

Algorand Protocol Description

Argimiro, CoinFabrik

February 22, 2023

Contents

1	Introduction	1
2	Node as a State Machine	2
3	Main algorithm	2
3.1	Block Proposal	3
3.2	Soft Vote	4
3.3	HandleProposal	4
3.4	HandleVote	4
3.5	HandleBundle	5
3.6	Recovery Attempt	5
3.7	Fast Recovery Attempt	6
4	Subroutines	6

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 associated 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%).

A network is formed with two kind of nodes: relay and participation nodes. Participation nodes can connect only to relay nodes. They listen to one or more participation nodes and they may send a message to a relay node. Relay nodes simply listen to participation nodes connected to them and relay the messages they receive.

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

Explain how
the network
is formed

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

2 Node as a State Machine

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 aliases for step number:

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

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 P and V of objects which we will call observed *proposal* and *vote* values, a Ledger L and a balance table BT , a table of public addresses for all accounts holding any stake in the network, and their balances. A Ledger is an ordered chain of blocks...

3 Main algorithm

Algorithm 1 Main node algorithm

```

1: function EventHandler(Event  $ev$ )
2:   if  $ev$  is TimeoutEvent then
3:      $time \leftarrow ev.time$ 
4:     if  $time = 0$  then
5:       BlockProposal()
6:     else if  $time = \text{FilterTimeout}(p)$  then
7:       SoftVote()
8:     else if  $time = \max\{4\lambda, A\} \vee time = \max\{4\lambda, A\} + 2^{st-3}\lambda + r, st \in [4, 252], r \in [0, 2^{st-3}\lambda]$ 
       then
9:       Recovery()
10:    else if  $time = k\lambda_f + r, k \in \mathbb{Z}, r \in [0, \lambda_f]$  then
11:      FastRecovery()
12:    else
13:       $msg \leftarrow ev.msg$ 
14:      if  $msg$  is Proposal  $p$  then
15:        HandleProposal( $p$ )
16:      else if  $msg$  is Vote  $v$  then
17:        HandleVote( $v$ )
18:      else if  $msg$  is Bundle  $b$  then
19:        HandleBundle( $b$ )

```

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 (could include the receiving node itself).

Furthermore, given a proposal value, we denote $\text{Proposal}(v) = e$ such that if v is a valid proposal value (validated according to context).

3.1 Block Proposal

Algorithm 2 Block proposal

```

1: function BlockProposal
2:   for  $a \in A$  do
3:      $\langle \text{sorthash}, \pi, j \rangle \leftarrow \text{Sortition}(a_{sk}, \text{getSortitionSeed}(ctx, r), t, \text{proposal},$ 
                                      $\text{getSortition}_w(ctx, r, a_{pk}), \text{getSortition}_W(ctx, r))$ 
4:     if  $j > 0$  then
5:       if  $p = 0 \vee \exists s' / \text{Bundle}(r, p - 1, s, \perp) \subset V$  then
6:          $e \leftarrow \text{AssembleBlock}()$ 
7:          $\text{priority} \leftarrow \min_{1 \leq n < j} \{\text{Hash}(\text{sorthash} || n)\}$ 
8:          $v \leftarrow \text{Proposal}_{\text{value}}(e)$ 
9:          $\text{Broadcast}(\text{Vote}(I, r, p, \text{proposal}, v))$ 
10:         $\text{Broadcast}(e)$ 
11:      else
12:         $v \leftarrow v_0 | \exists s, \text{Bundle}(r, p - 1, s, v_0) \subset V$ 
13:         $\text{Broadcast}(\text{Vote}(I, r, p, \text{proposal}, v))$ 
14:        if  $\text{Proposal}(v) \in P$  then
15:           $\text{Broadcast}(\text{Proposal}(v))$ 

```

Description:

The node loops through all of its managed accounts. Functionally, the node "plays" for every account it manages, as each online account represents a different player in the system.

3.2 Soft Vote**Algorithm 3** Soft Vote

```

1: function SoftVote
2:   lowestObservedHash  $\leftarrow \infty$ 
3:   v  $\leftarrow \perp$ 
4:   for vote  $\in V$ , vote.step = proposal do
5:     priorityHash  $\leftarrow \min_{i \in [0, j)} H(\text{vote.priority} || i)$ 
6:     if priorityHash < lowestObservedHash then
7:       lowestObservedHash  $\leftarrow \text{priorityHash}$ 
8:       v  $\leftarrow \text{vote.v}$ 
9:   for a  $\in \text{Accounts}$  do
10:     $\langle j, \text{hash}, \pi \rangle \leftarrow \text{sortition}(a, \text{soft})$ 
11:    if  $j > 0 \wedge \text{lowestObservedHash} < \infty$  then
12:      GOSSIP(Vote(r, p, soft, v))
13:      if Proposal(v)  $\in P$  then
14:        GOSSIP(Proposal(v))

```

Description:

The soft vote stage (also known as "filtering")...

3.3 HandleProposal**Algorithm 4** HandleProposal

```

1: function HandleProposal(Proposale)
2:   v  $\leftarrow \text{Proposal}_{\text{value}}(e)$ 
3:   if IsCommittable(v)  $\wedge s \leq \text{cert}$  then
4:     for a  $\in A$  do
5:        $\langle \text{sorthash}, \pi, j \rangle \leftarrow \text{Sortition}(a_{sk}, \text{getSortitionSeed}(\text{ctx}, r), t, \text{cert},$ 
6:                                      $\text{getSortition}_w(\text{ctx}, r, a_{pk}), \text{getSortition}_W(\text{ctx}, r))$ 
7:       if  $j > 0$  then
8:         Broadcast(Vote(a.I, r, p, cert, v))

```

Description:**3.4 HandleVote****Algorithm 5** HandleVote

```

1: function HandleVote(Votev)
2:   if notVerifyVote(v) then
3:     if then

```

Description:

3.5 HandleBundle

Algorithm 6 HandleBundle

```

1: function HandleBundle(Bundleb)
2:   if VerifyBundle(b)  $\wedge$  b.r = r  $\wedge$  b.p  $\geq$  p - 1 then
3:     for vote  $\in$  b do
4:       HandleVote(vote)

```

Description:

3.6 Recovery Attempt

Algorithm 7 Recovery

```

1: function Recovery
2:   s  $\leftarrow$  nexts
3:   ResynchronizationAttempt()
4:   for Accounta  $\in$  A do
5:      $\langle$ sorthash,  $\pi$ , j $\rangle \leftarrow$  Sortition(ask, getSortitionSeed(ctx, r), t, nexts,
                                     getSortitionw(ctx, r, apk), getSortitionW(ctx, r))
6:     if j > 0 then
7:       if  $\exists v | \text{IsCommitable}(v)$  then
8:         GOSSIP(Vote(I, r, p, nexts, v))
9:       else if  $\nexists s_0 > \text{cert} | \text{Bundle}(r, p - 1, s_0, \perp) \subseteq V \wedge$ 
               $\exists s_1 > \text{cert} | \text{Bundle}(r, p - 1, s_1, \bar{v}) \subseteq V$  then
10:        GOSSIP(Vote(I, r, p, nexts,  $\bar{v}$ ))
11:       else
12:        GOSSIP(Vote(I, r, p, nexts,  $\perp$ ))

```

Description:

3.7 Fast Recovery Attempt

Algorithm 8 *FastRecovery*

```

1: function FastRecovery
2:   ResynchronizationAttempt()
3:   for  $Accounta \in A$  do
4:     if IsCommitable( $v$ ) then
5:        $\langle sorthash, \pi, j \rangle \leftarrow \text{Sortition}(a_{sk}, \text{getSortitionSeed}(ctx, r), t, 'late',$ 
                                      $\text{getSortition}_w(ctx, r, a_{pk}), \text{getSortition}_W(ctx, r))$ 
6:       if  $j > 0$  then
7:         GOSSIP(Vote( $I, r, p, late, v$ ))
8:       else if  $\nexists s_0 > cert | \text{Bundle}(r, p-1, s_0, \perp) \subseteq V \wedge$ 
                $\exists s_1 > cert | \text{Bundle}(r, p-1, s_1, \bar{v}) \subseteq V$  then
9:          $\langle sorthash, \pi, j \rangle \leftarrow \text{Sortition}(a_{sk}, \text{getSortitionSeed}(ctx, r), t, 'redo',$ 
                                      $\text{getSortition}_w(ctx, r, a_{pk}), \text{getSortition}_W(ctx, r))$ 
10:        if  $j > 0$  then
11:          GOSSIP()
12:       else

```

Description:

The fast recovery algorithm is executed periodically every integer multiple of λ_F seconds. In it, nodes of the network make a synchronization attempt. In the first section, if there is a value v

4 Subroutines

Algorithm 9 *IsCommitable*

```

1: function IsCommitable(Proposalvalue $v$ )
2:   return Proposal( $v$ )  $\in P \wedge \text{Bundle}(r, p, soft, v) \subset V$ 

```

Algorithm 10 *ResynchronizationAttempt*

```

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

```

Description:

Algorithm 11 ComputeSeed

```
1: function ComputeSeed(ctx, r, B)
2:   if B  $\neq$  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:

- *ctx* = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- *r* = current round number
- *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 $r - 1 - (r \bmod R)$, 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 12 getSortitionSeed

```
1: function getSortitionSeed(ctx, r, apk)
   return ctx.block[ $r - 1 - (r \bmod R)$ ].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 13 getSortitionWeight

```
1: function getSortitionw(ctx, round, apk)
   return ctx.balanceTable[ $r - (R + SL)$ ][apk]
```

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 14 getSortitionTotalStake

```
1: function getSortitionTotalStake( $ctx, r$ )
   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 15 Sortition

```
1: function Sortition( $sk, seed, \tau, role, w, W$ )
2:    $\langle hash, \pi \rangle \leftarrow \text{VRF}(seed || role)$ 
3:    $p \leftarrow \frac{t}{W}$ 
4:    $j \leftarrow 0$ 
5:   while  $\frac{hash}{2^{hashlen}} \notin [\sum_{k=0}^j \mathbf{B}(k; w, p), \sum_{k=0}^{j+1} \mathbf{B}(k; w, p)]$  do
6:      $j++$ 
   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 integer j that will be positive (larger than 0) if the user has been selected, and its size corresponds to the amount of sub-users for a given committee member

Algorithm 16 VerifySortition

```

1: function VerifySortition( $pk, seed, \tau, role, w, W$ )
2:   if  $\neg \text{VerifyVRF}_{pk}(hash, \pi, seed || role)$  then return 0
3:    $p \leftarrow \frac{\tau}{W}$ 
4:    $j \leftarrow 0$ 
5:   while  $\frac{hash}{2^{hashlen}} \notin [\sum_{k=0}^j B(k; w, p), \sum_{k=0}^{j+1} B(k; w, p))$  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

Appendix: Notation

- R = sortition seed renewal rate (in number of rounds). Set to 2 in the specs. as of December 2022
- MAX_STEPS = maximum number of allowed steps in the main algorithm. Defined as 255 in the specs.
- τ_{step} = expected number of members on a regular step committee

- τ_{final} = expected number of members on a final consensus achieving committee
- T_{step} = fraction of expected members for a voting committee on a given step
- T_{final} = fraction of expected members for a voting committee on the final step (step = 2).
If observed, achieves final consensus on the given block hash
- $\lambda_{proposal}$ = time interval for the node to accept block proposals, after which it chooses the observed block with the highest priority (lowest hash)
- λ_{block} = waiting time for the full block to be received once decided by vote. If no full block is received, the node falls back to the empty block.
- SL = the account balances lookback interval, in number of rounds (integer). Set to 320 in specs.
- Public keys a_{pk} for every account linked to the node (doubles as their respective addresses)
- Public participation keys $a_{p-partkey}$ for all accounts registered as online (private participation keys are kept securely by respective users and should not be directly accesible by the node)
- Two levels of ephemeral keys, signed according to the signing scheme described in specs. (where first level keys are signed with the participation keys, and second level (aka. leaf) ephemeral keys are signed with first level ephemeral keys). These keys are used for actual participation in the consensus (abstracted as a_{sk} in the following code) They are ephemeral because they live for a single round, after which they are deleted
- A balance table *BalanceTable* for all accounts linked to the node, for the round $SL + R$ rounds before the one currently running

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.