Algorand Byzantine Fault Tolerance Protocol Specification

May 30, 2019

Abstract

The *Algorand Byzantine Fault Tolerance protocol* is an interactive protocol which produces a sequence of common information between a set of participants.

Contents

1 Conventions and Notation

This specification defines a *player* to be a unique participant in this protocol.

This specification describes the operation of a single *correct* player. A correct player follows this protocol exactly and is distinct from a *faulty* player. A faulty player may deviate from the protocol in any way, so this specification does not describe the behavior of those players.

Correct players do not follow distinct protocols, so this specification describes correct behavior with respect to a single, implicit player. When the protocol must describe a player distinct from the implicit player (for example, a message which originated from another player), the protocol will use subscripts to distinguish different players. Subscripts are omitted when the player is unambiguous. For instance, a player might be associated with some "address" I; if this player is the kth player in the protocol, then this address may also be denoted I_k.

This specification will describe certain objects as *opaque*. This document does not specify the exact implementation of opaque objects, but it does specify the subset of properties required of any implementation of some opaque object.

Opaque data definitions and semantics may be specified in other documents, which this document will cite when available.

All integers described in this document are unsigned.

2 Parameters

The protocol is parameterized by the following constants:

- $\lambda, \lambda_f, \Lambda$ are values representing durations of time.
- δ_s, δ_r are positive integers (the "seed lookback" and "seed refresh interval").

For convenience, we define δ_b (the "balance lookback") to be $2\delta_s\delta_r$.

Algorand v1 sets $\delta_s = 2$, $\delta_r = 80$, $\lambda = 4$ seconds, $\lambda_f = 5$ minutes, and $\Lambda = 17$ seconds.

3 Identity, Authorization, and Authentication

A player is uniquely identified by a 256-bit string I called an *address*.

Each player owns exactly one *participation keypair*. A participation keypair consists of a *public key* pk and a *secret key* sk. A keypair is an opaque object which is defined in the specification of participation keys in Algorand.

Let m, m' be arbitrary sequences of bits, $B_k, \bar{B}$ be 64-bit integers, $\tau, \bar{\tau}$ be 32-bit integers, and Q be a 256-bit string. Let $(\mathrm{pk}_k, \mathrm{sk}_k)$ be some valid keypair.

A secret key supports a *signing* procedure

$$y := \mathrm{Sign}(m, m', \mathrm{sk}_k, B_k, \bar{B}, Q, \tau, \bar{\tau})$$

where y is opaque and are cryptographically resistant to tampering, where defined. Signing is not defined on many inputs: for any given input, signing may fail to produce an output.

The following functions are defined on y:

- *Verifying*: $\mathrm{Verify}(y, m, m', \mathrm{pk}_k, B_k, \bar{B}, Q, \tau, \bar{\tau}) = w$, where w is a 64-bit integer called the *weight* of y. $w \neq 0$ if and only if y was produced by signing by sk_k (up to cryptographic security). w is uniquely determined given fixed values of $m', \mathrm{pk}_k, B_k, \bar{B}, Q, \tau, \bar{\tau}$.
- *Comparing*: Fixing the inputs $m', \bar{B}, Q, \tau, \bar{\tau}$ to a signing operation, there exists a total ordering on the outputs y. In other words, if $f(\mathrm{sk}, B) = \mathrm{Sign}(m, m', \mathrm{sk}, B, \bar{B}, Q, \tau, \bar{\tau}) = y$, and $S = \{(\mathrm{sk}_0, B_0), (\mathrm{sk}_1, B_1), \ldots, (\mathrm{sk}_n, B_n)\}$, then $\{f(x) | x \in S\}$ is a totally ordered set. We write that $y_1 < y_2$ if y_1 comes before y_2 in this ordering.
- *Generating Randomness*: Let y be a valid output of a signing operation with sk_k. Then $r = \mathrm{Rand}(y, \mathrm{pk}_k)$ is defined to be a pseudorandom 256-bit integer (up to cryptographic security). r is uniquely determined given fixed values of $m', \mathrm{pk}_k, B_k, \bar{B}, Q, \tau, \bar{\tau}$.

The signing procedure is allowed to produce a nondeterministic output, but the functions above must be well-defined with respect to a given input to the signing procedure (e.g., a procedure that implements Verify(Sign(...)) always returns the same value).

4 The Ledger of Entries

An *entry* is a pair $e = (o, Q)$ where o is some opaque object, and Q is a 256-bit integer called a *seed*.

The following functions are defined on e:

- *Encoding*: $\text{Encoding}(e) = x$ where x is a variable-length bitstring.
- *Summarizing*: $\text{Digest}(e) = h$ where h is a 256-bit integer. (h should be a cryptographic commitment to the contents of e.)

A *ledger* is a sequence of entries $L = (e_1, e_2, \ldots, e_n)$. A *round* r is some 64-bit index into this sequence.

The following functions are defined on L:

- *Validating*: $\text{ValidEntry}(L, o) = 1$ if and only if o is *valid with respect to* L. This validity property is opaque.
- *Seed Lookup*: If $e_r = (o_r, Q_r)$, then $\text{Seed}(L, r) = Q_r$.
- *Record Lookup*: $\text{Record}(L, r, I_k) = (\text{pk}_{k,r}, B_{k,r})$ for some address I_k, some public key $\text{pk}_{k,r}$, and some 64-bit integer $B_{k,r}$.
- *Digest Lookup*: $\text{DigestLookup}(L, r) = \text{Digest}(e_r)$.
- *Total Stake Lookup*: $\text{Stake}(L, r) = \sum_k \text{Record}(L, r, I_k)$.

A ledger may support an opaque *entry generation* procedure

$$o := \text{Entry}(L, Q)$$

which produces an object o for which $\text{ValidEntry}(L, o) = 1$.

5 Messages

Players communicate with each other by exchanging *messages*.

5.1 Elementary Data Types

A *period* p is a 64-bit integer.

A *step* s is an 8-bit integer. Certain steps are named for clarity. These steps are defined as follows:

- *propose* $= 0$
- *soft* $= 1$
- *cert* $= 2$
- *late* $= 253$

- $redo = 254$
- $down = 255$
- $next_s = s + 3$

The following functions are defined on s:

- *Committee Size*: CommitteeSize(s) is a 64-bit integer defined as follows:

$$\text{CommitteeSize}(s) = \begin{cases} 9 & : s = propose \\ 2990 & : s = soft \\ 1500 & : s = cert \\ 500 & : s = late \\ 2400 & : s = redo \\ 6000 & : s = down \\ 5000 & : \text{otherwise} \end{cases}$$

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

$$\text{CommitteeThreshold}(s) = \begin{cases} 0 & : s = propose \\ 2267 & : s = soft \\ 1112 & : s = cert \\ 320 & : s = late \\ 1768 & : s = redo \\ 4560 & : s = down \\ 3838 & : \text{otherwise} \end{cases}$$

A *proposal-value* is a tuple $v = (I, p, \text{Digest}(e), \text{Hash}(\text{Encoding}(e)))$ where I is an address (the "original proposer"), p is a period (the "original period"), and Hash is some cryptographic hash function. The special proposal where all fields are the zero-string is called the bottom proposal $\perp$.

5.2 Votes

Let I be an address, r be a round, p be a period, s be a step, and v be a proposal-value, let x be a canonical encoding of the 5-tuple (I, r, p, s, v), and let x' be a canonical encoding of the 4-tuple (I, r, p, s). Let y be an arbitrary bitstring. Then we say that the tuple

$$(I, r, p, s, v, y)$$

is a *vote from I for v at round r, period p, step s* (or *a vote from I for v at* (r, p, s)), denoted $\text{Vote}(I, r, p, s, v)$.

Moreover, let L be a ledger where $|L| \geq \delta_b$. Let (sk, pk) be a keypair, $B, \bar{B}$ be 64-bit integers, Q be a 256-bit integer, and $\tau, \bar{\tau}$ 32-bit integers. We say that this vote is *valid with respect to L* (or simply *valid* if L is unambiguous) if the following conditions are true:

- $r \leq |L| + 2$
- Let $v = (I_{orig}, p_{orig}, d, h)$. If $s = 0$, then $p_{orig} \leq p$. Furthermore, if $s = 0$ and $p = p_{orig}$, then $I = I_{orig}$.
- If $s \in \{propose, soft, cert, late, redo\}$, $v \neq \perp$. Conversely, if $s = down$, $v = \perp$.
- Let $(\text{pk}, B) = (\text{Record}(L, r - \delta_b), I)$, $\bar{B} = \text{Stake}(L, r - \delta_b)$, $Q = \text{Seed}(L, r - \delta_s)$, $\tau = \text{CommitteeThreshold}(s)$, and $\bar{\tau} = \text{CommitteeSize}(s)$.
  Then $\text{Verify}(y, x, x', \text{pk}, B, \bar{B}, Q, \tau, \bar{\tau}) \neq 0$.

Observe that valid votes contain outputs of the Sign procedure; i.e., $y := \text{Sign}(x, x', \text{sk}, B, \bar{B}, Q, \tau, \bar{\tau})$.

Informally, these conditions check the following:

- The vote is not too far in the future for L to be able to validate.
- "Propose"-step votes can either propose a new proposal-value for this period ($p_{orig} = p$) or claim to "re-propose" a value originally proposed in an earlier period ($p_{orig} < p$). But they can't claim to "re-propose" a value from a future period. And if the proposal-value is new ($p_{orig} = p$) then the "original proposer" must be the voter.
- The *propose*, *soft*, *cert*, *late*, and *redo* steps must vote for an actual proposal. The *down* step must only vote for $\perp$.
- The last condition checks that the vote was properly signed by a voter who was selected to serve on the committee for this round, period, and step, where the committee selection process uses the voter's stake and keys as of δ_b rounds before the vote.

An *equivocation vote pair* or *equivocation vote* $\text{Equivocation}(I, r, p, s)$ is a pair of votes which differ in their proposal values. In other words,

$$\begin{aligned}\text{Equivocation}(I, r, p, s) = (&\text{Vote}(I, r, p, s, v_1),\\ &\text{Vote}(I, r, p, s, v_2))\end{aligned}$$

for some $v_1 \neq v_2$.

An equivocation vote pair is *valid with respect to* L (or simply *valid* if L is unambiguous) if both of its constituent votes are also valid with respect to L.

5.3 Bundles

Let V be any set of votes and equivocation votes. We say that V *is a bundle for* v *in round* r*, period* p*, and step* s (or a *bundle for* v *at* (r, p, s)), denoted $\text{Bundle}(r, p, s, v)$.

Moreover, let L be a ledger where $|L| \geq \delta_b$. We say that this bundle is *valid with respect to* L (or simply *valid* if L is unambiguous) if the following conditions are true:

- Every element $a_i \in V$ is valid with respect to L.
- For any two elements $a_i, a_j \in V$, $I_i \neq I_j$.
- For any element $a_i \in V$, $r_i = r, p_i = p, s_i = s$.
- For any element $a_i \in V$, either a_i is a vote and $v_i = v$, or a_i is an equivocation vote.
- Let w_i be the weight of the signature in a_i. Then $\sum_i w_i \geq \text{CommitteeThreshold}(s)$.

5.4 Proposals

Let $e = (o, s)$ be an entry and y be the output of a Sign procedure. The pair (e, y) is a *proposal* or a *proposal payload*.

Moreover, let L be a ledger where $|L| \geq \delta_b$, and let $v = (I, p, h, x)$ be some proposal-value. We say that this proposal is *a valid proposal matching* v *with respect To* L (or simply that this proposal *matches* v if L is unambiguous) if the following conditions are true:

- $\text{ValidEntry}(L, e) = 1$.
- $h = \text{Digest}(e)$.
- $x = \text{Hash}(\text{Encoding}(e))$.
- The seed s and seed proof are valid as specified in the following section.
- Let $(B, \text{pk}) = \text{Record}(L, r - \delta_b, I)$. If $p = 0$, then $\text{Verify}(y, Q_0, Q_0, \text{pk}, 0, 0, 0, 0, 0) \neq 0$.

If e matches v, we write $e = \text{Proposal}(v)$.

5.5 Seed

Informally, the protocol interleaves δ_s seeds in an alternating sequence. Each seed is derived from a seed δ_s rounds in the past through either a hash function or through a VRF, keyed on the entry proposer. Additionally, every δ_r rounds, the digest of a previous entry (specifically, from round $r - \delta_r$) is hashed into the result. The seed proof is the corresponding VRF proof, or 0 if the VRF was not used.

More formally, suppose I is a correct proposer in round r and period p. Let $(B, \text{pk}) = \text{Record}(L, r - \delta_b, I)$ and sk be the secret key corresponding to pk. Let

α be a 256-bit integer. Then I computes the seed proof y for a new entry as follows:

- If $p = 0$:
  - $y = \text{VRF.Prove}(\text{Seed}(L, r - \delta_s), \text{sk})$
  - $\alpha = \text{Hash}(\text{VRF.ProofToHash}(y), I)$
- If $p \neq 0$:
  - $y = 0$
  - $\alpha = \text{Hash}(\text{Seed}(L, r - \delta_s))$

Now I computes the seed Q as follows:

$$Q = \begin{cases} H(\alpha, \text{DigestLookup}(L, r - \delta_s\delta_r)) & : (r \bmod \delta_s\delta_r) < \delta_s \\ H(\alpha) & : \text{otherwise} \end{cases}$$

The seed is valid if the following verification procedure succeeds:

1. Let $(B, \text{pk}) = \text{Record}(L, r - \delta_b, I)$; let $q_0 = \text{Seed}(L, r - 1)$.
2. If $p = 0$, check $\text{VRF.Verify}(y, q_0, \text{pk})$, immediately returning failure if verification fails. Let $q_1 = \text{Hash}(\text{VRF.ProofToHash}(y), I)$ and continue to step 4.
3. If $p \neq 0$, let $q_1 = \text{Hash}(q_0)$. Continue.
4. If $r \equiv (r \bmod \delta_s) \mod \delta_r\delta_s$, then check $Q = \text{Hash}(q_1, \text{DigestLookup}(L, r - \delta_s\delta_r))$. Otherwise, check $Q = q_1$.

Note: Round r leader selection and committee selection both use the seed from $r - \delta_s$ and the balances / public keys from $r - \delta_b$.

Note: For reproposals, the period p used in this section is the *original* period, not the reproposal period.

6 State Machine

This specification defines the Algorand agreement protocol as a state machine. The input to the state machine is some serialization of events, which in turn results in some serialization of network transmissions from the state machine.

We can define the operation of the state machine as transitions between different states. A transition N maps some initial state S_0, a ledger L_0, and an event e to an output state S_1, an output ledger L_1, and a sequence of output network transmissions $\mathbf{a} = (a_1, a_2, \ldots, a_n)$. We write this as

$$N(S_0, L_0, e) = (S_1, L_1, \mathbf{a}).$$

If no transmissions are output, we write that $\mathbf{a} = \epsilon$.

6.1 Events

The state machine *receives* two types of events as inputs.

1. *message events*: A message event is received when a vote, a proposal, or a bundle is received. A message event is simply written as the message that is received.
2. *timeout events*: A timeout event is received when a specific amount of time passes after the beginning of a period. A timeout event λ seconds after period p begins is denoted $t(\lambda, p)$.

6.2 Outputs

The state machine produces a series of network transmissions as output. In each transmission, the player broadcasts a vote, a proposal, or a bundle to the rest of the network.

A player may perform a special broadcast called a *relay*. In a relay, the data received from another peer is broadcast to all peers except for the sender.

A broadcast action is simply written as the message to be transmitted. A relay action is written as the same message except with an asterisk. For instance, an action to relay a vote is written as $\text{Vote}^*(r, p, s, v)$.

7 Player State Definition

We define the *player state* S to be the following tuple:

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

where

- r is the current round,
- p is the current period,
- s is the current step,
- $\bar{s}$ is the *last concluding step*,
- V is the set of all votes,
- P is the set of all proposals, and
- $\bar{v}$ is the *pinned* value.

We say that a player has *observed*

- $\text{Proposal}(v)$ if $\text{Proposal}(v) \in P$
- $\text{Vote}_l(r, p, s, v)$ if $\text{Vote}_l(r, p, s, v) \in V$
- $\text{Bundle}(r, p, s, v)$ if $\text{Bundle}_l(r, p, s, v) \subset V$
- that round r (period 0) has *begun* if there exists some p such that $\text{Bundle}(r-1, p, \textit{cert}, v)$ was also observed

- that round r, period $p > 0$ has *begun* if there exists some p such that either
  - Bundle$(r, p-1, s, v)$ was also observed for some $s > \mathit{cert}, v$, or
  - Bundle(r, p, soft, v) was observed for some v.

An event causes a player to observe something if the player has not observed that thing before receiving the event and has observed that thing after receiving the event. For instance, a player may observe a vote Vote, which adds this vote to V:

$$N((r, p, s, \bar{s}, V, P, \bar{v}), L_0, \text{Vote}) = ((r', p', \ldots, V \cup \{\text{Vote}\}, P, \bar{v}'), L_1, \ldots).$$

We abbreviate the transition above as

$$N((r, p, s, \bar{s}, V, P, \bar{v}), L_0, \text{Vote}) = ((S \cup \text{Vote}, P, \bar{v}), L_1, \ldots).$$

Note that *observing* a message is distinct from *receiving* a message. A message which has been received might not be observed (for instance, the message may be from an old round). Refer to the relay rules for details.

7.1 Special Values

We define two functions $\mu(S, r, p), \sigma(S, r, p)$, which are defined as follows:

$\mu(S, r, p)$ is defined as the proposal-value in the vote in round r and period p with the minimal credential. More formally, let

$$V_{r,p} = \{\text{Vote}(I, r, p, 0, v) | \text{Vote} \in V\}$$

where V is the set of votes in S. Then if $\text{Vote}_l(r, p, 0, v_l)$ is the vote with the smallest weight in $V_{r,p}$, then $\mu(S, r, p) = v_l$.

If $V_{r,p}$ is empty, then $\mu(S, r, p) = \perp$.

$\sigma(S, r, p)$ is defined as the sole proposal-value for which there exists a soft-bundle in round r and period p. More formally, suppose Bundle$(r, p, \mathit{soft}, v) \subset V$. Then $\sigma(S, r, p) = v$.

If no such soft-bundle exists, then $\sigma(S, r, p) = \perp$.

If there exists a proposal-value v such that Proposal$(v) \in V$ and $\sigma(S, r, p) = v$, we say that v is *committable for round r, period p* (or simply that v is *committable* if (r, p) is unambiguous).

8 Relay Rules

Here we describe how players handle message events.

Whenever the player receives a message event, it may decide to *relay* that or another message. In this case, the player will produce that output before

producing any subsequent output (which may result from the player's observation of that message; see the broadcast rules below).

A player may receive messages from a peer which indicates that the peer is misbehaving. These cases are marked with an asterisk (*) and enable the node to perform a special action (e.g., disconnect from the peer).

We say that a player *ignores* a message if it produces no outputs on receiving that message.

8.1 Votes

On receiving a vote $Vote_k(r_k, p_k, s_k, v)$ a player

- Ignores* it if Vote_k is invalid.
- Ignores it if $s = 0$ and $\text{Vote}_k \in V$.
- Ignores it if $s = 0$ and Vote_k is an equivocation.
- Ignores it if $s > 0$ and Vote_k is a second equivocation.
- Ignores it if
  - $r_k \notin [r, r+1]$ or
  - $r_k = r + 1$ and either
    - $p_k > 0$ or
    - $s_k \notin (next_0, late)$ or
  - $r_k = r$ and one of
    - $p_k \notin [p-1, p+1]$ or
    - $p_k = p + 1$ and $s_k \notin (next_0, late)$ or
    - $p_k = p$ and $s_k \notin (next_0, late)$ and $s_k \notin [s-1, s+1]$ or
    - $p_k = p - 1$ and $s_k \notin (next_0, late)$ and $s_k \notin [\bar{s}-1, \bar{s}+1]$.
- Otherwise, relays Vote_k, observes it, and then produces any consequent output.

Specifically, if a player ignores the vote,

$$N(S, L, \text{Vote}_k(r_k, p_k, s_k, v)) = (S, L, \epsilon)$$

while if a player relays the vote,

$$N(S, L, \text{Vote}_k(r_k, p_k, s_k, v)) = (S' \cup \text{Vote}(I, r_k, p_k, s_k, v), L', (\text{Vote}^*_k(r_k, p_k, s_k, v), \ldots)).$$

8.2 Bundles

On receiving a bundle $\text{Bundle}(r_k, p_k, s_k, v)$ a player

- Ignores* it if $\text{Bundle}(r, p, s, v)$ is invalid.
- Ignores it if
  - $r_k \neq r$ or
  - $r_k = r$ and $p_k + 1 < p$.

- Otherwise, observes the votes in $\text{Bundle}(r_k, p_k, s_k, v)$ in sequence. If there exists a vote which causes the player to observe some bundle $\text{Bundle}(r_k, p_k, s_k, v')$ for some s_k, then the player relays $\text{Bundle}(r_k, p_k, s_k, v')$, and then executes any consequent action; if there does not, the player ignores it.

Specifically, if the player ignores the bundle without observing its votes,

$$N(S, L, \text{Bundle}(r_k, p_k, s_k, v)) = (S, L, \epsilon);$$

while if a player ignores the bundle but observes its votes,

$$N(S, L, \text{Bundle}(r_k, p_k, s_k, v)) = (S' \cup \text{Bundle}(r_k, p_k, s_k, v), L, \epsilon);$$

and if a player, on observing the votes in the bundle, observes a bundle for some value (not necessarily distinct from the bundle's value),

$$N(S, L, \text{Bundle}(r_k, p_k, s_k, v)) = (S' \cup \text{Bundle}(r_k, p_k, s_k, v), L', (\text{Bundle}^*(r, p_k, s_k, v'), \ldots)).$$

8.3 Proposals

On receiving a proposal $\text{Proposal}(v)$ a player

- Relays $\text{Proposal}(v)$ if $\sigma(S, r+1, 0) = v$.
- Ignores it if it is invalid.
- Ignores it if $\text{Proposal}(v) \in P$.
- Relays $\text{Proposal}(v)$, observes it, and then produces any consequent output, if $v \in \{\sigma(S, r, p), \bar{v}, \mu(S, r, p)\}$.
- Otherwise, ignores it.

Specifically, if the player ignores a proposal,

$$N(S, L, \text{Proposal}(v)) = (S, L, \epsilon)$$

while if a player relays the proposal *after* checking if it is valid,

$$N(S, L, \text{Proposal}(v)) = (S' \cup \text{Proposal}(v), L', (\text{Proposal}^*(v), \ldots)).$$

However, in the first condition above, the player relays $\text{Proposal}(v)$ *without* checking if it is valid. Since the proposal has not been seen to be valid, the player cannot observe it yet, so

$$N(S, L, \text{Proposal}(v)) = (S, L, (\text{Proposal}^*(v))).$$

Note: An implementation may *buffer* a proposal in this case. Specifically, an implementation which relays a proposal without checking that it is valid may optionally choose to replay this event when it observes that a new round has begun (see below). In this case, on the conclusion of a new round, this proposal is processed once again as input.

Implementations may store and relay fewer proposals than specified here to improve efficiency. However, implementations are always required to relay proposals which match the following proposal-values (where r is the current round and p is the current period):

- $\bar{v}$
- $\sigma(S, r, p), \sigma(S, r, p-1)$
- $\mu(S, r, p)$ if $\sigma(S, r, p)$ is not set and $\mu(S, r, p+1)$ if $\sigma(S, r, p+1)$ is not set

9 Internal Transitions

After receiving message events, the player updates some components of its state.

9.1 New Round

When a player observes that a new round $(r, 0)$ has begun, the player sets $\bar{s} := s, \bar{v} := \perp, p := 0, s := 0$. Specifically, if a new round has begun,

$$N((r-i, p, s, \bar{s}, V, P, \bar{v}), L, \ldots) = ((r, 0, 0, s, V', P', \perp), L', \ldots)$$

for some $i > 0$.

9.2 New Period

When a player observes that a new period (r, p) has begun, the player sets $\bar{s} := s, s := 0$. Also, the player sets $\bar{v} := v$ if the player has observed Bundle$(r, p-1, s, v)$ given some values $s > \mathit{cert}$ (or $s = \mathit{soft}$), $v \neq \perp$; if none exist, the player sets $\bar{v} := \sigma(S, r, p-i)$ if it exists, where $p-i$ was the player's period immediately before observing the new period; and if none exist, the player does not update $\bar{v}$.

In other words, if Bundle$(r, p-1, s, v) \in V'$ for some $v \neq \perp, s > \mathit{cert}$ or $s = \mathit{soft}$,

$$N((r, p-i, s, \bar{s}, V, P, \bar{v}), L, \ldots) = ((r, p, 0, s, V', P, v), L', \ldots);$$

and otherwise, if Bundle$(r, p-1, s, \perp) \in V'$ for some $s > \mathit{cert}$ with $\sigma(S, r, p-i)$ defined,

$$N((r, p-i, s, \bar{s}, V, P, \bar{v}), L, \ldots) = ((r, p, 0, s, V', P, \sigma(S, r, p-i)), L', \ldots);$$

and otherwise

$$N((r, p-i, s, \bar{s}, V, P, \bar{v}), L, \ldots) = ((r, p, 0, s, V', P, \bar{v}), L', \ldots);$$

for some $i > 0$ (where $S = (r, p-i, s, \bar{s}, V, P, \bar{v})$).

9.3 Garbage Collection

When a player observes that either a new round or a new period (r, p) has begun, then the player *garbage-collects* old state. In other words,

$$N((r-i, p-i, s, \bar{s}, V, P, \bar{v}), L, \ldots) = ((r, p, \bar{s}, 0, V' \setminus V^*_{r,p}, P' \setminus P^*_{r,p}, \bar{v}), L, \ldots).$$

where

$$\begin{aligned} V^*_{r,p} = &\{\text{Vote}(I, r', p', \bar{s}, v) | \text{Vote} \in V, r' < r\} \\ &\cup \{\text{Vote}(I, r', p', \bar{s}, v) | \text{Vote} \in V, r' = r, p' + 1 < p\} \end{aligned}$$

and $P^*_{r,p}$ is defined similarly.

9.4 New Step

A player may also update its step after receiving a timeout event.

On observing a timeout event of 2λ, the player sets $s := \mathit{cert}$.

On observing a timeout event of $\max\{4\lambda, \Lambda\}$, the player sets $s := \mathit{next}_0$.

On observing a timeout event of $\max\{4\lambda, \Lambda\} + 2^{s_t}\lambda + r$ where $r \in [0, 2^{s_t}\lambda]$ sampled uniformly at random, the player sets $s := s_t$.

In other words,

$$\begin{aligned} N((r, p, s, \bar{s}, V, P, \bar{v}), L, t(2\lambda, p)) &= ((r, p, \mathit{cert}, \bar{s}, V, P, \bar{v}), L', \ldots) \\ N((r, p, s, \bar{s}, V, P, \bar{v}), L, t(\max\{4\lambda, \Lambda\}, p)) &= ((r, p, \mathit{next}_0, \bar{s}, V, P, \bar{v}), L', \ldots) \\ N((r, p, s, \bar{s}, V, P, \bar{v}), L, t(\max\{4\lambda, \Lambda\} + 2^{s_t}\lambda + r, p)) &= ((r, p, s_t, \bar{s}, V, P, \bar{v}), L', \ldots). \end{aligned}$$

10 Broadcast Rules

Upon observing messages or receiving timeout events, the player state machine emits network outputs, which are externally visible. The player may also append an entry to the ledger.

A correct player emits only valid votes. Suppose the player is identified with the address I and possesses the secret key sk, and the agreement is occurring on the ledger L. Then the player constructs a vote $\text{Vote}(I, r, p, s, v)$ by doing the following:

- Let $(\text{pk}, B) = \text{Record}(L, r - \delta_b, I)$, $\bar{B} = \text{Stake}(L, r - \delta_b)$, $Q = \text{Seed}(L, r - \delta_s)$, $\tau = \text{CommitteeThreshold}(s)$, $\bar{\tau} = \text{CommitteeSize}(s)$.
- Encode $x := (I, r, p, s, v), x' := (I, r, p, s)$.
- Try to set $y := \text{Sign}(x, x', \text{sk}, B, \bar{B}, Q, \tau, \bar{\tau})$.

If the signing procedure succeeds, the player broadcasts $\text{Vote}(I, r, p, s, v) = (I, r, p, s, v, y)$. Otherwise, the player does not broadcast anything.

For certain broadcast vote-messages specified here, a node is forbidden to *equivocate* (i.e., produce a pair of votes which contain the same round, period, and step but which vote for different proposal values). These messages are marked with an asterisk (*) below. To prevent accidental equivocation after a power failure, nodes should checkpoint their state to crash-safe storage before sending these messages.

10.1 Resynchronization Attempt

Where specified, a player attempts to resynchronize. A resynchronization attempt involves the following:

- First, the player broadcasts its *freshest bundle*, if one exists. A player's freshest bundle is a complete bundle defined as follows:
  - $\text{Bundle}(r, p, \mathit{soft}, v) \subset V$ for some v, if it exists, or else
  - $\text{Bundle}(r, p-1, s, \bot) \subset V$ for some $s > \mathit{cert}$, if it exists, or else
  - $\text{Bundle}(r, p-1, s, v) \subset V$ for some $s > \mathit{cert}, v \neq \bot$, if it exists.
- Second, if the player broadcasted a bundle $\text{Bundle}(r, p, s, v)$, and $v \neq \bot$, then the player broadcasts $\text{Proposal}(v)$ if the player has it.

Specifically, a resynchronization attempt corresponds to no additional outputs if no freshest bundle exists

$$N(S, L, \ldots) = (S', L', \ldots),$$

corresponds to a broadcast of a bundle after a relay output and before any subsequent broadcast outputs, if a freshest bundle exists but no matching proposal exists

$$N(S, L, \ldots) = (S', L', (\ldots, \text{Bundle}^*(r, p, \mathit{soft}, v), \ldots)),$$

and otherwise corresponds to a broadcast of both a bundle and a proposal after a relay output and before any subsequent broadcast outputs

$$N(S, L, \ldots) = (S', L', (\ldots, \text{Bundle}^*(r, p, \mathit{soft}, v), \text{Proposal}(v), \ldots)).$$

10.2 Proposals

On observing that (r, p) has begun, the player attempts to resynchronize, and then

- if $p = 0$ or there exists some $s > \mathit{cert}$ where $\text{Bundle}(r, p-1, s, \bot)$ was observed, then a player generates a new proposal $(v', \text{Proposal}(v'))$ and then broadcasts $(\text{Vote}(I, r, p, 0, v'), \text{Proposal}(v'))$.

- if $p > 0$ and there exists some $s_0 > \mathit{cert}, v$ where $\text{Bundle}(r, p-1, s_0, v)$ was observed, while there exists no $s_1 > \mathit{cert}$ where $\text{Bundle}(r, p-1, s_1, \perp)$ was observed, then the player broadcasts $\text{Vote}(I, r, p, 0, v)$. Moreover, if $\text{Proposal}(v) \in P$, the player then broadcasts $\text{Proposal}(v)$.

A player generates a new proposal by executing the entry-generation procedure and by setting the fields of the proposal accordingly. Specifically, the player creates a proposal payload $((o, s), y)$ by setting $o := \text{Entry}(L), Q := \text{Seed}(L, r-1)$, $y := \text{Sign}(Q, Q, 0, 0, 0, 0, 0, 0)$, and $s := \text{Rand}(y, \text{pk})$ if $p = 0$ and $s := \text{Hash}(\text{Seed}(L, r-1))$ otherwise. This consequently defines the matching proposal-value $v = (I, p, \text{Digest}(e), \text{Hash}(\text{Encoding}(e)))$.

In other words, if the player generates a new proposal,

$$N(S, L, \ldots) = (S', L', (\ldots, \text{Vote}(I, r, p, 0, v'), \text{Proposal}(v'))),$$

while if the player broadcasts an old proposal,

$$N(S, L, \ldots) = (S', L', (\ldots, \text{Vote}(I, r, p-1, 0, v), \text{Proposal}(v)))$$

if $\text{Proposal}(v) \in P$ and

$$N(S, L, \ldots) = (S', L', (\ldots, \text{Vote}(I, r, p-1, 0, v)))$$

otherwise.

10.3 Reproposal Payloads

On observing $\text{Vote}(I, r, p, 0, v)$, if $\text{Proposal}(v) \in P$ then the player broadcasts $\text{Proposal}(v)$.

In other words, if $\text{Proposal}(v) \in P$,

$$N(S, L, \text{Vote}(I, r, p, 0, v)) = (S', L', (\text{Proposal}(v))).$$

10.4 Filtering

On observing a timeout event of 2λ (where $\mu = (H, H', l, p_\mu) = \mu(S, r, p)$),

- if $\mu \neq \perp$ and if
  - $p_\mu = p$ or
  - there exists some $s > \mathit{cert}$ such that $\text{Bundle}(r, p-1, s, \mu)$ was observed.

  then the player broadcasts $\text{Vote}(I, r, p, \mathit{soft}, \mu)$.
- if there exists some $s_0 > \mathit{cert}$ such that $\text{Bundle}(r, p-1, s_0, \bar{v})$ was observed and there exists no $s_1 > \mathit{cert}$ such that $\text{Bundle}(r, p-1, s_1, \perp)$ was observed, then the player broadcasts* $\text{Vote}(I, r, p, \mathit{soft}, \bar{v})$.

- otherwise, the player does nothing.

In other words, in the first case above,

$$N(S, L, t(2\lambda, p)) = (S, L, \text{Vote}(I, r, p, \mathit{soft}, \mu));$$

while in the second case above,

$$N(S, L, t(2\lambda, p)) = (S, L, \text{Vote}(I, r, p, \mathit{soft}, \bar{v}));$$

and if neither case is true,

$$N(S, L, t(2\lambda, p)) = (S, L, \epsilon).$$

10.5 Certifying

On observing that some proposal-value v is committable for its current round r, and some period $p' \geq p$ (its current period), if $s \leq \mathit{cert}$, then then the player broadcasts* $\text{Vote}(I, r, p, \mathit{cert}, v)$. (It can be shown that this occurs either after a proposal is received or a soft-vote, which can be part of a bundle, is received.)

In other words, if observing a soft-vote causes a proposal-value to become committable,

$$N(S, L, \text{Vote}(I, r, p, \mathit{soft}, v)) = (S', L, (\ldots, \text{Vote}(I, r, p, \mathit{cert}, v)));$$

while if observing a bundle causes a proposal-value to become committable,

$$N(S, L, \text{Bundle}(r, p, \mathit{soft}, v)) = (S', L, (\ldots, \text{Vote}(I, r, p, \mathit{cert}, v)));$$

and if observing a proposal causes a proposal-value to become committable,

$$N(S, L, \text{Proposal}(v)) = (S', L, (\ldots, \text{Vote}(I, r, p, \mathit{cert}, v)));$$

as long as $s \leq \mathit{cert}$.

10.6 Commitment

On observing $\text{Bundle}(r, p, \mathit{cert}, v)$ for some value v, the player *commits* the entry e corresponding to $\text{Proposal}(v)$; i.e., the player appends e to the sequence of entries on its ledger L. (Evidently, this occurs either after a vote is received or after a bundle is received.)

In other words, if observing a cert-vote causes the player to commit e,

$$N(S, L, \text{Vote}(I, r, p, \mathit{cert}, v)) = (S', L||e, \ldots));$$

while if observing a bundle causes the player to commit e,

$$N(S, L, \text{Bundle}(r, p, \mathit{cert}, v)) = (S', L||e, \ldots)).$$

Note: Occasionally, an implementation may not have e at the point e becomes committed. In this case, the implementation may wait until it receives e somehow (perhaps by requesting peers for e). Alternatively, the implementation may continue running the protocol until it receives e. However, if the protocol chooses to continue running, it may not transmit any vote for which $v \neq \perp$ until it has committed e.

10.7 Recovery

On observing a timeout event of

- $T = \max\{4\lambda, \Lambda\}$ or
- $T = \max\{4\lambda, \Lambda\} + 2^{s_t}\lambda + r$ where $r \in [0, 2^{s_t}\lambda]$ sampled uniformly at random,

the player attempts to resynchronize and then broadcasts* $\text{Vote}(I, r, p, \mathit{next}_s, v)$ where

- $v = \sigma(S, r, p)$ if v is committable in (r, p),
- $v = \bar{v}$ if there does not exist a $s_0 > \mathit{cert}$ such that $\text{Bundle}(r, p-1, s_0, \perp)$ was observed and there exists an $s_1 > \mathit{cert}$ such that $\text{Bundle}(r, p-1, s_1, \bar{v})$ was observed,
- and $v = \perp$ otherwise.

In other words, if a proposal-value v is committable in the current period,

$$N(S, L, t(T, p)) = (S', L, (\ldots, \text{Vote}(I, r, p, \mathit{next}_s, v)));$$

while in the second case,

$$N(S, L, t(T, p)) = (S', L, (\ldots, \text{Vote}(I, r, p, \mathit{next}_s, \bar{v})));$$

and otherwise,

$$N(S, L, t(T, p)) = (S', L, (\ldots, \text{Vote}(I, r, p, \mathit{next}_s, \perp))).$$

10.8 Fast Recovery

On observing a timeout event of $T = k\lambda_f + r$ where k is a positive integer and $r \in [0, \lambda_f]$ sampled uniformly at random, the player attempts to resynchronize. Then,

- The player broadcasts* $\text{Vote}(I, r, p, \mathit{late}, v)$ if $v = \sigma(S, r, p)$ is committable in (r, p).
- The player broadcasts* $\text{Vote}(I, r, p, \mathit{redo}, \bar{v})$ if there does not exist a $s_0 > \mathit{cert}$ such that $\text{Bundle}(r, p-1, s_0, \perp)$ was observed and there exists an $s_1 > \mathit{cert}$ such that $\text{Bundle}(r, p-1, s_1, \bar{v})$ was observed.
- Otherwise, the player broadcasts* $\text{Vote}(I, r, p, \mathit{down}, \perp)$.

Finally, the player broadcasts all $\mathrm{Vote}(I, r, p, \mathit{late}, v) \in V$, all $\mathrm{Vote}(I, r, p, \mathit{redo}, v) \in V$, and all $\mathrm{Vote}(I, r, p, \mathit{down}, \perp) \in V$ that it has observed.