Algorand Protocol Description

Argimiro, CoinFabrik

September 13, 2023

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.

We relied on Algorand's documents ([GHM+17], [Mic16] and Algorand's official specifications) but mostly we consulted the code for the node (in the official repository) 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. 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 in its 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.

1.1 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 $s := 0 \in \mathbb{Z}_{\geq 0}$, 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 receives a self message with $timer = 0$. Then the algorithm calls the Block Proposal Algorithm (Algorithm 2).

The BlockProposal() algorithm runs, moving directly to the main loop. Said loop iterates over all the accounts that are registered in the node.

For each account a, the node runs Sortition$(a, proposal)$ (Algo. 33) which basically runs a Verifiable Random Function in order to decide whether this account can vote (by picking a random value) and provide a proof that the random value was generated according to protocol. If sortition for a particular account returns the zero value, this means the account has not been selected for block proposal. If after running sortition for every online account managed by the node, no positive integer value is returned, the algorithm exits and the node will no longer be a proposer in the proposal step. When it is nonzero, the node participates in the proposal voting representing the account. In a vanilla run scenario and at this point, the period is zero ($p = 0$). The algorithm performs a block assembly (Algorithm 11), sets v as the proposal value obtained from the vote e and does two separate broadcasts for Vote$(a_I, r, p, proposal, v, cred)$ and e. Also, we enter the "soft vote" step setting $s := 1$.

Assume that some time has passed and now $0 < timer <$ FilterTimeout(p), and that the node received a block proposal e' which was broadcasted from another node. Then, the event handler runs HandleProposal(e') (Algorithm 4). This algorithm receives the proposal e' and unpacks its contents, including the execution state (r', p', s'). Per assumptions, we have that $r = r'$ and $p = p' = 0$. The algorithm checks if the proposal is valid, calling VerifyProposal(v') on $v' = Proposal_{value}(e')$ (see Algorithm 20 for more details), it also checks if periods differ ($p \neq p'$), exiting if not. However, both checks pass per 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 $0 < timer <$ FilterTimeout(p) still holds. The event handler for the main algorithm thus calls 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 $vt_s = 0 = proposal$, the algorithm checks if RetrieveProposal$(vt_v) \neq \perp$ and broadcasts this proposal if it is available, ignoring if it is not.

Until $timer \geq$ FilterTimeout(p) the main algorithm will execute the above steps whenever a vote or a proposal are received, always recording new events and this happens.

Eventually, the timer is at $timer =$ FilterTimeout(p) and the main algorithm calls 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 fix notation!

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

and, if $j > 0$ it broadcasts Vote$(r, p, soft, v, credential'')$. Moreover, if the proposal $prop :=$

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 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 ($\sum_{vt \in V} vt.cred_j \geq$ CommitteeThreshold$(soft)$) is bigger than the committee threshold. Imagine that the first time this happens, it cannot form a bundle.

Revisar que este definido y apuntar a esa seccion.

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 Commit(v), set $r_{old} := r$, and calls StartNewRound$(r + 1)$ and GarbageCollect(r_{old}, p) ending the round. Calling the new round algorithm (Algo. 22) 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. 24) 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: $V := V \backslash V_{(r,p-1)}$, $P = P \backslash P_{(r,p-1)}$.

2 Strutcture

The Algorand protocol is implemented in a network over the Internet, where nodes form a peer-to-peer network. More explicitly, a set of whitelisted relay nodes interact with participation nodes where the latter receive transactions and implement the protocol while the former simply relay whatever they receive from the nodes.

A participation node includes:

- 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 virtual machine, the Algorand Virtual Machine, which can process TEAL bytecode etc etc.
- a network interface.
- A XX implementing the consensus algorithm.
- A database engine including at least the following tables:

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 not pass are discarded, and the result is cached by 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 "transfer an asset" to himself 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.

Nodes do have a limit, e.g, they will keep 1,000 transactions and ignore the rest. These transactions are processed by the node as follows. is this ok?

- The node orders the transactions as they arrive.
- The node verifies the signature of the transaction (i.e., the transaction has an address as its source, this address also identifies a public key that may be used to verify the signature); if it is a logicsig, it is verified; if it is an app call, then the run is simulated.
- At some point, the node starts assembling the node. It will now rerun all transactions (but no logicsigs) the transaction is interpreted by the AVM. It will either succeed or fail. If it succeeds, it may alter the ledger and balances for the next block. These changes are recorded. Changes to the balances and ledger are temporary recorded and checked for inconsistencies (e.g., local to the block creation)
- Once a timeout is reached, the node will collect all the changes into a block proposal. e.

During consensus, a node that receives an e it will validate same as above.

Once consensus accepted a block, e, it needs to run all transactions again but now modifies the ledger and balances.

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

2.1 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. Participation nodes can connect only to relay nodes, they run an instance of the Aglorand Virtual Machine (AVM). el voting y demas esta afuera de la avm?

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

Currently, any individual or entity is able to run a participation node, but relay nodes are permissioned. They must be *whitelisted* by Algorand Foundation. validar que es AF quien whitelistea

When a participation node is booted up, 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 node's code. Relay

nodes manage a significantly higher number of connections, as they are responsible for keeping the network's high throughput.

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". That is achieved by sending a special `keyreg` transaction, which registers a participation key for the account and creates a two-level ephemeral key tree. It is important to note that the registered account will only start participating in consensus δ_b rounds after the key registration transaction is approved. Any account participating in consensus does so with their full balance. There is no risk associated to participation, nor there are any rewards for running a node.

TODO: maybe include some voting stats?

2.2 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 signed 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.,

2.3 Keys

Algorand nodes interact with 3 types of keys: Root or spending keys, Voting or participation keys and VRF selection keys.

Generally all keys will be in pairs, in a public / private cryptographic scheme. This scheme supports a signing procedure, which takes a private key and a message and outputs a signature, and a verification procedure, which takes a public key, a message and a signature, and outputs a boolean value. A true output certifies that the signature was indeed produced by signing the relevant message with the private key of the claimed address. We now make a brief description of each of the three key types and their relevance in the protocol.

Spending keys are 32 byte keys used by accounts to sign and verify transactions on behalf of accounts. Nodes interact only with public spending keys, out of which Algorand addresses are derived. Private spending keys are used to sign transactions and should be kept privately by the user managing the relevant account for security concerns. This is, unless the account has undergone a process called "rekeying", in which case the private key of the account designated by the rekeying transaction is the key authorized to sign transactions on behalf of this account.

Participation keys are used to sign all consensus messages. They are organized in a tree scheme. The private keys in the scheme are stored as secrets by the node, while their public keys are a mandatory part of online account records. They correspond to the ed25119 eliptic curve cryptographic standard (TODO: CITAR standard). Previous to online account registration, the node generates a root participation key, a batch of subkeys of size n, and a sub-batch of ephemeral keys of size k, where $k = keyDillution$ (which defaults to the square root of the round range; e.g. for a 3e+06 round range a 1712 sub-batch of ephemeral keys is generated).

according to the key dilution field. A keyreg transaction needs the root public participation key as a valid field. Ephemeral keys are leaves of the two level tree, and are used to sign consensus messages for a given round. They are deleted after the succesful advancement of a round. An

agreement vote message is valid only when it is signed with the correct voting key. When generated, a validity range is specified.

Selection or VRF keys are keys used for committee membership selection and verification. VRF public keys are kept inside account records (in the BT), and are needed in order for an account to vote, as they will be used to verify committee membership by nodes receiving votes. They make use of an Algorand fork of the Sodium C library (citar!), implementing VRF signing, verification and key generation functions. The account's private keys are kept as secrets by the node, and are guaranteed to be non-persistant in memory.

On a key generation command, a node takes an address, a range of rounds and a 64 bit integer key dilution parameter. The round range has a maximum of $2^{24} - 1$, and a recommended value of $3e06$. It then uses a function in the Algorand sodium library fork (LINK TO GITHUB), to produce a public VRF key, and a private VRF key that will be kept as a secret by the node.

Once these are created and associated with the given address, a keyreg transaction will take the account online and have it start participating. An account record will be created for this account (if it did not already exist in the Balance table BT), the VRF public key registered, as well as the public participation key, the created intermediate batch keys and the first sub-batch of ephemeral keys. All private counterparts of these will be kept as secrets by the node and used extensively in succesive voting rounds, starting δ rounds after the keyreg transaction is committed inside a block (currently $\delta = 320$).

2.4 Node as a State Machine

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

Online accounts are registered to a node. This node takes care of voting for these accounts. Hence, the state for these accounts, is the state of the node. As many accounts may be registered in one node, the node will manage the participation for all of these accounts concurrently.

The state of a node is given by the execution state which is defined as the 3-tuple of nonnegative integers (r, p, s), a set of observed proposals, P, a set of observed votes, V, a Ledger L and a balance table BT. Explicitly, we define the state of a node, S, as a tuple of values,

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

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, and increases by one after each block is accepted. revisar wording
- $p \in \mathbb{Z}_{\geq 0}$ is the current period within the round and recorded as a 64-bits integer. When the round starts, p is set to 0 ($p = 0$), and it is increased via the function (F) (see **??**).
- $s \in \mathbb{Z}, 0 \leq s \leq 256$ is the current step within the round and recorded as a 64-bits integer. When the round starts, s is set to 0 ($s = 0$), and it is increased by the main algorithm (see 1) and labeled according to table (see (1)).
- V is the set of all valid observed votes and equivocation votes in the current execution state $((r, p, s))$.
- P is the set of all valid observed proposals in the current execution state $((r, p, s))$.
- $\bar{v}$ is the pinned proposal value.

- $\bar{s} \in \mathbb{Z}_{\geq 0}$, a 64-bits integer, is the last finished step..
- L is the ledger, a structure that holds a sequence of states comprising the common information established by the current instantiation of the Algorand protocol. [1].
- BT is the current balance table that includes 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 minumum ALGO balance for this), the global storage for this account–in case it belongs to a smart contract, the consensus participation status of this account (e.g., if this account is online and voting or not), in case the account is online and voting, it also includes the public VRF key (selection key), a public participation key, a set of public batch participation keys, and a set of public sub-batch ephemeral keys.
- TP is the node's transaction pool.
- The current time `time` in seconds, and the elapsed time since the last time reset was called.

The first 7 values are the consensus state parameters, whereas L and BT are ledger state parameters.

Let r and p be unsigned 64-bits integers representing the round and period number respectively, and let s be an unsigned 8 bits integer representing the step number. For convenience and readability, we define the following step name - step enumeration assignment:

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

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

- Genesis identifier and genesis hash, unambiguously defining the ledger to which the state belongs.
- Protocol version and update state.
- Timestamp (in milliseconds since genesis block), identifying when the state was first proposed.
- 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.

A balance table, BT, is a mapping of addresses to account records where an account record comprises a 64-bits unsigned integer raw balance, a 3-state flag status (can be "online", "offline", or "non-participating"), its registered voting keys (more info on subsection **??**, keys) and two

64-bits unsigned integers r_{start} and r_{last}, which represent the validity interval (in rounds) for the participation key (which acts as the root voting key for the Merkle tree).

Pero la raíz del merkle tree donde esta definida?

A node may query the current state of a particular account by indexing the balance table by address, or it might query a previous round's account state by adding a round indexing. Note that past account states are always computable by iterating over the Ledger sequence of blocks and reverting transaction updates. Internally however, certain past records might be cached and maintained to improve performance.

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

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

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

- The block's round, which matches the round of the state it is transitioning into. (The block with round 0 is special in that this block specifies not a transition but rather the entire initial state, which is called the genesis state. This block is correspondingly called the genesis block).
- The block's genesis identifier and genesis hash, constant and decided on Ledger creation.
- The block's upgrade vote, which results in the new upgrade state. The block also duplicates the upgrade state of the state it transitions into.
- The block's timestamp, which matches the timestamp of the state it transitions into. The timestamp is decided by the proposer of the block using their own internal clock. To be valid, a timestamp must be greater than the last committed block's timestamp and must not be more than 25 seconds away from said time.
- The block's seed, which matches the seed of the state it transitions into.
- The block's reward updates, which results in the new reward state. The block also duplicates the reward state of the state it transitions into.
- A cryptographic vector commitment, using SHA512/256 hash function, to the block's transaction sequence.
- A cryptographic vector commitment, using SHA256 hash function, to the block's transaction sequence (useful for compatibility with systems where SHA512/256 is not supported).
- The block's previous hash, which is the cryptographic hash of the previous block in the sequence. (The previous hash of the genesis block is 0).

cual de los dos hashes? sha256 o sha512/256?

- The block's transaction counter, which is the total number of transactions issued prior to this block. This count starts from the first block with a protocol version that supported the transaction counter.

[1]Note that, thanks to the genesis hash and previous hash components, a ledger state unambiguously defines the whole history of state changes.

- The block's expired participation accounts, which contains an optional slice of public keys of accounts. These accounts are expected to have their participation key expire by the end of the round (or was expired before the current round). The block's expired participation accounts slice is valid as long as the participation keys of all the accounts in the slice are expired by the end of the round or were expired before, the accounts themselves would have been online at the end of the round if they were not included in the slice, and the number of elements in the slice is less or equal to 32. A block proposer may not include all such accounts in the slice and may even omit the slice completely

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

los account records no tienen states!

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

A Proposal 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 block creator, the sole account that computed the sortition seed and proof.

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

$Proposal_{value}(I, p, prop) := (I, p, H(prop), H(Encoding(prop))$ is the function used by a block proposer to create a proposal-value.

For convenience, we define the function $\mathsf{RetrieveProposal} : v \mapsto \mathsf{RetrieveProposal}(v)$ for a given proposal value v, such that $\mathsf{RetrieveProposal}(v) = prop$ if, and only if, $prop$ is the proposal that, when hashed, corresponds to the proposal hash $H(prop)$ of the proposal value v.

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

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

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

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 is the vote's round (64-bit integer).
- p is the vote's period (64-bit integer).
- s is the vote's step (8-bit integer).
- v is the vote's proposal value.
- $cred$, of type *credentials*, the committee credentials of the voter.

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

Let I be an address, $r, s, p \in \mathbb{Z}_{\geq 0}$, $cred$ a credential, and v_0, v_1 different vote proposals. An equivocation vote happens when a pair of votes $vt_0 = \mathsf{Vote}(I, r, p, s, v_0, cred)$ and $vt_1 = \mathsf{Vote}(I, r, p, s, v_1, cred)$. An equivocation vote is valid if both of its constituent votes are valid. It is important to keep in mind that nodes are forbidden from equivocating on any steps $s \neq soft$.

What does this mean? How do nodes enforce this?

A bundle is a set $b = \mathsf{Bundle}(r, p, s, v)$ such that for any pair of distinct values $vt_0, vt_1 \in b$ the following conditions hold: a) $vt_{0_r} = vt_{1_r}$, b) $vt_{0_p} = vt_{1_p}$, c) $vt_{0_s} = vt_{1_s}$, d) for each pair of distinct elements, either $(vt_0)_I \neq (vt_1)_I \wedge vt_0.v = vt_1.v$ or the pair vt_0, vt_1 is an equivocation vote, and e) 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.

We define the observed bundles for the current execution state, B, as the set of all of the bundles observed by the node in the current execution state.

3 Main algorithm

The node is reactive. The main algorithm for the node, thus, starts when it receives any input, it makes computations –probably modifying the state of the node– and may end by broadcasting data to other nodes.

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 is of type Proposal p then
17:             HandleProposal(p)
18:         else if msg is of type Vote v then
19:             HandleVote(v)
20:         else if msg 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 ($Bundle(r, p, s, v)$ 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 $FilterTimeout(p)$ 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).

3.1 Block Proposal

Algorithm 2 Block Proposal

```
1: function BlockProposal( )
2:     ResynchronizationAttempt()
3:     for a ∈ A do
4:         cred := Sortition(a, 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 4), 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 $cred_j = \mathsf{Sortition}(a, proposal) + j > 0$) 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.

3.2 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, 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 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., $V* = \{vt' \in V : vt'_s = proposal\}$. 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 Sortition() inside the *cred* credentials structure), the node hashes the concatenation of sh and i, H($sh||i$) (where 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.

3.3 HandleProposal

Algorithm 4 HandleProposal

```
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
7:          return //ignore proposal
8:      if v ∉ {σ, v̄, μ} then
9:          return //ignore proposal
10:     Relay(prop)
11:     P := P ∪ e
12:     if IsCommitable(v) ∧ s ≤ cert then
13:         for a ∈ A do
14:             cred := Sortition(a, cert)
15:             if cred_j > 0 then
16:                 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.

Este request está en el agoritmo? Línea?

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.

3.4 HandleVote

Algorithm 5 HandleVote

```
1: function HandleVote(Vote vt)
2:     if not VerifyVote(vt) then
3:         PenalizePeer(SENDER_PEER(vt))  //optional
4:         return  //ignore invalid vote
5:     if vt_s = 0 ∧ (vt ∈ V ∨ IsEquivocation(vt)) then
6:         return  //ignore vote, equivocation not allowed in proposal votes
7:     if vt_s > 0 ∧ IsSecondEquivocation(vt) then
8:         return  //ignore vote if it's a second equivocation
9:     if vt_r < r then
10:        return  //ignore vote of past round
11:    if vt_r = r + 1 ∧ (vt_p > 0 ∨ vt_s ∈ {next_0, ..., next_249}) then
12:        return  //ignore vote of next round if non-zero period or next_k step
13:    if vt_r = r ∧ (vt_p ∉ {p − 1, p, p + 1}∨
14: (vt_p = p + 1 ∧ vt_s ∈ {next_1, ..., next_249})∨
15: (vt_p = p ∧ vt_s ∈ {next_1, ..., next_249} ∧ vt_s ∉ {s − 1, s, s + 1})∨
16: (vt_p = p − 1 ∧ vt_s ∈ {next_1, ..., next_249} ∧ vt_s ∉ {s̄ − 1, s̄, s̄ + 1})) then
17:        return  //ignore vote
18:    V := V ∪ vt  //observe vote
19:    Relay(vt)
20:    if vt_s = proposal then
21:        if RetrieveProposal(vt_v) ≠ ⊥ then
22:            Broadcast(RetrieveProposal(vt_v))
23:    else if vt_s = soft then
24:        if ∃v|Bundle(vt_r, vt_p, soft, v) ⊂ V then
25:            for a ∈ A do
26:                cred := Sortition(a_sk, cert)
27:                if cred_j > 0 then
28:                    Broadcast(Vote(a_I, r, p, cert, v, cred))
29:    else if vt_s = cert then
30:        if ∃v|Bundle(vt_r, vt_p, cert, v) ⊂ V then
31:            if RetrieveProposal(v) = ⊥ then
32:                RequestProposal(v) //waits or keeps playing without voting power
33:            Commit(v)
34:            r_old := r
35:            StartNewRound(vt_r + 1)
36:            GarbageCollect(r_old, p)
37:    else if vt_s > cert then
38:        if ∃v|Bundle(vt_r, vt_p, vt_s, v) ⊂ V then
39:            p_old := p
40:            StartNewPeriod(vt_p + 1)
41:            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 penalize the sender of the vote (by disconnecting or blacklisting it, for example). Equivocation votes on a proposal step are not allowed, so a check for this condition is performed on line 5. Furthermore, second equivocations are never allowed (line 7). Any votes for rounds prior to the current round are discarded (line 9). On the special case that we received a vote for a round immediately after the current round, we observe it only if 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.

3.5 HandleBundle

Algorithm 6 HandleBundle

labelalgo:habdle-bundle

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

Description:
The bundle handler is invoked whenever a bundle message is received. If the received bundle is invalid (line 2), 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 $b\prime = Bundle(b.r, b.p, b.s, v\prime)$ 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 $b.r < r$.

3.6 Recovery Attempt

Algorithm 7 Recovery

```
1: function Recovery
2:     ResynchronizationAttempt()
3:     for Account a ∈ A do
4:         cred := Sortition(a_sk, 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 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.

3.7 Fast Recovery Attempt

Algorithm 8 FastRecovery

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

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

4 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:         returnSHA512/256(in)
4:     else
5:         returnSHA256(in)
```

Arguments:

- *Hashable in* = some hashable data (plain bytes)

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

Returns:

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

Algorithm 11 AssembleBlock

```
1: function AssembleBlock(Address I)
2:     Block b
3: //define block header
4:     b_r := r
5:     b_prevHash := H(L[r − 1])
6:     (Q, π_seed) := ComputeSeedAndProof(I)
7:     b_Q := Q
8:     b_txnCommitment := MerkleTree(TP[−s])_root
9:     b_timestamp := TIMESTAMP() //timestamp in seconds since epoch //TODO: timestamp
       must be inside an interval. Validate this with code in block validation
10:    b_GenesisID := L_genesisID
11:    b_GenesisHash := H(L[0])
         //should we include reward state, update vote and update state?
12:    b_txnCounter := NEXTTXNCOUNTER
13:    b_stateProof := StateProofTrackers
14:    b_partUpdates := {a ∈ G ∫ G′}
15:
16: //define block body, ordered set of signed transactions
17:    while Time() < Deadline do
18:        txn := TP.pop()
19:        if Valid(b_payset, txn) then
20:            b_payset := b_payset||txn
21:
22:    return (b, π_seed)
```

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

Algorithm 12 VerifyBlock

1: **function** VerifyBlock($Block\ b, \pi_{seed}, p_{orig}, I_{orig}$)
2: **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 (RetrieveProposal(v) $\neq \perp$) $\wedge$ (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: $L := L||(\text{RetrieveProposal}(v))_e$
3: UpdateBT((RetrieveProposal(v))$_e$)

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 RetrieveProposal(v). Afterwards, it updates the balance table BT with all state changes called for the committed entry. The function 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

```
1: function VerifyVote(Vote vt)
2:     valid := vt_r ≤ r + 2
3:     if vt_s = 0 then
4:         valid := valid ∧ vt.v.p_orig ≤ vt_p
5:         if vt_p = vt.v.p_orig then
6:             valid := valid ∧ vt.v.I_orig = vt_I
7:     if vt_s ∈ {propose, soft, cert, late, redo} then
8:         valid := valid ∧ vt.v ≠ ⊥
9:     else if vt_s = down then
10:        valid := valid ∧ vt.v = ⊥
11:    valid := valid ∧ VerifyPartSignature(vt)
12:    valid := valid ∧ VerifyVRF(vt_π, vt_I, vt_r, vt_p, vt_s)
13:    return valid
```

Description:
Vote verification gets an unauthenticated vote and returns an authenticated vote and a boolean indicating whether authentication succeeded or failed.

Algorithm 16 VerifyVote

```
1: function VerifyPartSignature(Vote vt)
       //get voter's relevant ephemeral key for this round
2:     x
       //verify key's validity interval
3:     x
       //use the cryptographic signature verification function
4:     return ed25519_verify(part_pk, encode(vt))
```

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 17 Proposal

```
1: function RetrieveProposal(Proposal_value v)
2:     if ∃prop ∈ P | Proposal_value(prop) = v then
3:         return prop
4:     else
5:         return ⊥
```

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 18 Make Proposal

1: **function** MakeProposal($Block\ e, \pi_{seed}, uint64\ p_{orig}, Address\ I_{orig}$)
2: $proposal\ prop := (e, \pi_{seed}, p_{orig}, I_{orig})$
3: 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 19 Proposal-value

1: **function** $\mathsf{Proposal}_{\mathsf{value}}(Proposal\ e)$
2: $proposal_{value}\ v := (e.I_{orig}, e.p_{orig}, \mathsf{H}(e), \mathsf{H}(\mathsf{Encoding}(e)))$
3: 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 20 VerifyProposal

1: **function** VerifyProposal($ProposalPayload\ pp$)
2: $valid := \mathsf{ValidEntry}(pp.e, L)$
3: $valid := valid \wedge \mathsf{VerifySignature}(pp.y)$

Description:

Algorithm 21 VerifyBundle

1: **function** VerifyBundle($Bundle\ b$)
//all individual votes are valid
2: $valid := (\forall vt \in b)(VerifyVote(vt))$
//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: $valid := valid \wedge (\forall vt \in b)(vt.v = b.v \vee (b.s = soft \wedge IsEquivocation(vt.v, b)))$
//summation of weights should surpass the relevant threshold
6: $valid := valid \wedge \sum_{vt \in b}(vt.w) \geq CommitteeThreshold(b.s)$
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 aforementioned 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 22 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 23 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). Finally, 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 24 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 25 Request Proposal

1: **function** RequestProposal($Proposal_{value}$ v) TBD

Description:

ver como hacer el request...como lo resuelve el nodo?

Algorithm 26 Penalize Peer

1: **function** DisconnectFromPeer($PEER\_ID$)
2: Remove $PEER\_ID$ from available connections

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 its 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 27 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 28 Compute Seed And Proof

```
1: function ComputeSeedAndProof(Account a)
2:     if p = 0 then
3:         y := VRF.Prove(L[r − δ_s]_Q, a_sk)
4:         α := H(VRF.ProofToHash(y), a_I)
5:     else
6:         y := 0
7:         α := H(L[r − δ_s]_Q)
8:     if r mod δ_s δ_r < δ_s then
9:         Q := H(α||H(L[r − δ_s δ_r]))
10:    else
11:        Q := H(α)
12:    return (Q, y)
```

Arguments:

- *Account I*

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 29 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) then
5:              return False
6:          q_1 := H(VRF.ProofToHash(π)||I)
7:      else
8:          q_1 := H(q_0)
9:      if r ≡ (r mod δ_s) mod δ_s δ_r then
10:         return Q = H(q_1||H(L[r − δ_s δ_b]))
11:     else
12:         return Q = q_1
```

Arguments:

- *Account* I

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 30 Get Sortition Seed

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

Arguments:

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

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

Returns:

- a sortition seed to be used in the round r

Algorithm 31 getSortitionWeight

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

Arguments:

- ctx = a helper structure to retrieve ledger context information (e.g. the last confirmed block)
- r = current round number
- a_{pk} = the account's public key for the lookup 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 32 getTotalOnlineStake

```
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:

- $uint64$ 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 34 Sortition

1: **function** Sortition($Account\ a, uint64\ step$)
2: $Q \leftarrow L[r-\delta_s]_Q$ //ACLARAR QUE ESTE BALANCE ES EL DEL LOOKBACK (NO EL ACTUAL)
3: $w := a.balance$ //$BT[a.address][r-\delta_s-\delta_b].balance$
4: $W := \sum_{a \in A_{r-\delta_s-\delta_b}} a.balance$
5: $\tau :=$ CommitteeThreshold(p)
6: $\langle hash, \pi \rangle :=$ VRF$_{\mathsf{a_{sk}}}(Q||step)$
7: $t := \frac{\tau}{W}$
8: $j := 0$
9: **while** $\frac{hash}{2^{hashlen}} \notin [\sum_{k=0}^{j} \mathsf{B}(k; w, t), \sum_{k=0}^{j+1} \mathsf{B}(k; w, t))$ **do**
10: $j := j + 1$
return $credentials(hash, \pi, j)$

Arguments:

- *Account* a, an online account's balance table record
- $uint64$ *step*, the step to run sortition for

Description:

A lot of notation to describe. Review.

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, 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 $\frac{hash}{2^{hashlen}}$ 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 35 VerifySortition

1: **function** VerifySortition($I, Q, \tau, round, period, step, w, W$)
2: $I_{VRFKey} := BT[round - \delta_s\delta_r][I]_{VRFKey}$
3: $success :=$ VRF$_{\mathsf{verify}}(\pi, Q||round||period||step, I_{VRFKey})$
4: **if** $\neg success$ **then** return 0
5: $hash :=$ VRF$_{\mathsf{proofToHash}}(\pi)$
6: $\tau :=$ CommitteeSize($step$)
7: $W :=$ getTotalOnlineStake($round - \delta_s\delta_r$)
8: $t := \frac{\tau}{W}$
9: $j := 0$
10: **while** $\frac{hash}{2^{hashlen}} \notin [\sum_{k=0}^{j} \mathsf{B}(k; w, t), \sum_{k=0}^{j+1} \mathsf{B}(k; w, t))$ **do**
11: $j++$
return j

Arguments:

- pk = a user's public key (their address)
- Q = 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

MISSING VRF(), VerifyVRF()

5 Appendix: Notation

- *credentials*, a data structure containing the results of running the sortition algorithm for a specified account
- sh, the 64-byte sortition hash output by running $VRF_{a_{sk}}$ algorithm over a desired input. Usually wrapped inside a *credentials* structure.
- π, the 32-byte sortition proof output by running $VRF_{a_{sk}}$ algorithm over a desired input. Usually wrapped inside a *credentials* structure.
- j, an unsigned 64-bit integer, representing the weight of a given account's vote inside a specific committee (for a given round, period and step). Usually wrapped inside a *credentials* structure.
- λ_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`.

[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.

Index