Algorithmically, every monetary unit the user has is considered a "sub-user", and then each one of them is selected with probability $p = \frac{\tau}{W}$, where τ is the expected amount of users to be selected for the given role. The semi-open interval $[0, 1)$ is then split into consecutive intervals using an accumulated binomial distribution, and wherever the fraction $\frac{hash}{2^{hashlen}}$ belongs to the interval, that's the ammount of chosen sub-users for the subroutine caller.

Returns:

- an object of type *credentials*, containing

Algorithm 30 VerifySortition

```
1: function VerifySortition(pk, seed, τ, role, w, W)
2:     if ¬VerifyVRF_pk(hash, π, seed||role) then return 0
3:     p ← t/W
4:     j ← 0
5:     while hash/2^hashlen ∉ [Σ_{k=0}^{j} B(k; w, p), Σ_{k=0}^{j+1} B(k; w, p)) do
6:         j++
       return j
```

Arguments:

- pk = a user's public key (their address)
- $seed$ = the sortition seed to be used
- τ = the expected committee size for the given role
- $role$ = a flag specifying the role for the sortition to take place (e.g. block proposer)
- w = the user's weight (i.e., its relevant stake)
- W = the total relevant stake for the given round

Description:
The sortition verification procedure takes Sortition's output and utilizes VRF properties to verify the validity of said output. Once the check is passed, it repeats Sortition's sub-user selection procedure, and outputs the amount of times the user was chosen according to their respective observed stake.

Returns:

- an integer j that will be positive (larger than 0) if the user has been selected, and it's size corresponds to the amount of sub-users for a given committee member

6 Appendix: Notation

- *credentials*, a data structure containing the results of running the sortition algorithm for a specified account
- sh, the 64 byte sortition hash output by running $VRF_{a_{sk}}$ algorithm over a desired input. Usually wrapped inside a *credentials* structure.
- π, the 32 byte sortition proof output by running $VRF_{a_{sk}}$ algorithm over a desired input. Usually wrapped inside a *credentials* structure.

- j, an unsigned 64 bit integer, representing the weight of a given account's vote inside a specific committee (for a given round, period and step). Usually wrapped inside a *credentials* structure.
- λ, time interval for the node to accept block proposals, after which it chooses the observed block with the highest priority (lowest hash).

References

[GHM+17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017.

[Mic16] Silvio Micali. ALGORAND: the efficient and democratic ledger. *CoRR*, abs/1607.01341, 2016.