Algorand Protocol Description

Argimiro del Pozzo Ariel Waissbein

CoinFabrik
February 21, 2024

Contents

Abstract

Algorand is a pure proof-of-stake blockchain protocol, designed by Silvio Micali, that has been implemented and deployed since 2019. Micali and co-authors designed the basic blocks

of the protocol and implementors published some design details. None of these cover the full protocol and many details are left undefined, moreover, many protocol decisions have changed from design to implementation. The security implications of this design should be studied. In our attempt to provide such a study we have analyzed the source code for the node and probed the blockchain for answers. To our account this is the first and only complete description of the protocol.

1 Introduction

We aim to describe how the Algorand protocol works. This document is meant to reflect how the Algorand mainnet is behaving today. To our knowledge, there is no accurate and up-to-date technical description of the protocol.

While we are aquainted with Algorand's seminal papers and documentation ([GHM+17], [Mic16] and Algorand's official specifications) this document is based on the code running in Algorand's mainnet, and the reference implementation maintained by Algorand inc. To this end, we relied in the sourcecode for the node in the reference implementation and probed the network when information was unclear or unavailable. At the time of this writing the protocol is at `ConsensusV38` (protocol version 38) which is specified in the file `go-algorand/protocol/consensus.go` by the constant `ConsensusCurrentVersion`.

Algorand is a proof-of-stake blockchain cryptocurrency protocol using a decentralized Byzantine Agreement protocol that leverages pure proof of stake for consensus. Protocol parties are represented by accounts, which may represent users or smart contracts.

The Algorand blockchain is operated by nodes. Users register their accounts in nodes. It is the nodes which call the actions. When an account sends a transaction, it sends it to a node.

The structure of a node is simple: it has a consensus algorithm, a virtual machine (the Algorand Virtual Machine or AVM) and networking capabilities.

Basically, the node receives transactions and puts them in a list called the transaction pool, validates their signature and processes them (in no protocol-imposed order), leveraging the AVM if there was bytecode associated to them, with the goal of an eventual commitment of the transactions to a block in the Ledger and the subsequent observation of their consequences in the state of the network.

The node maintains a ledger, which encodes the protocol status, that is modified via consensus. In particular, this ledger encodes how many ALGO tokens (the native token) holds each account.

Blocks encode status updates in 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. Mainnet runs one blockchain that is univocally identified by the ID of the genesis block (i.e. a hash value).

Algorand blockchain is designed so that the voting power is proportional to the stake (in ALGO) held by the accounts associated to the node. In practice, not all accounts are registered live, so the voting power is proportional to the tokens held by (the subset) of online accounts. At the genesis of the Algorand blockchain, 10Bn ALGO were minted. As of September 2022, circulating supply is approximately 6.9b ALGO, distributed through different forms as ecosystem support and community incentives. The remaining ALGO tokens 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 x can be in, as well as all kinds of events that cause transitions and network output. This work will provide a basis for an Algorand blockchain simulator currently in development.

2 Preliminaires

The Algorand protocol is implemented in a network over the Internet, where nodes implementing a service form a peer-to-peer (p2p) network: they listen to other nodes and to transactions sent by clients, generate blocks, vote and continue to write and maintain the blockchain.

2.1 Nodes

Actually, Algorand protocol establishes two kinds of nodes: **relay nodes** and **participation nodes**.

Relay nodes are responsible for relaying messages coming from other relay nodes and the messages they receive from participation nodes. A relay node will queue (multiplex) both kinds of messages. Relay nodes also keep a complete copy of the ledger, updating it on block confirmation, and run a service answering to participation queries about the ledger. They are permissioned: they are *whitelisted* by Algorand Inc. and this whitelist is hardcoded into the source code of both kinds of nodes. Relay nodes do not participate in consensus. We shall not go in further detail about relay nodes.

Participation nodes receive transactions from clients, generate and vote for blocks, and when these blocks are accepted, they record the changes. Each participation node may connect to one or more relay nodes: their addresses and keys are hardcoded into the code of participation nodes (a list of the public key's hashes is included in the code).

A participation node runs the following services.

- A timer which may receive a reset() order or a time() order. It will record the current time when reset is called (e.g., by querying the OS where the node runs) and will report the elapsed time since the last reset in the latter case.
- A transaction pool service (described below).
- An Algorand Virtual Machine (AVM for short), which can process bytecode from applications or attached to smart signatures upon receiving transactions (see 2.6 for more details).
- A p2p and a websocket network interface.
- Ledger and balances (see Sect. **??**).
- An implementation of the consensus algorithm, with a player state machine module and an asynchronous cryptographic computation module. This is explained in Section **??** in greater detail.

Participation nodes are non-archival in nature, meaning they do not keep the whole ledger in memory at all times. At the time of this writing, participation nodes keep data for the latest 1,000 blocks. Any queries that require nonavailable data are made through the relay node they are connected to.

2.1.1 The Transaction Pool

The node actually implements two queues for the transaction pool service. At the start of a round, the node receives unconfirmed transactions (see Sect. 2.4) from relay nodes or via the node's websocket from accounts managed by this node and sends them to a priority queue. The queue size is limited, e.g, the node will keep the first 1,000 transactions it receives and ignore the rest[1]

[1]Actually, if this limit is reached, then the node will prioritize transactions according to the fees they pay and may thus allow for newer transactions and drop some of the older ones. Note that this is how the reference

As transactions are put into this priority queue, they are checked, and they are either rejected or enqueued into the transaction pool. The check includes validating if the distance between the round the node is in and the round included in the transaction are not further than a threshold, verifying their signature (if the transaction is signed by a single account), asking the AVM to verify a smart signature/logicsig if that is the case, and if this were an application call, then asking the AVM to simulate the transaction. Any transaction not passing this check is marked as invalidated.

The transaction pool is now ready to be called by the block assembly procedure. It supports the following functions. a $\mathsf{head}(\cdot)$ function to get the next element to unqueue, an $\mathsf{iterator}(\cdot)$ helper function which returns an iterator into the queue, and is able to retrieve all queued elements in priority order. It also supports an $\mathsf{UpdateTP}(TP, e)$ procedure, which takes a valid payset (i.e., the list of all transactions) from a block e and correctly unqueues all transactions included in the intersection $e_{payset} \cap TP$, and then drops all elements that have been invalidated by round advancement (e.g., transactions have a validity range).

2.2 Network

A network is formed of two kinds of nodes: relay nodes and participation nodes. Both nodes operate on servers that are connected to the internet and run publicly available code from Algorand's code repository.

When a participation node boots, it establishes a peer-to-peer (P2P) channel with at least one relay (4 on average), using a phone book which is included as part of the source code of the node. Relay nodes manage a significantly higher number of connections, as they are responsible for keeping a high throughput.

Network messages between the nodes are broadcast through P2P connections that are established and kept going, while transactions are received utilizing a websocket protocol. Each account may register to a participation node, e.g., in order to vote. Participation nodes keep a list of online accounts, and while each account should register in at most one participation node, this is not enforced.

2.3 Accounts

An Algorand account is a protocol entity represented by a 58-bytes address associated to a signing public key (see Section 2.5 for details with cryptographic algorithms and keys). For example, a user may generate a signing key pair and derive the address from the public key. On the other hand, when a contact is deployed, it is deployed in an address for which the private key is not known. There is a bijection between addresses and public keys: the address is derived from the public key by adding a 4-bytes checksum and encoding the result in base32; the inverse is computed by simply decoding and dropping the last 4 bytes.

An account need not register to a node in order to send transactions.

Accounts are managed by participation nodes. Accounts should be registered in at least 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" (see Sect. 2.4). That is achieved by sending a special `keyreg` transaction, which registers a participation key for the account.

For convenience, we assume that each participation node maintains a list of the registered accounts, plus access to all updated balance records for the accounts it manages which are flagged as online and have a valid participation key and VRF key registered on this node (see Sect. 2.4 for more details). This array is named A from now on, and each entry $a \in A$ has an account address a_I.

implementation works, however neither the implementation nor the protocol will enforce an order, so that a node could order transactions arbitrarily into a new block and the block would not be rejected during voting.

2.4 Client API and Interface

Algorand distributes a command-line interface (CLI) called GOAL[2] that allows users to interact with the protocol. The CLI supports a series of commands that allow users to generate wallets, keys, sing and send transactions to a node and more. When the CLI command involves the network (e.g., transactions), the CLI sends a message to a relay node over the Internet, and nodes will decide how to process this transaction.

There are currently seven transaction types on Algorand: Payment, Key Registration, Asset Configuration, Asset Freeze, Asset Transfer, Application Call, State Proof). A participation node keeps a transaction pool. Whenever a transaction arrives, its signature is verified (if this is a smart signature, then it is checked against the AVM); those which do not pass are discarded, and the result of computation is cached for those which do pass. It checks that current round is inside the first valid (round) and last valid (round) included in the transaction fields. It also checks if the account holds ALGO to pay fees. Then, there are other checks which depend on the transaction type and go as follows. It checks that the genesis hash included in the transaction is the same as the one configured in the node. If this is a payment transaction, it checks that the sender has sufficient funds for transfer and fee. If this is a key registration transaction, it checks the current round is inside the first valid (round) and last valid (round) defined in key registration transaction fields for the validity of the key. If this is an asset configuration or asset freeze transaction, it also checks whether the sender is set as the manager for this asset, and that this asset is not frozen among other checks. If this is an asset transfer transaction, it depends on several details; we do provide some relevant cases which include an opt in, in which a user technically "transfers an asset" to themselves and in this case it is only checked that the asset ID exists, a standard asset transfer in which it is checked that the receiver has opted in to this asset ID and the receiver holds enough ALGOs to receive the asset, that the sender has enough ALGO to pay for the transfer. If this is an application call, it runs the code against the AVM and may either discard the transaction or accept it (balance or state changes are not applied at this time). If this is a state proof transaction, checks that the last attesting round is bigger than the current round for the node, and it also checks the structure of the state proof message. Readers interested in more details should check the specifications ([Alg23]). Finally, the transaction is stored in the transaction pool.

The CLI also has a minimum fee function, that when queried returns a suggested fee to pay. This is basically constant, and only grows in the case of congestion.

2.5 Cryptrographic Algorithms

Algorand protocols make use of two hash functions of the SHA-2 family as defined in [Dan12]: SHA256 and SHA512/256.

A cryptographic singing scheme defines a triplet of algorithms ($\mathsf{Generate}(\,), \mathsf{Sign}(\,), \mathsf{Verify}(\,)$). These include a key-generation algorithm $\mathsf{Generate}(\,)$ that takes a random string and returns a public and secret key; a signing procedure $\mathsf{Sign}(\,)$ which takes a private key and a packet of data of arbitrary length and outputs a signature (a bit-string of fixed length); and a verification algorithm $\mathsf{Verify}(\,)$ which takes the public key, some data of arbitrary length and a signature and outputs a boolean value. A `true` output certifies that the signature is valid, and a `false` output implies the signature is invalid. When using cryptographic signatures, Algorand uses the ED25519 elliptic-curve cryptographic signatures standard which implements elliptic-curve Diffie-Hellman algorithm over the curve `Curve25519` ([Ber06]). It uses 256-bits keys.

A verifiable random function (VRF for short) scheme includes a key generation algorithm that outputs a pair of secret and public keys, an algorithm that given the secret key and a value produces a pseudorandom output and a *proof* (object), and a verification algorithm, which given

[2]https://developer.algorand.org/docs/clis/goal/goal/

the output and proof, verifies that the result is correct (see [SM99]).

The Algorand Protocol uses three types of cryptographic keys: root or spending keys, voting or participation keys and selection keys.

Spending and participation both use ED25519 cryptographic signatures as follows. Accounts sign their transactions (e.g., via the CLI) before sending them to a node, and the node will verify the signature before accepting the transaction (since the transaction must include the address it is destined to, and the public key can be derived from the address, the node does not need to hold the public keys in advance).

The Algorand protocol allows for a process called "rekeying", in which case the owner of a private key can authorize a new key to sign transactions on behalf of this account (and disable the older one).

At different points in the protocol provably-random accounts selection needs to happen. Basically, all accounts may generate one random number per stake using the VRF scheme. Only those accounts generating a random number smaller than a threshold are selected. The selected accounts must provide the random number and proof generated by the VRF scheme in order to participate, other parties can verify this to accept the selection. This threshold is a constant (that could change from protocol version to protocol version) that been tuned so that the number of accounts selected fall close to a target value.

Participation and VRF keys are generated on demand in the node by issuing a key generation CLI call. Keys are not permanent, so that users may need to call the key generation algorithms periodically to continue their participation. the key generation command takes an address, a range of rounds and a 64-bit integer key-dilution parameter (k). In the reference implementation, k defaults to the square root of the round range (rounded down to the nearest integer) when not specified, e.g. for a 3 million round range a 1712 batch of subkeys is generated.

Let $n := \lceil \frac{\text{last}-\text{first}}{k} \rceil$. When initialized, the key generation algorithm will do as follows:

- Generate a root key pair $(pub_r, priv_)$.
- Generate n batch key pairs $(pub^{(b_1)}, priv^{(b_1)}), \ldots, (pub^{(b_n)}, priv^{(b_n)})$ signing the public each of the batch keys with the root key $priv_r$.
- Generate $n \cdot k$ ephemeral key pairs $(pub_{e_j}^{(b_i)}, priv_{e_j}^{(b_i)})$ for $1 \le i \le n$, $1 \le j \le k$, each public ephemeral key $pub_{e_j}^{(b_i)}$ signed with the appropriate batch key $priv^{(b_i)}$.
- Issue a key registration command with the node that includes pub_r in the balance for the issuing account. (This transaction is signed, as all transactions, with the account's spending key.)

(Actually, the keys are generated as they are needed k at a time. Only the root public key is needed for verification.)

When signing, the signature algorithm will use the ephemeral keys one at a time, in lexicographical order starting with $priv_1^{(1)}$, continuing with the first batch of k ephemeral keys, and moving to the next one. After using an ephemeral key, it is deleted and never reused. Say, the signature of a first message m will include the signed message and the required signature trail:

$$\Big(\mathsf{Sign}(priv_1^{(1)}, m)\Big); \quad \Big(pub_{b_1}^{(1)}, \mathsf{Sign}(priv_{b_1}, pub_{e_1}^{(1)})\Big); \quad (pub_{b_1}, \mathsf{Sign}(priv_r, pub_{b_1}))$$

Verification goes in the opposite way: given a signed message

$$(m, s); \quad (m', s'); \quad (m'', s'')$$

(here m' is the batch public key and m' is the ephemeral public key), the algorithm

- uses the root public key pub_r which can be retrieved from the account's balance to validate the last signature $\mathsf{Verify}(pub_r, m'', s'')$.
- If the validation passes, m'' is assumed to be a batch public key and used to validate an ephemeral key: $\mathsf{Verify}(m'', m', s')$.
- If the validation passes, m' is assumed to be an ephemeral public key and used to validate the message: $\mathsf{Verify}(m'', m, s)$.
- Once this happens, the ephemeral key is recorded as used and cannot be reused.

Participation keys are used to sign all **consensus messages** (or messages of the consensus type), which are messages broadcasted (or received from other nodes and relayed) to the other nodes, and which contain votes, proposals or bundles. In particular, private participation and VRF keys are stored in the node (not their spending keys).

Selection keys used for committee membership selection and verification (see Sections **??**). VRF keys are needed in order for an account to vote, as they will be used to compute committee membership credentials, and to verify said credentials by nodes receiving consensus messages. More specifically, in order to participate in consensus on round r, an account needs to have an entry in BT with valid VRF public keys for the lookback round $r - \delta_s(1 + \delta_r)$. They make use of an Algorand fork of the Sodium C library, implementing VRF signing, verification and key generation functions. In the following work, these cryptographic functions are abstracted as $\mathsf{VRF}_{\mathsf{prove}}(.)$, $\mathsf{VRF}_{\mathsf{verify}}(.)$, $\mathsf{VRF}_{\mathsf{generate}}(.)$ and the helper function $\mathsf{VRF}_{\mathsf{proofToHash}}(.)$. Refer to the provided repository for implementation details.

We define a $\mathsf{secrets}(I, r')$ helper function, which takes a user address and a valid round number (that is lesser or equal to r), and is able to retrieve private VRF keys and private ephemeral participation keys (and living private participation subkeys) for a given user I and a valid round number r' when applicable. Note that past round ephemeral private keys are not recoverable by design. As a precondition it is assumed that in any call to $\mathsf{secrets}(.)$, the keys being retrieved were registered and valid for I in the round r', and I generated them in the node making the call (the user with address I has an entry in the local $\mathsf{Accounts}$ array).

2.6 The Algorand Virtual Machine

> *"The Algorand virtual machine (AVM) runs on every node in the Algorand blockchain. This virtual machine contains a stack engine that evaluates smart contracts and smart signatures against the transactions they're called with. These programs either fail and reject the transaction or succeed and apply changes according to the logic and contents of the transactions."* From Algorand.org

The AVM receives transactions that may or may not alter the state of the blockchain and interprets TEAL, or the Transaction Execution Approval Language, an assembly-like language designed by Algorand for the blockchain. Currently, TEAL is at version 10; opcodes are published here.

The AVM may verify what Algorand calls stateless contracts, smart signatures or logicsigs, which includes code which does not change the state and evaluates to true or false. These stateless contracts will typically require 2 or more accounts to sign, or may be valid only in a given timeframe, or even evaluate more complicated predicates.

The AVM may be called through the transaction pool to evaluate a smart signature, in which case it evaluates the code returning True or False. It may also be called to simulate an application call, in which case it simply executes the instructions but without altering the state (ledger, balances, etc).

If it is called through the block assembly algorithm, then it executes the code and records changes to ledger and balances, which will be made only after block confirmation.

2.7 Node as a State Machine

A participation node can be modeled as a state machine in its involvement in the consensus protocol, i.e., each time Algorithm 1 is called, either becasue of a time event or a message event, a procedure is called and this causes the state of the node to change. In this section we will define all primitives and data structures necessary to define the state of a node.

The state of a node is defined as follows.

$$(r, p, s, V, P, \bar{v}, \bar{s}, \mathsf{L}, \mathsf{BT})$$

where

- $r \in \mathbb{Z}_{\geq 0}$ is the current round and recorded as a 64-bits integer. The round variable start as $r = 0$, when the genesis block is generated.
- $p \in \mathbb{Z}_{\geq 0}$ is the current period within the round and recorded as a 64-bits integer. When the round starts, p is set to 0 ($p = 0$).
- $s \in \mathbb{Z}, 0 \leq s \leq 256$ is the current step within the round and recorded as a 64-bits integer. When the round starts, s is set to 0 ($s = 0$). We often use the labels of Table 1 to identify step numbers. The round, period and step parameters change during protocol execution, they are either increased by one or reset, by the Algorithms 1, 21 and 22.
- V is the set of all valid observed votes and equivocation votes in the current execution state (r, p, s). It is initialized as the empty set $V = \emptyset$, and during execution either votes are added or the set is reemptied.
- P is the set of all valid observed proposals in the current execution state (r, p, s), and it is also initialized as the empty set.
- $\bar{v}$ is the pinned proposal value, and it is initialized to $\bar{v} = \perp$.
- $\bar{s} \in \mathbb{Z}_{\geq 0}$, a 64-bits integer, is the last finished step and it is initialized to $\bar{s} = 0$.
- L is the ledger, a structure that holds a sequence of states which record some round parameters as described below.
- BT is the current balance table described below.
- TP is the transaction pool for the node (see **??**).
- The current time `time` in seconds, and the elapsed time since the last time reset was called.

The 3-tuple of nonnegative integers (r, p, s) are referred to as the execution state, the first 7 values $(r, p, s, V, P, \bar{v}, \bar{s})$ are the consensus state parameters,, and L and BT are referred to as the ledger state parameters. For convenience and readability, we define the following step name - step enumeration assignment:

$$\begin{cases} \mathsf{proposal} & \text{if } s = 0 \\ \mathsf{soft} & \text{if } s = 1 \\ \mathsf{cert} & \text{if } s = 2 \\ \mathsf{next}_{s-3} & \text{if } s \in [3, 252] \\ \mathsf{late} & \text{if } s = 253 \\ \mathsf{redo} & \text{if } s = 254 \\ \mathsf{down} & \text{if } s = 255 \end{cases} \tag{1}$$

The balance table BT allows the node to query and insert values into a set of maps representing the ALGO owned by each account, the ASA owned by each account (when an account opts-in to an ASA, an entry is created within his records for this ASA–and he must have a minimum balance

for this to happen), the local storage for all the applications that the account has opted in (same as above, he needs a minimum ALGO balance for this), the global storage for this account–in case it belongs to an application, the consensus participation status of this account (if this is a user account, e.g., if this account is online and voting or not), in case the account is online and voting, it also includes the selection and participation keys for the account (see Sect. 2.5). For completeness, given an address I for an account that does not exist as of round r, $\mathsf{BT}[r][I] = \perp$. Once a block is accepted, the node will update the balance table according to block instructions.

The LedgerL, identified by a string called the genesis identifier and the hash of the genesis string, keeps record of the ledger states. It records and allows querying a sequence of ledger states on each round including the following items.

- A genesis identifier and genesis hash, unambiguously defining the ledger to which the state belongs.
- A protocol version and update state.
- A timestamp (in milliseconds), identifying the time elapsed between the genesis block proposal and the block for round r proposal.
- A 64-byte seed, source of randomness 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.

Once a block is accepted, the node will record a new ledger state.

A block e is associated to each state and represented by a header and a body. The block header holds the following data.

- A round r, which matches the round of the state it is associated to. (The block with round 0 is special as it contains the entire initial state and is correspondingly called the genesis block).
- The genesis identifier and genesis hash for the block.
- The upgrade vote. This is a placeholder that may be used to upgrade protocol versions. Its use is not implemented nor described in the literature.
- A timestamp defined by the proposer of the block (e.g., from the node's clock). To be valid, a timestamp must be greater than the timestamp of the last block and cannot be more than 25 seconds away from said time.
- A seed.
- The reward updates for the current block. This has been discontinued.
- A pair of cryptographic vector commitments to the block's transaction sequence, i.e., this is a hash for the root of the Merkle tree that encodes all the tranasctions, one computed using SHA512/256 hash function and a second one using (the older) SHA256 hash function. This may be useful for compatibility with systems where SHA512/256 is not supported.
- A hash for the previous block (either SHA512/256 or SHA256), which is the cryptographic. In the case of the genesis block, this is set to 0.
- An integer, the transaction counter, representing the number of transactions issued prior to this block. This count starts from the first block with a protocol version that supported the transaction counter.

- A list of expired participation accounts, which contains an optional slice of public keys of accounts. This list is computed by the block proposer and includes the accounts whose keys expired in the current or earlier rounds in the proposal node generating the block. 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.

The body of a block contains a transaction sequence, which describes the sequence of updates to each of the account states and box states.

2.8 Proposal, Vote and Bundle

A Proposal is a tuple $prop = (e, \pi_{seed}, p_{orig}, I_{orig})$, where e is a full block, π_{seed} is a VRF proof of the seed computed on block creation, p_{orig} is the period in which the block was created, and I_{orig} is the address of the account propsing the block.

A proposal-value is a tuple $v = (I, p, \mathsf{H}(prop), \mathsf{H}(\mathsf{Encoding}(prop)))$ where I is an address (the "original proposer"), p is a period (the "original period"), and $\mathsf{H}(\cdot)$ and $\mathsf{Encoding}(\cdot)$ are a hash and an encoding function specified elsewhere. A proposal value also includes a signature and VRF proof of the proposal-value which are used to validate the proposer, but are omitted in the notation for the sake of brevity. There is a special proposal where all fields are the zero-string is called the **bottom proposal** and is denoted as $\perp$.

We denote by $\mathsf{Proposal}_{\mathsf{value}} : (I, p, prop) \mapsto (I, p, \mathsf{H}(prop), \mathsf{H}(\mathsf{Encoding}(prop))$ the function used by a block proposer to create a proposal-value. For convenience, we define the function $\mathsf{RetrieveProposal} : v \mapsto \mathsf{RetrieveProposal}(v)$ which is an inverse for $\mathsf{Proposal}_{\mathsf{value}}$ and is implemented by having the first function store input and output on each call.

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 weight of the vote.

An object of this type is output by the $\mathsf{Sortition}(.)$ procedure (Algo. 32).

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

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

Let $vt = \mathsf{Vote}(I, r, p, s, v, cred)$ and $vt' = \mathsf{Vote}(I', r', p', s', v', cred')$ be a pair of votes. An equivocation vote happens if all but the proposal values agree, i.e., $I = I'$, $r = r'$, $p = p'$, $s = s'$, $v \neq v'$ and $cred = cred'$. An equivocation vote is valid if both of its constituent votes are valid. According to engineering specifications, nodes cannot equivocate on any steps $s \neq soft$ (although this is may not be enforced by the reference implementation).

A bundle is a set of votes $\mathsf{Bundle}(r, p, s, v)$ such that for any pair of distinct values $vt = \mathsf{Vote}(I, r, p, s, v, cred)$, $vt' = \mathsf{Vote}(I', r', p', s', v', cred')$ in b the following conditions hold: a) they have the same execution state (i.e., $r = r'$, $p = p\prime$, and $s = s'$) b) for each pair of distinct elements, either $I \neq I'$ and $v = v'$ or the pair vt, vt' is an equivocation vote, and c) 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. (This is a protocol constant.)

For convenience, we define the observed bundles for the current execution state, B, as the set of all of the bundles observed by a node in the current execution state. Said set is always consistent with an in-place reconstruction by grouping elements of V in all possible subsets with the required properties of a valid bundle.

3 Vanilla run

Let us assume a genesis block was generated, the blockchain has been running and has already generated blocks, with a set of nodes and accounts. We are now at round $r - 1$ ($r > 2$), meaning that $r - 1$ blocks have been generated and confirmed by the blockchain. Moreover, the node has received transactions which have been verified to be correctly signed by Algorand accounts and validated according to Ledger context, and has added them to its transaction pool and relayed them to other nodes. For this section, we assume that all nodes behave according to protocol and that they are in sync (e.g., the context (r, s, p) for all nodes is the same and their internal clocks are synchronized).

The node keeps some values in memory or storage. In particular the timer, which we assume is now at $timer = 0$, a round number $r \in \mathbb{Z}_{\geq 0}$, a step $s := Table\ 1 = 0 \in \mathbb{Z}_{\geq 0}$ (see Table 1), a period $p := 0 \in \mathbb{Z}_{\geq 0}$, a last finished step $\bar{s} := 0 \in \mathbb{Z}_{\geq 0}$, and a pinned vote $v := \perp$.

The main algorithm (Algorithm 1) for consensus is an event handler that receives two types of input from the node. It receives messages, and it is also able to read the timer.

As the main algorithm starts a round, it is called with the event $timer = 0$ and calls the Block Proposal Algorithm (Algorithm 2). The $\mathsf{BlockProposal}()$ algorithm runs, moving directly to a loop in which it iterates over all the accounts that are registered in the node.

For each account a, the node runs $\mathsf{Sortition}(a, \mathsf{proposal})$ (Algo. 32) which basically runs a Verifiable Random Function in order to decide whether this account is selected into the voting process (by picking a random value). If sortition returns the zero value, this means the account has not been selected for block proposal. If no account is selected, the algorithm exits and the node will no longer be a proposer in the proposal step. When at least one account gets selected, the node participates in the proposal voting representing the account and starts with block assembly (Algorithm 11). This procedure will traverse the transaction pool, calling the AVM and executing transactions one at a time, obtaining as a result a new block e. It will assemble a proposal *prop* a vote proposal v, set v as the proposal value obtained from the vote e, and make two separate broadcasts for $\mathsf{Vote}(a_I, r, p, proposal, v, cred)$ and e. Then, the main algorithm enters the "soft vote" step setting $s := 1$.

Assume that some time has passed and now $0 < timer < \mathsf{FilterTimeout}(p)$, and that the node received a block proposal e' which was broadcasted from another node. Then, the event handler runs $\mathsf{HandleProposal}(e')$ (Algorithm 4). This algorithm receives the proposal e' and unpacks its contents, including the execution state (r', p', s'). By assumption, we have that $r = r'$ and $p = p' = 0$. The algorithm checks if the proposal is valid, calling $\mathsf{VerifyProposal}(v')$ on $v' = Proposal_{value}(e')$ (see Algorithm 19 for more details), it also checks if periods differ ($p \neq p'$), exiting if not. However, both checks pass by the vanilla run assumptions. Next, if $e' \in P$, it returns; else HandleProposal re-broadcasts e', adds e' to a set P of stored proposals, and exits.

Let us say that the node received a broadcasted vote vt (e.g., as those sent by the HandleProposal algorithm) and assume that $0 < timer < \mathsf{FilterTimeout}(p)$ still holds. The event handler for the

main algorithm thus calls $\mathsf{HandleVote}(vt)$ (Algorithm 5). The algorithm exits on checks –which are passed with the vanilla run assumptions. It also exits if the vote received has already been recorded in the votes set V. If it is new, the node adds the vote to the votes set V and broadcasts the vote to other nodes. Since nodes are synchronized, it holds that $vt_s = 0 = \mathsf{proposal}$, the algorithm checks if $\mathsf{RetrieveProposal}(vt_v) \neq \perp$ and broadcasts this proposal if it is available, ignoring if it is not.

Until $timer \geq \mathsf{FilterTimeout}(p)$ the main algorithm will execute the above steps whenever a vote or a proposal are received.

Eventually, the timer is at $timer = \mathsf{FilterTimeout}(p)$ and the main algorithm calls $\mathsf{SoftVote}()$ (Algorithm 3). The soft vote is about selecting a priority block and voting it in, and it proceeds as follows. It initializes to an empty vote $v = \perp$ and with the lowest observed hash at infinity ($lowestObservedhash := \infty$). The node will go through all the votes $vt' \in V$ in its votes set that are in the proposal step ($vt'_s = 0$). Given the credentials for the vote $vt'_{credential} = (sh, \pi, j)$, where sh is the hash, π is the proof and $j \in \mathbb{Z}_{\geq 0}$, the algorithm computes the priority hash

$$priorityHash := \min\{\mathsf{Hash}(sh \| i) : 0 \leq i < j\},$$

and whenever this value (as an integer) is lower than the recorded $lowestObservedHash$ it sets $lowestObservedHash := priorityHash$ and $v := vt'_v$. Next, if there was at least one vote in V, for every registered account $a \in A$ it computes

$$(sh, \pi, j) := credential'' := \mathsf{Sortition}(a, soft) = \mathsf{Sortition}(a, 1)$$

and, if $j > 0$ it broadcasts $\mathsf{Vote}(r, p, soft, v, credential'')$. Moreover, if the proposal $prop := \mathsf{RetrieveProposal}(v)$ is not $\perp$, it also broadcasts $prop$.

When the main algorithm receives a message of type proposal $prop$ as above, it will run the handle proposal algorithm as before.

When the main algorithm receives a message of type vote $\mathsf{Vote}(r, p, soft, v, credential)$ as above, it relays the vote, adds it to the votes set (if it is new) and checks whether it can form a bundle from the votes in V, i.e., it checks if there is a vote v such that for all the votes $vt \in V$ with round $vt_r = r, vt_p = 0, vt_s = soft$ that have the same vote $vt.v$ and such that the sum of its weights ($\sum_{vt \in V} vt_{cred_j} \geq \mathsf{CommitteeThreshold}(soft)$) is bigger than the committee threshold. Imagine that the first time this happens, it cannot form a bundle.

Eventually it basically adds the accepted block to the ledger, modifies the state according to the new block, garbage collects and sets the new round. That is, it will broadcast the proposal if it is not in the proposal set P, and then commit v, calling $\mathsf{Commit}(v)$, set $r_{old} := r$, and calls $\mathsf{StartNewRound}(r+1)$ and $\mathsf{GarbageCollect}(r_{old}, p)$ ending the round. Calling the new round algorithm (Algo. 21) will reset variables as: $\bar{s} := s$, $\bar{v} := \perp$, $r := r+1$, $p := 0$ and $s := proposal = 0$; and calling the garbage collection algorithm (Algo. 23) will compute

$$\begin{aligned} V_{(r,p-1)} &:= \{vt \in V : vt_r < r \text{ or } (vt_r = r \text{ and } vt_p + 1 < p)\} \\ P_{(r,p-1)} &:= \{pr \in PP : pr_r < r \text{ or } (pr_r = r \text{ and } pr_p + 1 < p)\} \end{aligned}$$

and then remove these sets from the votes and participation sets: $V := V \backslash V_{(r,p-1)}$, $P = P \backslash P_{(r,p-1)}$.

4 Main algorithm

??

The consensus algorithm implements a service which can spawn processes depending on the timer and on messages received.

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(); s := 1
6:         else if time = FilterTimeout(p) then
7:             s := 2; SoftVote()
8:         else if time = max{4λ, Λ} then
9:             s := 3; Recovery()
10:        else if time ∈ [max{4λ, Λ} + 2^(s_t−3) λ, max{4λ, Λ} + 2^(s_t−2) λ) for some s ≤ s_t ≤ 252 then
11:            s := s_t; Recovery()
12:        else if time = kλ_f + r for some k, r ∈ ℤ, k > 0, 0 ≤ r ≤ λ_f then
13:            FastRecovery()
14:    else //ev is MessageEvent
15:        msg := ev_msg
16:        if msg_data is of type Proposal pp then
17:            HandleProposal(pp)
18:        else if msg_data is of type Vote v then
19:            HandleVote(v)
20:        else if msg_data is of type 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 committed to the ledger). However, multiple periods might be run inside a round until a certification bundle ($\mathsf{Bundle}(r, p, s, v)$ 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.

Timeout Events 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 $\mathsf{FilterTimeout}(p)$ 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 an $next_0$ 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 $st \in [4, 252]$ 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.

Message Events 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:      for a ∈ A do
4:          cred := Sortition(a_I, r, p, proposal)
5:          if cred_j > 0 then
6:              if p = 0 ∨ ∃s'/Bundle(r, p − 1, s', ⊥) ⊂ V then
7:                  (e, π_seed) := AssembleBlock(a_I)
8:                  prop := Proposal(e, π_seed, p, a_I)
9:                  v := Proposal_value(prop)
10:                 Broadcast(Vote(a_I, r, p, proposal, v, cred))
11:                 Broadcast(prop)
12:             else
13:                 Broadcast(Vote(a_I, r, p, proposal, v̄, cred))
14:                 if RetrieveProposal(v̄) ≠ ⊥ then
15:                     Broadcast(RetrieveProposal(v̄))
```

Description:
This algorithm is the first called on every round and period. Starting on line 2, the node attempts a resynchronization (described in 5), which only has any effect on periods $p > 0$.

The algorithm loops on all of its managed online accounts ($a \in A$). This is a pattern that we'll see in every other main algorithm subroutine that performs any form of committee voting. For each account, the sortition algorithm is run to check if the account is allowed to participate in the proposal. If an account a is selected by sortition (because $cred_j = \mathsf{Sortition}(a_I, r, p, proposal)_j > 0$) there are two options.

i) If this is a proposal step ($p = 0$) or if the node has *observed* a bundle $\mathsf{Bundle}(r, p-1, s', \perp)$ (meaning there is no valid pinned value), then the algorithm assembles a block, computes the proposal value for this block, broadcast a proposal vote by the account a, and broadcasts the full block in a proposal type message.

ii) Otherwise, a value $\bar{v}$ has been pinned supported by a bundle observed in period $p-1$, and on line 12 the node gets this value, assembles a vote $\mathsf{Vote}(a_I, r, p, proposal, \bar{v}, cred)$, broadcasts this vote, and will also broadcast the proposal for the pinned vote if it was already observed. Then, for every account selected, a proposal vote for this pinned value is broadcast. Afterwards, if the corresponding full proposal has been observed, then it is also broadcast.

4.2 Soft Vote

Algorithm 3 Soft Vote

```
1:  function SoftVote( )
2:      lowestObservedHash := ∞
3:      v := ⊥
4:      for vt' ∈ V with vt'_s = proposal do
5:          priorityHash := min_{i∈[0,vt'_{cred_j})} {H(vt'_{cred_sh} || i)}
6:          if priorityHash < lowestObservedHash then
7:              lowestObservedHash := priorityHash
8:              v := vt_v
9:      if lowestObservedHash < ∞ then
10:         for Account a ∈ A do
11:             cred := Sortition(a_I, r, p, soft)
12:             if cred_j > 0 then
13:                 Broadcast(Vote(r, p, soft, v, cred))
14:                 if RetrieveProposal(v) then
15:                     Broadcast(RetrieveProposal(v))
```

Description:
The soft vote stage (also known as "filtering") is run after a timeout of $\mathsf{FilterTimeout}(p)$ (where p is the executing period of the node) is observed by the node. That is to say, filtering is triggered after either a $2 \cdot \lambda_0$ or $2 \cdot \lambda$ timeout is observed according to whether $p = 0$ or $p > 0$ respectively. Let V^* be all proposal votes received, e.g., $V^* = \{vt' \in \mathsf{V} : vt'_s = proposal\}$. With the aid of a priority hash function, this stage performs a filtering action, keeping the lowest hashed value observed. The priority function (lines 4 to 8) should be interpreted as follows. Consider every proposal vote vt in V^*. Given the sortition hash sh output by the VRF for the proposer account, and for each sub-user unit i in the interval from 0 (inclusive) to the vote weight (exclusive; the j output of $\mathsf{Sortition}(.)$ inside the *cred* credentials structure), the node hashes the concatenation of sh and i, $\mathsf{H}(sh||i)$ (where $\mathsf{H}(.)$ is the node's general cryptographic hashing function). On lines 6 to 8, then, it keeps track of the proposal-value that minimizes this concatenation and subsequent hashing procedure. After running the filtering algorithm for all proposal votes observed, and assuming

there was at least one vote in V^*, the broadcasting section of the filtering algorithm is executed (lines 9 to 15). For every online managed account selected to be part of the $soft$ voting committee, a $soft$ vote is broadcast for the previously found filtered value v. If the full proposal has been observed and is available in P, it is also broadcast. If the previous assumption of non-empty $V*$ does not hold, no broadcasting is performed and the node produces no output in its filtering step.

4.3 HandleProposal

Algorithm 4 Handle Proposal

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

Here μ 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.

σ 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.

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 the current context and the characteristics of the proposal.

In lines 3 to 5, it is checked if the proposal is from the next first period of the round next to the current, in which case the node relays this proposal and then ignores it for the purpose of the current round. Whenever the node catches up (i.e., observes a round change), and only if necessary, it will request this proposal back from the network. Lines 6 and 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 and 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 passed, in lines 10 and 11, the algorithm relays and observes the proposal (by adding it to the observed proposals set, P).

Next, and only if the proposal value is committable and the current step is lower than or equal to a certification step (i.e., it is 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 Handle Vote

```
1: function HandleVote(Vote vt)
2:     if not VerifyVote(vt) then
3:         DisconnectFromPeer(SENDER_PEER(vt)) return //ignore invalid vote
4:     if vt_s = 0 ∧ (vt ∈ V ∨ IsEquivocation(vt)) then return //ignore vote, equivocation not allowed in proposal votes
5:     if vt_s > 0 ∧ IsSecondEquivocation(vt) then return //ignore vote if it's a second equivocation
6:     if vt_r < r then return //ignore vote of past round
7:     if vt_r = r + 1 ∧ (vt_p > 0 ∨ vt_s ∈ {next_0, ..., next_249}) then return //ignore vote of next round if non-zero period or next_k step
8:     if vt_r = r ∧ (vt_p ∉ {p − 1, p, p + 1}∨
9: (vt_p = p + 1 ∧ vt_s ∈ {next_1, ..., next_249})∨
10: (vt_p = p ∧ vt_s ∈ {next_1, ..., next_249} ∧ vt_s ∉ {s − 1, s, s + 1})∨
11: (vt_p = p − 1 ∧ vt_s ∈ {next_1, ..., next_249} ∧ vt_s ∉ {s̄ − 1, s̄, s̄ + 1})) then return //ignore vote
12:     V := V ∪ vt //observe vote
13:     Relay(vt)
14:     if vt_s = proposal then
15:         if RetrieveProposal(vt_v) ≠ ⊥ then
16:             Broadcast(RetrieveProposal(vt_v))
17:     else if vt_s = soft then
18:         if ∃v|Bundle(vt_r, vt_p, soft, v) ⊂ V then
19:             for a ∈ A do
20:                 cred := Sortition(a_sk, r, p, cert)
21:                 if cred_j > 0 then
22:                     Broadcast(Vote(a_I, r, p, cert, v, cred))
23:     else if vt_s = cert then
24:         if ∃v|Bundle(vt_r, vt_p, cert, v) ⊂ V then
25:             if RetrieveProposal(v) = ⊥ then
26:                 RequestProposal(v) //waits or keeps playing without voting power
27:                 if p < vt_p then
28:                     p_old := p
29:                     StartNewPeriod(vt_p)
30:                     GarbageCollect(r, p_old)
31:             Commit(v)
32:             r_old := r
33:             StartNewRound(vt_r + 1)
34:             GarbageCollect(r_old, p)
35:     else if vt_s > cert then
36:         if ∃v|Bundle(vt_r, vt_p, vt_s, v) ⊂ V then
37:             p_old := p
38:             StartNewPeriod(vt_p + 1)
39:             GarbageCollect(r, p_old)
```

A vote vt is an uple consisting of a secret key vt_{sk}, a round vt_r, a step vt_r and a period vt_p.

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 is 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 disconnect from the sender of the vote. In that case, we use SENDER_PEER(msg) as a helper network module function, that retrieves the network ID of the original message sender. 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 it is 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, RetrieveProposal(v) is broadcast if it has been observed (that is, the player performs a re-proposal 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(.) sub-procedure 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 $prop$ = RetrieveProposal(v) implies that the associated entry is committable. 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 can be committed, 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 Handle Bundle

```
1: function HandleBundle(Bundle b)
2:     if ¬VerifyBundle(b) then
3:         DisconnectFromPeer(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), it is 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 $b\prime = \mathsf{Bundle}(b_r, b_p, b_s, v\prime)$ is observed (where $v\prime$ is not necessarily equal to b_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( )
2:     ResynchronizationAttempt()
3:     for Account a ∈ A do
4:         cred := Sortition(a_I, r, p, s)
5:         if cred_j > 0 then
6:             if ∃v = Proposal_value(prop, prop_p, prop_I) for some prop ∈ P|IsCommitable(v) then
7:                 Broadcast(Vote(a_I, r, p, s, 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:                 Broadcast(Vote(a_I, r, p, s, v̄, cred))
10:            else
11:                Broadcast(Vote(a_I, r, p, s, ⊥, cred))
```

Description:
The recovery algorithm that is executed periodically, whenever a *cert* bundle has not been observed before $\mathsf{FilterTimeout}(p)$ 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 can be committed in the current context, a $next_k$ vote for v is broadcast by the player.
If no proposal-value can be committed, 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_I, r, p, late)
6:             if cred_j > 0 then
7:                 Broadcast(Vote(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_I, r, p, redo)
10:            if cred_j > 0 then
11:                Broadcast(Vote(r, p, redo, v̄, cred))
12:        else
13:            cred := Sortition(a_I, r, p, down)
14:            if cred_j > 0 then
15:                Broadcast(Vote(r, p, down, ⊥, cred))
16:    for vt ∈ V|vt_s ≥ 253 do
17:        Broadcast(vt)
```

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 main difference is that it emits votes for any of three different steps (*late*, *redo* and *down*) according to sortition results for every account. Finally, the algorithm broadcasts all fast recovery votes observed. That is, all votes $vt \in \mathsf{V}$ for which vt_s is a fast recovery step (*late*, *redo* or *down*).

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:
The function FilterTimeout(.) provides the timeout constant for the filtering (i.e., "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:         return SHA512/256(in)
4:     else
5:         return SHA256(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, it falls back to $SHA256$.

Returns: The result of hashing the input *in* with the selected algorithm based on underlying system support

Algorithm 11 Assemble Block

```
1: function AssembleBlock(Address I)
2:     Block e
3:     //define block body, ordered set of signed transactions
4:     it_TP := iterator(TP_head)
5:     while Time() < Deadline ∧ hasNext(it_TP) do
6:         txn := next(it_TP)
7:         if TxnSetValid(b_payset||txn) then
8:             e_payset := e_payset||txn
9:
       //define block header
10:    e_r := r
11:    e_prevHash := H(PrevBlock())
12:    (Q, π_seed) := ComputeSeedAndProof(I)
13:    e_Q := Q
14:    e_timestamp := time() //timestamp in seconds since epoch
15:    e_GenesisID := PrevBlock()_GenesisID
16:    e_GenesisHash := PrevBlock()_GenesisHash
17:    e_txnCommitment := root(MerkleTree(e_payset))
18:    return (e, π_seed)
```

Description:
The PrevBlock() function used in block assembly is a helper function that, accesses the Ledger L of the node, and returns the last block committed. That is, PrevBlock() $\equiv L[r-1]$

This function will run until it times out or there are no more transactions to process in the transaction pool. It will call the AVM to process them all, record the changes that need to happen, assemble the block as described in Section 2.7 and exits.

Algorithm 12 VerifyBlock

1: **function** VerifyBlock($Block\ b, \pi_{seed}, p_{orig}, I_{orig}$)
2: //Header pre-validation
3: **if** $b_r \neq \mathsf{PrevBlock}()_r + 1$ **then return** False
4:
5: **if** $L[r-1]_{timestamp} > 0$ **then** //TODO: translate this code to algorithm

Description:

Algorithm 13 IsCommitable

1: **function** IsCommitable($Proposal_{value}\ v$)
2: **return** ($\mathsf{RetrieveProposal}(v) \neq \perp$) $\wedge$ ($\mathsf{Bundle}(r, p, soft, v) \subset V$)

Arguments:

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

Description:
Checks that the value v is committable in the current node's context. To be committable, 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 committability of the argument v.

Algorithm 14 Commit

1: **function** Commit($Proposal_{value}\ v$)
2: $e := \mathsf{RetrieveProposal}(v)_e$
3: $L := \mathsf{L}||e$
4: $\mathsf{UpdateBT}(e)$
5: $\mathsf{UpdateTP}(e_{payset})$

Arguments:

- $Proposal_{value}\ v$, a proposal-value to be committed.

Description:
Commits the corresponding block for the proposal value received into the ledger. The proposal value must be committable (which implies validity and availability of the full ledger entry and seed). Line 2 means the algorithm will append the block contained in the proposal $\mathsf{RetrieveProposal}(v)$. Afterwards, it updates the balance table BT with all state changes called for the committed entry. The function $\mathsf{UpdateBT}(.)$ is responsible for updating the table by incorporating all changes outlined by transactions in the block body (we defer to the specifications for further details).

Algorithm 15 VerifyVote

function VerifyVote($Vote\ vt$)
  $valid := vt_r \leq r + 2$
  if $vt_s = 0$ **then**
    $valid := valid \wedge vt.v.p_{orig} \leq vt_p$
    if $vt_p = vt.v.p_{orig}$ **then**
      $valid := valid \wedge vt.v.I_{orig} = vt_I$
  if $vt_s \in \{propose, soft, cert, late, redo\}$ **then**
    $valid := valid \wedge vt.v \neq \perp$
  else if $vt_s = down$ **then**
    $valid := valid \wedge vt.v = \perp$
  $valid := valid \wedge$ VerifyPartSignature(vt)
  $valid := valid \wedge$ VerifyVRF($vt_\pi, vt_I, vt_r, vt_p, vt_s$)
  return $valid$

Description:
Vote verification gets an unauthenticated vote and returns an authenticated vote and a boolean indicating whether authentication succeeded or failed. The procedure retrieves the relevant ephemeral participation public key for the user given the vote round vt_r, checking that it exists and taking into account the lookback period $\delta_s\delta_b$. It then checks that the currently executing round (minus lookback) is inside the validity interval for said key. Finally, the internal cryptographic signature verification function (leveraging Algorand's fork of the sodium C library) is called to check the signature itself against the provided key and the vote serialized as plain bytes.

Algorithm 16 Proposal

function RetrieveProposal($Proposal_{value}\ v$)
  if $\exists prop \in P \mid Proposal_{value}(prop) = v$ **then**
    return $prop$
  else
    return $\perp$

Description:
Gets the proposal associated to a given proposal-value, if it has been observed in the current context. Otherwise, it returns $\perp$, defined as the "empty proposal" (special value where all proposal fields are zeroes).

Algorithm 17 Make Proposal

function MakeProposal($Block\ e, \pi_{seed}, uint64\ p_{orig}, Address\ I_{orig}$)
  $proposal\ prop := (e, \pi_{seed}, p_{orig}, I_{orig})$ **return** $prop$

Description:
This function takes a block (which should have been validated by the node), a VRF proof of the seed in the block header, the original period in which it was created and the address of the original proposer (creator) of said block. It then packages the block, and all the other data which corresponds with everything needed to validate it, into a *proposal* structure.

Algorithm 18 Proposal-value

function Proposal$_{\text{value}}$($Proposal\ pp$)
  $proposal_{value}\ v := (pp_{I_{orig}}, pp_{p_{orig}}, \mathsf{H}(pp_e), \mathsf{H}(\text{Encoding}(pp)))$
  return v

Description:
Constructs a proposal-value v for the input block e. The resulting proposal-value is a tuple that contains the address of the original proposer for the block, the original period in which it was assembled, the hash of the block (taken as plain bytes) and the hash of the msgpack encoding of the block.

Algorithm 19 VerifyProposal

1: **function** VerifyProposal($ProposalPayload\ pp$)
2: $valid := \mathsf{ValidEntry}(pp_e, pp_{\pi_{seed}}, L)$
3: $valid := valid \wedge \mathsf{VerifySignature}(pp_y)$

Description:

Algorithm 20 VerifyBundle

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

Description:
This procedure verifies a received Bundle, through a series of checks. On line 2, it ensures that all individual constituting b are individually valid. Line 3 checks that the Bundle has no repeat votes. Line 4 checks that all votes in b match the same (r, p, s) execution context. Line 5 checks that all votes are towards the same proposal value, with the exception of equivocation votes which are interpreted as wildcard towards any proposal value. Finally, on line 6 it checks that the sum of the weight of all constituting votes is greater or equal to the corresponding committee threshold (otherwise, the bundle would not be complete). After all aformentioned checks have passed, a boolean true value is returned. If any of them failed, a false value is returned instead, and b is deemed invalid.

Here

$$CommitteeSize(s) := \begin{cases} 20 & \text{if } s = Proposal \\ 2990 & \text{if } s = Soft \\ 1500 & \text{if } s = Cert \\ 500 & \text{if } s = Late \\ 2400 & \text{if } s = Redo \\ 6000 & \text{if } s = Down \\ 5000 & \text{otherwise} \end{cases}$$

Committee Threshold: $CommitteeThreshold(s)$ is a 64-bit integer defined as follows:

$$CommitteeThreshold(s) := \begin{cases} 0 & \text{if } s = Proposal \\ 2267 & \text{if } s = Soft \\ 1112 & \text{if } s = Cert \\ 320 & \text{if } s = Late \\ 1768 & \text{if } s = Redo \\ 4560 & \text{if } s = Down \\ 3838 & \text{otherwise} \end{cases}$$

Algorithm 21 Start New Round

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

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 $proposal = 0$ respectively.

Algorithm 22 Start New Period

1: **function** StartNewPeriod($uint64\ newPeriod$)
2: $\bar{s} := s$
3: $s := proposal$
4: **if** $\exists v = Proposal_{value}(e), s' \,|e \in P,\ v \neq \perp \wedge (s' = soft \vee s' > cert) \wedge Bundle(r, newPeriod - 1, s, v) \subset V$ **then**
5: $\bar{v} := v$
6: **else if** $\sigma \neq \perp$ **then**
7: $\bar{v} := \sigma$
8: $p := newPeriod$
9: Reset $time$.

σ 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.

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 existence 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 σ 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 σ is a valid non-empty proposal-value, the pinned value $\bar{v}$ is set to this (line 7). If none of the above conditions were met, the pinned value remains unchanged going into the new period. Finally, the node updates p to match the period to start.

Algorithm 23 Garbage Collect

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

Description:
Garbage collection algorithm, for the finished $(r', p') \in \mathbb{Z}^2_{\geq 0}$ context. The procedure discards all votes in V and proposals in P where the round of emission is less than the new round, or the round of emission is equal to the new round and the period of emission is below the period directly before the current one. In particular, when starting a new round both V and P are set to the empty set, if this is a new period for the same round, it keeps votes and proposals for the current round and both the current and previous periods.

Algorithm 24 Request Proposal

1: **function** RequestProposal($Proposal_{value}$ v)
2: async request $e \neq \perp \mid e \in \mathsf{P} \implies$ RetrieveProposal$(v) == e$ from PEERS

Description:
This call opens a channel to asynchronously request the block for which the node has observed a complete certificate, but has not observed a proposal. In particular, it does not have the corresponding block for ledger commitment. The specs. allow some implementation divergence in this case. A node may continue participating in consensus while it asynchronously waits for the proposal in the open channel for this purpose. When taking this approach, the node is able to vote, but may not emit votes for a specific value until it has received the missing block or blocks, and may only produce votes for $\perp$ when applicable. Alternatively, the node may choose to stall, stopping its participation in consensus until the full ledger is available.

Algorithm 25 Disconnect from Peer

1: **function** DisconnectFromPeer(PEER_ID)
2: Remove PEER_ID from PEERS

Description: Function to send a hint for disconnection from a given peer, on a network level. PEER_ID is a libp2p peer ID, a base58 encoded multihash string. Refer to libp2p official documentation for further details. Algorand works using libp2p to setup a P2P network, where nodes gossip messages to one another through websockets, and mantain a list of active peers. Consider a peer acts in one of the following ways:

- sends badly formed votes, bundles or proposals,
- its connection becomes slow or idle,
- remains the least performing peer for a given amount of time,
- requests a disconnect (e.g. is going offline),
- sends messages with invalid network credentials (e.g. an invalid peer ID),

- its connection is found to be duplicated in the peer list or
- on a proposal request scenario, sends more data than requested.

In any of these cases, the outcome of accepting the message is generally adversarial to the correct functioning of the network. The node disconnects from the peer, closing the websocket and taking its entry off of the available peer list, and the content of the message is ignored and discarded.

Algorithm 26 Resynchronization Attempt

```
1:  function ResynchronizationAttempt( )
2:      Val = ⊥
3:      if ∃v|Bundle(r, p, soft, v) ⊂ V then
4:          Broadcast(Bundle(r, p, soft, v))
5:          val = v
6:      else if ∃s_0 > cert|Bundle(r, p − 1, s_0, ⊥) ⊂ V then
7:          Broadcast(Bundle(r, p − 1, s_0, ⊥))
8:      else if ∃s_0, v|s_0 > cert ∧ v ≠ ⊥ ∧ Bundle(r, p − 1, s_0, v) ⊂ V then
9:          Broadcast(Bundle(r, p, s_0, v))
10:         val = v
11:     if val ≠ ⊥ ∧ RetrieveProposal(v) ≠ ⊥ then
12:         Broadcast(RetrieveProposal(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 RetrieveProposal(v) is broadcast if it has been observed.

Algorithm 27 Broadcast

```
1:  function Broadcast(data)
        Send msg(data) to all peers in PEERS
```

Description:
This function serves as a nexus between the networking module and the modules responsible for running consensus, abstracting away networking details and allowing a node to produce output for other nodes in the network to observe. msg(.) abstracts away the process of packing all data to be sent into the relevant message type, according to the nature of the data, filling out its type field depending on whether it is a transaction, vote, proposal or bundle.

Algorithm 28 Relay

```
1:  function Relay(data)
        Send msg(data) to all peers in PEERS except for SENDER_PEER(data)
```

Description:
Similar to the broadcast function, but in this case the node excludes explicitly from the list of senders a peer whose ID is that of the original sender of the input data. The function is used to re-send received messages while avoiding unnecesary forwarding loops that may congest the network.

5.1 Sortition and seed computation

In the next set of functions, we make use of special cryptographic computation functions $\mathsf{VRF}_{\mathsf{prove}}(.)$, $\mathsf{VRF}_{\mathsf{verify}}(.)$ and $\mathsf{VRF}_{\mathsf{proofToHash}}(.)$. Implementation-wise, the node makes use of an Algorand specific fork of the sodium library. Conceptually, the prove algorithm is used to execute a VRF for some given input data and a given secret VRF key, outputing an 80 byte string which we refer to as sortition proof π. The proof to hash algorithm is a helper function to obtain the sortition hash sh out of the proof π. Finally, the verify function is used to check for validity of a VRF result given input data, a public VRF key, a sortition hash and a proof, certifying that the hash and proof are output by running an instance of $\mathsf{VRF}_{\mathsf{prove}}(.)$ with the corresponding secret VRF key and data. In other words,
$\mathsf{VRF}_{\mathsf{verify}}(\pi, sh, I_{VRFKey}) = \mathsf{True} \leftrightarrow \mathsf{VRF}_{\mathsf{prove}}(in, secrets(I, r')_{VRFkey}) = \pi \wedge \mathsf{VRF}_{\mathsf{proofToHash}}(\pi) = sh$
for a given round $r' \leq r$ in which there is a valid VRF key pair for I. Refer to the aformentioned library for details on implementation of these cryptographic functions.

Algorithm 29 Compute Seed And Proof

1: **function** $\mathsf{ComputeSeedAndProof}(Address\ I)$
2: **if** $p = 0$ **then**
3: $y := \mathsf{VRF}_{\mathsf{prove}}(L[r - \delta_s]_Q, secrets(I)_{VRFkey})$
4: $\alpha := \mathsf{H}(\mathsf{VRF}_{\mathsf{proofToHash}}(y), a_I)$
5: **else**
6: $y := 0$
7: $\alpha := \mathsf{H}(L[r - \delta_s]_Q)$
8: **if** $r \mod \delta_s\delta_r < \delta_s$ **then**
9: $Q := \mathsf{H}(\alpha || \mathsf{H}(L[r - \delta_s\delta_r]))$
10: **else**
11: $Q := \mathsf{H}(\alpha)$
12: **return** (Q, y)

Arguments:

- *Address* I, the address of an online user who will be computing the seed.

Description:
Computes the cryptographic seed that goes in the block for round r, and will be used as a source of randomness for sortition in a future round. The seed is computed according to whether the function is called in a first period, $p = 0$, or not. It also computes the proof π_{seed}, bundled up with the block inside a proposal structure for broadcasting, and used by nodes receiving the proposal as part of the proposal validation process.

Returns:
the computed seed Q for the given block, ledger context, round and period, as well as the sortition proof π_{seed} to validate it

Algorithm 30 Verify Seed

```
1: function VerifySeed(Q, π_seed, Address I_proposer)
2:     q_0 := L[r − δ_s]_seed
3:     if p = 0 then
4:         if ¬VRF_verify(π_seed, q_0, I_proposer) then return False
5:         q_1 := H(VRF.ProofToHash(π)||I_proposer)
6:     else
7:         q_1 := H(q_0)
8:     if r ≡ (r mod δ_s) mod δ_s δ_r then
9:         return Q = H(q_1||H(L[r − δ_s δ_b]))
10:    else
11:        return Q = q_1
```

Arguments:

- Q, a 32 byte computed seed to verify
- π_{seed}, the 80 byte proof for the $\mathsf{VRF}_{\mathsf{prove}}(.)$ call that computed Q
- *Address* $I_{proposer}$, the address of the user that proposed a block and computed Q

Description:
The seed verification procedure certifies the validity of a computed seed Q.

Returns:
A boolean value. True if the seed Q, with proof π_{seed}, computed by I, is valid for the current node context, otherwise returns False.

Algorithm 31 Get Total Online Stake

```
1: function getTotalOnlineStake(r′)
2:     sum := 0
3:     for br ∈ BT[r′] do
4:         if br_online ∧ br_VRFkey then
5:             sum := sum + br_balance
6:     return sum
```

Arguments:

- *uint*64 r', a valid round ($r' \leq r$, where r is the node's currently executing round).

Description:
This helper function computes the sum of all online stake in a given round. For each account that had been created up to round r', that is registered as online and has an active VRF public key in its balance record for the given r', it adds the recorded balance for that context. It returns the sum of that balance over all accounts, being the total online stake in the network on that particular round r'.

Returns:
the total stake at play in the relevant round r'.

Algorithm 33 Sortition

1: **function** Sortition($Address\ I, uint64\ round, uint64\ period, uint8\ step$)
2: $Q := \mathsf{L}[r - \delta_s]_Q$
3: $\pi := \mathsf{VRF}_{\mathsf{prove}}(Q||round||period||step, \mathsf{secrets}(I, r - \delta_s\delta_r)_{VRFKey})$
4: $sh := \mathsf{VRF}_{\mathsf{ProofToHash}}(\pi)$
5: $w := \mathsf{BT}[round - \delta_s\delta_r][I]_{balance}$
6: $W := \mathsf{getTotalOnlineStake}(round - \delta_s\delta_r)$
7: $\tau := \mathsf{CommitteeSize}(step)$
8: $t := \frac{\tau}{W}$
9: $j := 0$
10: **while** $\frac{sh}{2^{\lfloor \log_2(sh) \rfloor}} \notin [\sum_{k=0}^{j} \mathsf{B}(k; w, t), \sum_{k=0}^{j+1} \mathsf{B}(k; w, t))$ **do**
11: $j := j + 1$
12: **return** credentials(sh, π, j)

Arguments:

- *Address* I, the address of an online account. Their keys are used for computation of VRFs.
- *uint*64 *round*, a round that is valid according to node context ($round \leq r$).
- *uint*64 *period*, a period that is valid according to node context ($period \leq p$).
- *uint*8 *step*, a step to run sortition. Used to get specific step-dependant committee size and threshold.

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. The function starts by retrieving a sortition seed Q. This seed will be a tamper-proof source of randmoness committed as part of a block header δ_s rounds in the past. Then, on line 3 it computes the sortition proof π by running $\mathsf{VRF_p rove}(.)$, using the secret VRF key from user I, who should have a valid VRF public/private key pair for $\delta_s\delta_r$ rounds in past. Algorithmically, every monetary unit the user has is considered a "sub-user", and then each one of them is selected with probability $t = \frac{\tau}{W}$, where τ is the expected amount of users to be selected for the given role, and W is the total stake online for the relevant round. The semi-open interval $[0, 1)$ is then split into consecutive intervals using an accumulated binomial distribution, and wherever the fraction $\frac{sh}{2^{\lfloor \log_2(sh) \rfloor}}$ belongs to the interval, that's the amount of chosen sub-users for the subroutine caller.

Returns:

an object of type *credentials*, containing the sortition hash and proof (output of VRF computation) and an unsigned integer j representing the weight that player has in the committee, for the desired round, period and step.

Algorithm 34 Verify Sortition

```
1: function VerifySortition(I, π, round, period, step)
2:     Q := L[r − δ_s]_Q
3:     I_VRFKey := BT[round − δ_s δ_r][I]_VRFKey
4:     success := VRF_verify(π, Q||round||period||step, I_VRFKey)
5:     if ¬success then return 0
6:
7:     h := VRF_proofToHash(π)
8:     τ := CommitteeSize(step)
9:     w := BT[round − δ_s δ_r][I]_balance
10:    W := getTotalOnlineStake(round − δ_s δ_r)
11:    t := τ / W
12:    j := 0
13:    while sh / 2^(log2(h)) ∉ [Σ_{k=0}^{j} B(k; w, t), Σ_{k=0}^{j+1} B(k; w, t)) do
14:        j := j + 1
       return j
```

Arguments:

- I, a user's address
- Q, the 32 byte sortition seed to be used
- π, an 80 byte VRF proof
- *round*, the round number to be used inside the VRF function
- *period*, the period to be used inside the VRF function
- *step*, the step to be used for committee size and the VRF function

Description:
The sortition verification procedure takes the address of an account, a valid sortition seed to be used as a source of entropy, a VRF proof π obtained from running $\mathsf{VRF}_{\mathsf{prove}}$ and a thruple representing the consensus context of a node (*round*, *period* and *step*) 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 for the relevant *round*.

Returns:
an integer j that will be positive (larger than 0) if the user has been selected, and it's size corresponds to the participation weight for a given committee member. If the user I is not chosen as a committee member for the context passed, a value of $j = 0$ is returned.

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.

- π_{seed}, the 32-byte sortition proof output by running $VRF_{a_{sk}}$ algorithm for a cryptographic seed computation.
- 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_0, time interval for the node to accept block proposals (when $p = 0$), after which it chooses the observed block with the highest priority (lowest hash).
- λ, same as λ_0 but for $p > 0$.
- δ_s, sortition seed renewal rate, in number of rounds. Set to 2 in specs. as of July 2023.
- δ_b, balance lookback interval, in number of rounds. Set to 320 in specs.

References

[Alg23] Algorand Foundation. Algorand transaction execution approval language. Github Repository for the Algorand Foundation, May 2023. `https://github.com/algorandfoundation/specs/releases/tag/abd3d48`.

[Ber06] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *Proceedings of the 9th International Conference on Theory and Practice of Public-Key Cryptography*, PKC'06, page 207–228, Berlin, Heidelberg, 2006. Springer-Verlag.

[Dan12] Quynh Dang. Secure hash standard (shs), 2012-03-06 2012.

[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.

[SM99] Salil Vadhan Silvio Micali, Michael Rabin. Verifiable random functions. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, pages 120–130, 1999.

Index