Unicity BFT Core

The Unicity Team

Specification

Version TestNet V1 (main/c665b5ae)

January 29, 2026

Contents

1 General Description

1.1 System Architecture

The Unicity protocol employs a hierarchical three-layer architecture designed for scalability and decentralization. Each layer serves a distinct function in the system:

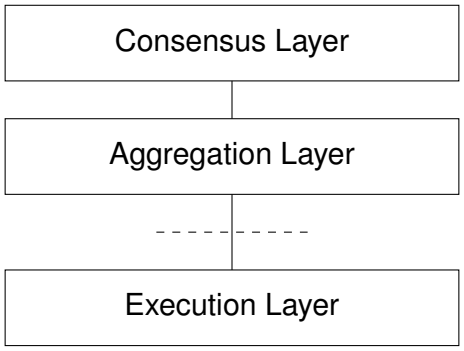

Figure 1. Hierarchical architecture of the Unicity Network.

Consensus Layer PoW chain and BFT Core. Provides token minting, decentralized agreement and finality through Byzantine Fault Tolerant consensus. This layer verifies the integrity of the Aggregation Layer's state transitions and serves as the root of trust for the entire system. The BFT Core certifies SMT root hashes submitted by the aggregation layer.

Aggregation Layer Maintains a global append-only registry of spent token states using sharded Sparse Merkle Trees (SMT). Each partition within this layer processes state certification requests, provides inclusion proofs, and uses consistency proofs to enable trustless verification. The layer is organized into partitions for service design flexibility, with each partition potentially divided into shards for horizontal scalability.

Execution Layer Handles off-chain transaction processing and business logic. Token transactions occur peer-to-peer between users, with only cryptographic commitments to state transitions submitted to the Aggregation Layer for certification. This design moves computational overhead off-chain while maintaining trustless double-spending prevention.

The BFT Core certifies state transitions by cryptographically signing the root hashes of the Aggregation Layer's SMTs. This hierarchical design enables linear scalability: the BFT Core's throughput requirements grow only with the number of shards, not with the number of transactions processed by the system.

2 Framework Data Structures

2.1 Sharding Schemes

Sharding scheme $\mathcal{SH}$ of type $\mathbb{SH}$ is an irreducible prefix-free code (Appendix A).

If $\mathcal{SH} = \{\bigsqcup\}$, then there is a single shard.

In the description of a sharding scheme, the shard identifiers are listed in the topological order $\sigma_1 \prec \sigma_2 \prec \ldots \prec \sigma_n$ (Appendix A).

Every sharding scheme $\mathcal{SH}$ induces a sharding function $f_{\mathcal{SH}} : \mathbb{I} \to \mathcal{SH}$. The shard $f_{\mathcal{SH}}(\text{sid})$ responsible for recording a token state with identifier sid is the shard whose identifier σ_i is a prefix of sid. With $\mathcal{SH}$ an irreducible prefix-free code, there is exactly one such σ_i.

2.2 Networks

In practice, there will be several instances of Unicity networks, each consisting of its own BFT Core and its own set of partitions. Each such network will have an identifier α:

- $\alpha = 1$ is the public mainnet instance;
- $\alpha = 2$ is the public testnet instance;
- $\alpha = 3$ is a local development instance;
- the identifiers $4 \ldots 8$ are reserved for future extensions;
- any additional instances of Unicity networks should use identifiers starting from 9.

To reduce the risk of confusion, it is recommended for each major deployment to use a unique identifier. However, there is no central registry to enforce this constraint.

It is strongly recommended to avoid using the identifier 0, so that an uninitialized identifier (which defaults to 0 in many programming languages) would not match any actual network.

2.3 Partitions and Shards

Unicity network's aggregation layer consists of multiple *partitions*. Each partition is identified by a partition identifier $\beta \in \mathbb{C}$. Each partition runs an unicity proving service instance, or some other utility service instance[1].

A partition may be divided into one or more *shards* according to a sharding scheme (Sec. 2.1). Each shard is implemented by one or more *aggregation layer nodes* that maintain the SMT for their portion of the key space.

[1] Roughly, this is similar to Layer 1, Layer 2 division of Ethereum type blockchains

2.4 Partition Type and Partition Type Descriptor

Every aggregation partition in the Unicity Framework with partition identifier β has a type that is either:

1. a *standard type* (SMT), or
2. a *defined type* that is described as a data structure of type $\mathbb{SD}$.

2.4.1 Partition Type Descriptor

A *partition type descriptor* is a tuple $\mathcal{SD} = (\text{hf}, \mathbb{CP}, V_{CP}, \gamma_{CP}) \in \mathbb{SD}$, where:

- hf – hash function identifier (e.g., SHA-256)
- $\mathbb{CP}$ – consistency proof type
- $V_{CP}\colon \mathbb{IR} \to \mathbb{CP}$ – consistency proof computation function
- $\gamma_{CP}\colon \mathbb{IR} \times \mathbb{CP} \to \mathbb{B}$ – consistency proof verification predicate

For the SMT-based aggregation, the partition maintains a sparse Merkle tree mapping state identifiers to transaction hashes. The hash function hf is used for all cryptographic operations, and the encoding scheme enc specifies how data structures are serialized before hashing (see Sec. 2.7.1 for SMT structure).

The *consistency proof* is an opaque data structure that allows BFT Core nodes to verify that aggregation layer shard nodes correctly performed SMT manipulations during a round. The actual mechanism and content of consistency proofs will be defined separately for each partition type.

2.4.2 Standard Partition Types

Unicity SMT (st $= 1$) – Sparse Merkle Tree based unicity prover

EVM (st $= 2$) – Enshrined EVM runtime

2.4.3 Partition Description Record

A *partition description record* is a tuple $\mathcal{CD} = (\alpha, \beta, \text{st}, \mathcal{SD}, \mathcal{SH}) \in \mathbb{CD}$, where:

- $\alpha \in \mathbb{A}$ is the network identifier;
- $\beta \in \mathbb{C}$ is the partition identifier;
- st $\in \mathbb{ST}$ is the system type identifier (for standard types, st > 0);
- $\mathcal{SD} \in \mathbb{SD} \cup \{\perp\}$ is the system type descriptor (exists if st $= 0$);
- $\mathcal{SH} \in \mathbb{SH}$ is the sharding scheme.

Instead of $\mathcal{CD}.\mathcal{SD}.x$ (where x is any field of $\mathcal{SD}$), we use the shorthand notation $\mathcal{CD}.x$. For standard type partitions, the field $\mathcal{CD}.\mathcal{SD}$ does not exist. In this case, the notation $\mathcal{CD}.x$ will mean the constant value of x of the standard type st.

2.5 Unicity Service Request

A *Unicity Service Request* is a request to the Unicity Service for state transition certification. It is a tuple $Q = (\text{pk}, h_{\text{st}}, h_{\text{tx}}, \sigma)$, where:

- pk – current owner's public key
- $h_{\text{st}} \in \mathbb{H}$ – current state hash
- $h_{\text{tx}} \in \mathbb{H}$ – transaction data hash
- σ – owner's digital signature authorizing the state transition

The Unicity Service processes the request by:

1. Verifying the digital signature: $\text{Verify}_{\text{pk}}(H(h_{\text{st}}, h_{\text{tx}}), \sigma) = 1$
2. Computing the state identifier: $\text{sid} = H(\text{pk}, h_{\text{st}})$
3. Checking uniqueness: the state has not been spent before (i.e., sid does not already exist in the SMT)
4. If all checks pass: setting the SMT leaf $(\text{sid}, h_{\text{tx}})$ in the tree

The request is accepted without returning a synchronous response. The detailed validation algorithm will be specified in Sec. 2.6.

2.6 Unicity Service Request Validation

The aggregation layer nodes process Unicity Service Requests by validating them and, if valid, updating their SMT. Each shard maintains an SMT for its portion of the key space.

2.6.1 Validation and Processing Algorithm

Given a Unicity Service Request $Q = (\text{pk}, h_{\text{st}}, h_{\text{tx}}, \sigma)$, the aggregation layer node performs the following checks:

1. **Signature verification**: Verify that the signature is valid:

$$\text{Verify}_{\text{pk}}(H(h_{\text{st}}, h_{\text{tx}}), \sigma) = 1$$

2. **Uniqueness check**: Compute the state identifier and check that it does not already exist as a leaf in the SMT:

$$\text{sid} = H(\text{pk}, h_{\text{st}})$$

   Verify that the SMT does not contain a leaf with key sid (i.e., the state has not been spent before).

If both conditions are satisfied, the request is accepted and the aggregation layer node sets the SMT leaf:

$$\text{Set SMT leaf: } (\text{sid}, h_{\text{tx}})$$

This updates the SMT by inserting the new leaf.

If either check fails, the request is rejected:

- If signature verification fails: the request is invalid and must be rejected

- If uniqueness check fails (leaf with key sid already exists): the state has already been spent (double-spend attempt) and the request must be rejected

After processing a batch of requests, the aggregation layer nodes compute the new SMT root hash h and send a certification request to the BFT Core. The BFT Core verifies the consistency proof π_{CP} and that last state is extended correctly and certifies the new root hash h via the Unicity Certificate.

2.7 Certificates

Certificates are compact proofs of inclusion (or sometimes uniqueness, or exclusion) of some data item in an authenticated data structure. Certificates may be chained. Combining certificates, it is possible to put together *proofs* (Sec. 2.8) proving e.g. inclusion, non-inclusion or uniqueness of transactions.

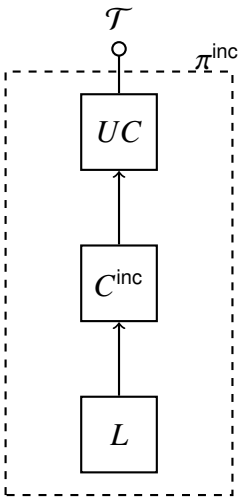

Figure 2. Inclusion Proof: SMT leaf L with certificate chain through UC to trust base $\mathcal{T}$

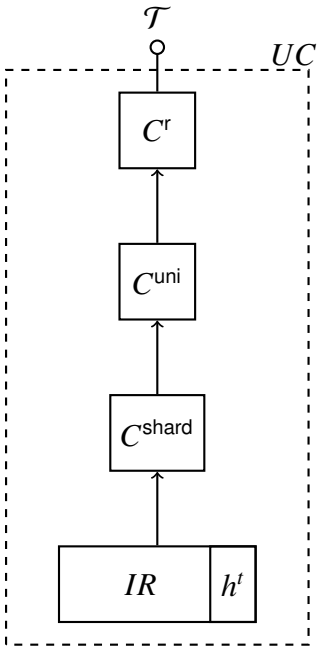

Figure 3. Unicity Certificate is a chain of certificates and a certified input data record IR

See Fig. 2 for the Inclusion Proof structure and Fig. 3 for the Unicity Certificate. The Inclusion Proof contains an SMT leaf $L = (\text{sid}, h_{\text{tx}})$ where $\text{sid} = H(\text{pk}, h_{\text{st}})$ is a state identifier and h_{tx} is a transaction hash. The Inclusion Certificate C^{inc} connects the leaf to the SMT root hash h in the Input Record. The Unicity Certificate then chains through the shard tree, unicity tree, and unicity seal to the global root of trust: Unicity Trust Base $\mathcal{T}$.

2.7.1 Sparse Merkle Tree

The aggregation layer maintains a *Sparse Merkle Tree* (SMT) in compressed paths form. The SMT maps state identifiers to transaction hashes and provides compact inclusion proofs. Refer to Appendix C.3 for details.

2.7.1.1 SMT Leaf Structure

An SMT leaf contains a key-value pair:

- **Key**: State identifier sid $= H(\text{pk}, h_{\text{st}})$ where:
  - pk – public key of an unicity token owner
  - h_{st} – state hash representing the current state of a token
- **Value**: Transaction data hash h_{tx}.

The SMT effectively maintains a mapping sid $\mapsto h_{\text{tx}}$ that proves a state transition from state identifier sid was authorized via transaction with hash h_{tx}. This prevents double-spending: once a state identifier is registered in the SMT with a transaction hash, that state cannot be spent again.

2.7.1.2 Inclusion Certificate

An *Inclusion Certificate* C^{inc} for a leaf with key sid is a sequence of pairs:

$$C^{\text{inc}} = \langle (p_1, d_1), (p_2, d_2), \ldots, (p_n, d_n) \rangle$$

where each pair (p_i, d_i) consists of:

- $p_i \in \{0,1\}^*$ – a path segment (bit-string)
- $d_i \in \mathbb{H} \cup \mathbb{OCT}^*$ – for $i = 1$ this is the leaf data, for $i > 1$ this is a sibling hash value

The first pair (p_1, d_1) contains the leaf's path segment and data. Each subsequent pair (p_i, d_i) for $i > 1$ represents a step up the tree where p_i is the path segment from the current node to its parent, and d_i is the sibling hash value.

The concatenated path $p_1 \| p_2 \| \ldots \| p_n$ equals the full key sid. The respective value d_1 records h_{st}.

The verification function SMT_VERIFY_INCLUSION() is provided in Appendix C.3.5.

2.7.2 Shard Tree Certificate

Shard Tree Certificate connects the SMT root hash from a shard aggregator to a leaf of unicity tree. It is a tuple $C^{\text{shard}} = (\sigma; h^s_1, \ldots, h^s_{|\sigma|})$, where:

1. $\sigma = \sigma_1 \sigma_2 \ldots \sigma_{|\sigma|}$ is a shard identifier of type $\{0,1\}^*$
2. $h^s_1, \ldots, h^s_{|\sigma|}$ – sibling hashes of type $\mathbb{H}$

For the single shard case, the sharding scheme is $\{\sqcup\}$ and the certificate is $(\sqcup, \perp)$.

2.7.2.1 Sharded SMT Structure

In a sharded setting, each shard aggregator node maintains an SMT for its portion of the key space. The parent aggregator maintains a non-sparse tree (with all leaves present) where each leaf corresponds to a shard's SMT root hash.

To facilitate integration, each shard aggregator computes its root hash as $h_{\text{root}} = H(p, h_L, h_R)$ where p is a 1-bit path consisting of the leftmost bit of the shard identifier. This ensures the shard root hash can be directly used as a leaf value in the parent aggregator's tree.

2.7.2.2 Creation: CreateShardTreeCert

Input:

1. σ – shard identifier
2. χ – shard tree of type $\chi\colon \overline{\mathcal{SH}} \to \mathbb{H}$

Output: shard tree certificate $C^{\text{shard}} = (\sigma; h^s_1, \ldots, h^s_{|\sigma|})$

Computation:

$C \leftarrow ()$
for $i \leftarrow |\sigma|$ downto 1 **do**
  $C \leftarrow C \,||\, \chi(\sigma_1\sigma_2 \ldots \sigma_{i-1}\overline{\sigma_i})$
end for
return $(\sigma; C)$

where $\sigma_1\sigma_2 \ldots \sigma_m$ is the binary representation of σ, and $\overline{\sigma_i}$ is the binary complement of σ_i.

For the single shard case, the shard certificate is $(\bigsqcup, \bot)$.

2.7.2.3 Computation: CompShardTreeCert

Input:

1. $(\sigma; h^s_1, \ldots, h^s_{|\sigma|})$ – shard tree certificate, where $\sigma = \sigma_1\sigma_2 \ldots \sigma_{|\sigma|}$ is a shard identifier
2. IR – Shard Input Record
3. h_t – hash value of type $\mathbb{H}$

Output: Root hash r of type $\mathbb{H}$

Computation:

$r \leftarrow H(IR \,||\, h_t)$
for $i \leftarrow |\sigma|$ downto 1 **do**
  if $\sigma_i = 0$ **then** $r \leftarrow H(r, h^s_i)$
  if $\sigma_i = 1$ **then** $r \leftarrow H(h^s_i, r)$
end for
return r

For the single shard case, CompShardTreeCert$((\bigsqcup, \bot), IR, h_t)$ returns $H(IR \,||\, h_t)$.

2.7.3 Unicity Tree Certificate

Unicity Tree Certificate (Fig. 3) is a tuple $C^{\text{uni}} = (\beta, \text{dhash}; (\beta_2, h_2), \ldots, (\beta_\ell, h_\ell))$, where:

1. β – partition identifier of type $\mathbb{C}$
2. dhash – partition description hash of type $\mathbb{H}$
3. $(\beta_2, h_2), \ldots, (\beta_\ell, h_\ell)$ – a sequence of partition identifier and sibling hash pairs

2.7.3.1 Creation: CreateUnicityTreeCert

Input:

1. β – partition identifier of type $\mathbb{C}$
2. C – set of partition identifiers of type $\mathbb{C}$
3. $\mathcal{CD}$ – partition description of type $\mathbb{CD}[C]$
4. $\mathcal{IH}$ – input hashes of type $\mathbb{H}[C]$

Output: unicity tree certificate $(\beta, \text{dhash}; (\beta_2, h_2), \ldots, (\beta_\ell, h_\ell))$

Computation:

$n \leftarrow |C|$ ▹ Number of partitions
$C \leftarrow \text{sorted}(C)$ ▹ Sorted list
for $i \leftarrow 1 \ldots n$ **do**
  $x_i \leftarrow H(\mathcal{IH}[C_i] \,||\, H(\mathcal{CD}[C_i]))$
end for
$\langle(\beta_1, h_1), (\beta_2, h_2), \ldots, (\beta_\ell, h_\ell)\rangle \leftarrow$ INDEX_TREE_CHAIN$((C_1, x_1), (C_2, x_2), \ldots, (C_n, x_n), \beta)$
return $(\beta, H(\mathcal{CD}[\beta]); (\beta_2, h_2), \ldots, (\beta_\ell, h_\ell))$ ▹ Drop redundant first hash step

2.7.3.2 Computation: CompUnicityTreeCert

Input:

1. $(\beta, \text{dhash}; (\beta_2, h_2), \ldots, (\beta_\ell, h_\ell))$ – unicity tree certificate
2. x – input hash (output of the CompShardTreeCert function)

Output: Root hash r of type $\mathbb{H}$

Computation:

$L \leftarrow \langle(\beta, H(x, \text{dhash})), (\beta_2, h_2), \ldots, (\beta_\ell, h_\ell)\rangle$ ▹ Restore the first hash step
return INDEX_TREE_OUTPUT$(L; \beta)$ ▹ Sec. C.2.3

2.7.4 Unicity Seal

Unicity Seal is a tuple $C^{\text{r}} = (\alpha, n_r, e_r, t_r, r_-, r; s)$, where:

1. α – Network identifier
2. n_r – BFT Core's Round number
3. e_r – BFT Core's Epoch number
4. t_r – Round creation time (wall clock value specified and verified by the BFT Core), with one-second precision. See Appendix B for encoding
5. r_- – Root hash of previous round's Unicity Tree
6. r – Root hash of the Unicity Tree (denoted as r^{root} if necessary for clarity)
7. s – Signature, computed by the BFT Core over preceding fields ($s = \text{Sign}_{\text{sk}_r}(n_r, e_r, t_r, r_-, r)$). The formulation of signature field depends on underlying consensus mechanism and its parameters. Here we assume an opaque data structure which can be verified based on the Unicity Trust Base, i.e., there is an implementation of an abstract function $\text{Verify}_{\mathcal{T}}((n_r, e_r, t_r, r_-, r), s)$, encapsulated into the implementation of VerifyUnicitySeal.

VerifyUnicitySeal – unicity seal verification function of type $\mathbb{H} \times \mathbb{US} \times \mathbb{UB} \rightarrow \mathbb{B}$. This function also verifies, if Unicity Seal's network identifier matches with the Unicity Trust Base.

2.7.5 Unicity Certificate

Unicity Certificate is a tuple $UC = (IR, h_t, C^{\text{shard}}, C^{\text{uni}}, C^{\text{r}})$, where:

1. $\mathcal{IR}$ is a shard input record of type $\mathbb{IR}$ (Sec. 3.1.1),
2. h_t is the hash (of technical record) of type $\mathbb{H}$,
3. C^{shard} is a shard tree certificate,
4. C^{uni} is a unicity tree certificate, and
5. C^{r} is a unicity seal

Elements of the tuple form an authenticated chain, see function Verication: VerifyUnicityCert.

2.7.5.1 Verication: VerifyUnicityCert

Verifies if unicity certificate is valid, based on unicity trust base as the root of trust.

Input:

1. $UC = (IR, h_t, C^{\text{shard}}, C^{\text{uni}}, C^{\text{r}})$ – Unicity Certificate
2. $\mathcal{T}$ – Unicity Trust Base

Output: True or False

Computation:
$r \leftarrow$ CompShardTreeCert$(UC.C^{\text{shard}}, UC.IR, UC.h_t)$
$r \leftarrow$ CompUnicityTreeCert$(UC.C^{\text{uni}}, r)$
return VerifyUnicitySeal$(r, C^{\text{r}}, \mathcal{T}) = 1$

2.8 Proofs

2.8.1 Inclusion Proof

Inclusion Proof proves that a state transition (mapping from state identifier to transaction hash) has been registered in the SMT and certified by the BFT Core. It is a tuple $\pi^{\text{inc}} = (L, C^{\text{inc}}, UC)$, where:

1. L – SMT leaf containing the state transition
2. C^{inc} – Inclusion Certificate (Sec. 2.7.1.2)
3. UC – unicity certificate (Sec. 2.7.5)

2.8.1.1 SMT Leaf

An SMT leaf L is a tuple $L = (\text{sid}, h_{\text{tx}})$, where:

- sid $= H(\text{pk}, h_{\text{st}})$ – state identifier, computed as the hash of token owner's public key and state hash
- h_{tx} – transaction data hash

SMT Leaf can be extracted from Inclusion Certificate like this:

Input: $C^{\text{inc}} = \langle (p_1, d_1), (p_2, d_2), \ldots, (p_n, d_n) \rangle$ – Inclusion Certificate

Output: $L = (\text{sid}, h_{\text{tx}})$ – SMT Leaf

Computation:

sid $\leftarrow p_1 \| p_2 \| \ldots \| p_n$ ▷ Concatenate all path segments
$h_{\text{tx}} \leftarrow d_1$ ▷ Extract leaf data from first pair
return $(\text{sid}, h_{\text{tx}})$

2.8.1.2 Verification: VerifyInclusionProof

VerifyInclusionProof – inclusion proof verification function of type $\mathbb{IP} \times \mathbb{UB} \times \mathbb{CD} \rightarrow \mathbb{B}$

Input:

1. $\pi^{\text{inc}} = (C^{\text{inc}}, UC)$ – inclusion proof
2. $\mathcal{T}$ – trust base

Output: TRUE or FALSE

Computation:

$h \leftarrow UC.IR.h$ ▷ Extract SMT root hash from IR
if ¬SMT_VERIFY_INCLUSION(sid, $h_{\text{tx}}, C^{\text{inc}}, h$) **then return** FALSE
end if
if ¬VerifyUnicityCert($UC, \mathcal{T}$) **then return** FALSE
end if
return TRUE

The inclusion proof applies to the leaf which matches the leaf extracted from the Inclusion Certificate.

2.8.2 Non-Inclusion Proof

Non-Inclusion Proof proves that a state identifier does NOT exist in the SMT. It cryptographically demonstrates that a StateID is absent from the tree by showing the path to where it would be located if it existed.

A non-inclusion proof uses the same data structure as the inclusion proof but with different interpretation. The non-inclusion proof $\pi^{\text{exc}} = (C^{\text{inc}}, UC)$ of sid is valid if VerifyInclusionProof($\pi^{\text{exc}}, \mathcal{T}$) = TRUE and p_1 is a prefix of sid and $d_1 = \emptyset$.

2.8.3 Consistency Proof

A *Consistency Proof* (also called *non-deletion proof*) is a cryptographic construction that validates one round of operation of the append-only SMT maintained by aggregation layer nodes. It proves that the state transition from previous SMT root hash h' to current SMT root hash h was performed correctly: only new leaves were added, and no pre-existing leaves were removed or modified.

The consistency proof enables the BFT Core to verify the integrity of the Aggregation Layer's operations in a trustless manner.

For a batch of state transitions processed in a single round, the consistency proof π_{CP} of type $\mathbb{CP}$ is a tuple (B, Π) where B is the batch of new SMT leaves and Π contains the sibling hashes needed for verification (see Appendix C.3.7 for detailed structure).

2.8.3.1 Verification Function

VerifyConsistencyProof – consistency proof verification function of type $\mathbb{CP} \times \mathbb{H} \times \mathbb{H} \to \mathbb{B}$

Input:

1. $\pi_{CP} = (B, \Pi)$ – consistency proof where $B = \langle(\text{sid}_1, h_{\text{tx},1}), \ldots, (\text{sid}_j, h_{\text{tx},j})\rangle$
2. h' – previous SMT root hash (before the round)
3. h – current SMT root hash (after the round)

Output: True or False

The verification algorithm is defined in Appendix C.3.7.3.

A valid consistency proof guarantees: (1) **Append-only property**: No existing leaves were deleted or modified; (2) **Correctness**: All new leaves in batch B were correctly inserted at positions determined by their state identifiers; (3) **Completeness**: The state transition from h' to h includes exactly the leaves in B and no others.

3 BFT Core

3.1 Data Structures of the BFT Core

3.1.1 Shard Input Record

Shard input record (IR) of a shard of a partition (of type $\mathbb{IR}$) is a tuple (n, e, h', h, t), where:

1. n – shard's round number of type $\mathbb{N}_{64}$
2. e – shard's epoch number of type $\mathbb{N}_{64}$
3. h' – previous round's SMT root hash of type $\mathbb{H}$
4. h – current round's SMT root hash of type $\mathbb{H}$
5. t – reference time for request validation of type $\mathbb{N}_{64}$

3.1.2 Statistical Record

Statistical record of type $\mathbb{SR}$ is a tuple SR $= (n_e, \overline{\ell}_B, \overline{\ell}_S, \hat{\ell}_B, \hat{\ell}_S)$, where:

1. n_e – number of non-empty rounds (adding at least one new leaf), of type $\mathbb{N}_{64}$
2. $\overline{\ell}_S$ – memory usage of the SMT, of type $\mathbb{N}_{64}$
3. $\hat{\ell}_B$ – maximum number of SMT additions processed during a round, of type $\mathbb{N}_{64}$

There is one Statistical Record of current epoch where the values are being updated, reflecting the current state since the beginning of the epoch; and one invariant Statistical Record of the preceding epoch of every shard of every public partition.

3.1.3 Technical Record

Technical record of type $\mathbb{TE}$ is a tuple TE $= (n_r, e_r, \nu_\ell, h_{\text{sr}})$, where:

1. n_r – suggested next round number of type $\mathbb{N}_{64}$
2. e_r – suggested next epoch number of type $\mathbb{N}_{64}$
3. ν_ℓ – suggested leader identifier of type $\{0, 1\}^*$
4. h_{sr} – hash of statistical records; type $\mathbb{H}$

Technical record is delivered with an Unicity Certificate. When a shard is extending the Unicity Certificate with a next round certification attempt, then it must use the suggested values provided in Technical Record.

There is one Technical Record for every shard of every partition, providing synchronization for the next round.

3.1.4 Shard Info

Shard info of type $\mathbb{SI}$ is a tuple $(n, e, h_-, \text{SR}_-, \text{SR}, \mathcal{V}, v_\ell, UC_-)$, where:

1. n – shard's round number of type $\mathbb{N}_{64}$
2. e – shard's epoch number of type $\mathbb{N}_{64}$
3. h_- – last-certified root hash
4. SR_- – statistical record of the previous epoch
5. SR – statistical record of the current epoch, initially (at each epoch) $(0, 0, 0, 0, 0)$
6. $\mathcal{V}$ – validators of the shard, a set of identifiers, each of type $\{0,1\}^*$
7. v_ℓ – leader identifier of type $\{0,1\}^*$ (it is assumed that $v_\ell \in \mathcal{V}$)
8. UC_- – last created unicity certificate

3.1.5 Shard Tree

For a partition β, let $\mathcal{SH}_\beta = \mathcal{CD}[\beta].\mathcal{SH}$, and $\mathcal{IRT}_\beta$ be of type $(\mathbb{IR}, \mathbb{H})[\mathcal{SH}_\beta]$, i.e. for every $\beta \in C$ and $\sigma \in \mathcal{SH}_\beta$, $\mathcal{IRT}_\beta[\sigma] = (\mathcal{IR}, h_t)$ (for some $\mathcal{IR}$ and h_t).

If there is no input from a shard to certify then $\mathcal{IRT}_\beta[\sigma] = \mathcal{IRT}_\beta[\sigma]_-$, that is, the value from the previously built Shard Tree is used. If there is no previous value then $\mathcal{IRT}_\beta[\sigma] = \perp$.

Shard tree for a partition β is a function $\chi_\beta \colon \overline{\mathcal{SH}_\beta} \to \mathbb{H}$ such that:

1. If $\sigma \in \mathcal{SH}_\beta$, then $\chi_\beta(\sigma) = H(\mathcal{IRT}_\beta[\sigma]) = H(\mathcal{IRT}_\beta[\sigma].\mathcal{IR} \parallel \mathcal{IRT}_\beta[\sigma].h_t)$.
2. If $\sigma \in \overline{\mathcal{SH}_\beta} \backslash \mathcal{SH}_\beta$, then $\chi_\beta(\sigma) = H(\chi_\beta(\sigma\|0) \parallel \chi_\beta(\sigma\|1))$.

The value $\chi_\beta(\text{␣})$ is called the root hash of the shard tree.

Shard tree certificate for a shard $\sigma \in \mathcal{SH}_\beta$ is a sequence $h^s_1, \ldots, h^s_m$ of sibling hash values of type $\mathbb{H}$, where $m = |\sigma|$ (the number of bits in σ); see Sec. 2.7.2.

3.1.5.1 Shard Tree Creation: CreateShardTree

Input:

1. $\mathcal{SH}$ – sharding scheme of type $\mathbb{SH}$
2. $\mathcal{IRT}$ – shard-specific data of type $(\mathbb{IR}, \mathbb{H})[\mathbb{SH}]$

Output: χ – shard tree of type $\overline{\mathcal{SH}} \to \mathbb{H}$

Computation: $\chi \leftarrow \perp$, genST(␣)

where genST is the following recursive function of type $\{0,1\}^* \to \mathbb{H}$ with side effects:

genST(σ):

1. **if** $\sigma \in \mathcal{SH}$, **then store** $\chi(\sigma) \leftarrow H(\mathcal{IRT}[\sigma])$ and **return** $\chi(\sigma)$.
2. **if** $\sigma \in \overline{\mathcal{SH}} \backslash \mathcal{SH}$, **then store** $\chi(\sigma) \leftarrow H(\text{genST}(\sigma\|0) \parallel \text{genST}(\sigma\|1))$ and **return** $\chi(\sigma)$

For the single shard case, the shard tree is $\{(\text{␣}, H(\mathcal{IRT}[\sigma])\}$, i.e. it only has a single record for the root hash $\chi(\text{␣}) = H(\mathcal{IRT}[\sigma])$.

3.1.6 Unicity Tree

Unicity Tree is an indexed Merkle tree that aggregates the state of all partitions in the network.

3.1.6.1 Creation: CreateUnicityTree

Input:

1. C – set of partition identifiers of type $\mathbb{C}$
2. $\mathcal{CD}$ – partition description of type $\mathbb{CD}[C]$
3. $\mathcal{IH}$ – input hashes of type $\mathbb{H}[C]$

Output: r – unicity tree root hash of type $\mathbb{H}$

Computation:

$n \leftarrow |C|$ ▹ Number of partitions
$C \leftarrow \text{sorted}(C)$ ▹ Sorted list
for $i \leftarrow 1 \ldots n$ **do**
  $x_i \leftarrow H(\mathcal{IH}[C_i] \,||\, H(\mathcal{CD}[C_i]))$
end for
return INDEX_TREE_ROOT$((C_1, x_1), (C_2, x_2), \ldots, (C_n, x_n))$ ▹ Sec. C.2.1

3.2 State of the BFT Core

State of the BFT Core is a tuple $(\alpha, n, e, r_{-}, \mathcal{T}, \mathcal{C}, \mathcal{CD}, \mathcal{SI})$, where:

1. α – network identifier of type α
2. n – BFT Core's round number of type $\mathbb{N}_{64}$
3. e – BFT Core's epoch number of type $\mathbb{N}_{64}$
4. r_{-} – previous root hash of the unicity tree of type $\mathbb{H}$
5. $\mathcal{T}$ – unicity trust base of type $\mathbb{UB}$
6. $\mathcal{C}$ – set of partition identifiers, with elements of type $\mathbb{C}$
7. $\mathcal{CD}$ – partition descriptions of type $\mathbb{CD}[\mathbb{C}]$
8. $\mathcal{SI}$ – shard info of type $\mathbb{SI}[\mathbb{C}, \{0,1\}^*]$

Epoch number can be interpreted as the version number of some shard's or BFT Core's configuration. It is used by supporting layers like orchestration and consensus. On static configuration, the epoch is 0.

3.3 Messages of the BFT Core

3.3.1 Certification Request

Certification Request (CR) of a shard is a message $\langle \alpha, \beta, \sigma, \nu; \mathcal{IR}, \ell_B, \ell_S, [\pi_{CP}]; s; \rangle$, where:

1. α – network identifier of type $\mathbb{A}$
2. β – partition identifier of type $\mathbb{C}$

3. σ – shard identifier of type $\mathbb{SH}$
4. ν – validator identifier (aggregation layer node identifier) of type $\{0,1\}^*$
5. $\mathcal{IR}$ – shard input record of type $\mathbb{IR}$
6. ℓ_B – number of added new SMT leaves during the round, type $\mathbb{N}_{64}$
7. ℓ_S – SMT memory usage in bytes of type $\mathbb{N}_{64}$
8. π_{CP} – consistency proof of type $\mathbb{CP}$ (Sec. 2.8.3)
9. s – signature authenticating the message

Certification requests are sent to the BFT Core by the aggregation layer nodes (validators) implementing the shards.

3.3.2 Certification Response

Certification Response (CReS) of a shard is a message $\langle \alpha, \beta, \sigma, \text{TE}; UC \rangle$, where:

1. α – network identifier of type $\mathbb{A}$
2. β – partition identifier of type $\mathbb{C}$
3. σ – shard identifier of type $\mathbb{SH}$
4. TE – technical record of type $\mathbb{TE}$
5. UC – certified unicity certificate

Certification response is sent by the BFT Core to the nodes of an aggregation layer nodes as a response to a certification request message. This is an asynchronous message: response is not immediate, and there can be multiple subsequent responses.

3.4 Functional Description of the BFT Core

During every round, the BFT Core receives certification requests from the shards of partitions $\beta \in \mathcal{C}$.

The BFT Core certifies the SMT root hashes provided by the aggregation layer nodes. Each shard maintains an SMT for its portion of the key space. When aggregation layer nodes process Unicity Service Requests (as specified in Sec. 2.6), they update their SMTs by setting leaves. The BFT Core certifies these updates by:

1. Receiving certification requests containing the new SMT root hash h and previous root hash h'
2. Verifying the consistency proof π_{CP} that proves correct SMT manipulation
3. Certifying the new root hash via the Unicity Certificate

As every shard σ of a partition β is implemented by a redundant distributed system with certain number $m = |\mathcal{SI}[\beta, \sigma].\mathcal{V}|$ of validator machines, it is possible that several certification requests $\text{CR}_{\beta,\sigma}$ with the same $\text{CR}_{\beta,\sigma}.\mathcal{IR}.h'$ but different $\text{CR}_{\beta,\sigma}.\mathcal{IR}.h$ are received. This is solved by a majority voting mechanism requiring that a required majority, at least $\lfloor m/2 \rfloor + 1$ of the dedicated validators, send $\text{CR}_{\beta,\sigma}$ with an identical value of $\text{CR}_{\beta,\sigma}.\mathcal{IR}.h$. It is possible that not every shard sends its shard input record to the BFT Core during the round, and the majority voting may fail (see the Consensus specification for details).

For every incoming certificate request CR $= \langle \alpha, \beta, \sigma, \nu; t, \mathcal{IR}, \ell_B, \ell_S \rangle$ the following checks are made (if any of them fails, CR is discarded):

1. CR.$\alpha = \alpha$ – request came from the same network instance
2. CR.$\beta \in C$ and $\sigma \in \mathcal{CD}$[CR.β].$\mathcal{SH}$ – request came from a legitimate shard of a registered partition
3. CR.$\nu \in \mathcal{SI}$[CR.β, CR.σ].$\mathcal{V}$ – request came from an authorized aggregation layer node
4. CR.$\mathcal{IR}.n = \mathcal{SI}[\beta, \sigma].n + 1$ – round number is correctly incremented
5. CR.$\mathcal{IR}.e = \mathcal{SI}[\beta, \sigma].e$ – epoch number matches
6. CR.$\mathcal{IR}.h' = \mathcal{SI}[\beta, \sigma].h_-$ – previous round's SMT root hash in IR matches the recorded one
7. $\mathcal{CD}[\beta].\gamma_{CP}$(CR.$\mathcal{IR}$, CR.$\pi_{CP}$) – consistency proof verification (Sec. 2.8.3) succeeds: verify VerifyConsistencyProof(CR.π_{CP}, CR.$\mathcal{IR}.h'$, CR.$\mathcal{IR}.h$) = TRUE and $|B|$ = CR.ℓ_B (batch size matches).
8. CR.$\mathcal{IR}.t = \mathcal{SI}[\beta, \sigma].UC_-.IR.C^{\mathrm{r}}.t$ – time reference is equal to the time field of the previous unicity seal

Only one (majority-voted) certificate request (denoted by $\mathrm{CR}_{\beta,\sigma}$) is accepted from every shard.

When the BFT Core's round n is completed, then:

1. For every $\beta \in C$, and $\sigma \in \mathcal{SH}_\beta$ with accepted certificate request [2]:
   - 1.1 $\mathcal{SI}[\beta, \sigma].n \leftarrow$ CR.$\mathcal{IR}.n$
   - 1.2 $\mathcal{SI}[\beta, \sigma].h_- \leftarrow$ CR.$\mathcal{IR}.h$
   - 1.3 $\mathcal{SI}[\beta, \sigma]$.SR.$n_e \leftarrow \mathcal{SI}[\beta, \sigma]$.SR.$n_e + 1$
   - 1.4 $\mathcal{SI}[\beta, \sigma]$.SR.$\overline{\ell}_B \leftarrow \mathcal{SI}[\beta, \sigma]$.SR.$\overline{\ell}_B +$ CR.$\mathcal{IR}.\ell_B$
   - 1.5 $\mathcal{SI}[\beta, \sigma]$.SR.$\overline{\ell}_S \leftarrow \mathcal{SI}[\beta, \sigma]$.SR.$\overline{\ell}_S +$ CR.$\mathcal{IR}.\ell_S$
   - 1.6 $\mathcal{SI}[\beta, \sigma]$.SR.$\hat{\ell}_B \leftarrow \max\{\mathcal{SI}[\beta, \sigma]$.SR.$\hat{\ell}_B$, CR.$\mathcal{IR}.\ell_B\}$
   - 1.7 $\mathcal{SI}[\beta, \sigma]$.SR.$\hat{\ell}_S \leftarrow \max\{\mathcal{SI}[\beta, \sigma]$.SR.$\hat{\ell}_S$, CR.$\mathcal{IR}.\ell_S\}$
   - 1.8 The leader for next round:
     $\mathcal{SI}[\beta, \sigma].\nu_\ell \leftarrow$ LEADERFUNC$(UC_-, \mathcal{SI}[\beta, \sigma].\mathcal{V})$
   - 1.9 Technical record:
     $\mathcal{TE}_{\beta,\sigma} \leftarrow (\mathcal{SI}[\beta, \sigma].n + 1, \mathcal{SI}[\beta, \sigma].e, \mathcal{SI}[\beta, \sigma].\nu_\ell, h_{\mathrm{sr}})$,
     where $h_{\mathrm{sr}} = H(\mathcal{SI}[\beta, \sigma]$.SR$_-, \mathcal{SI}[\beta, \sigma]$.SR$)$
2. For every registered partition $\beta \in C$, the collected certification requests $\mathrm{CR}_{\beta,\sigma}$ are converted to a temporary data structure $\mathcal{IRT}_\beta$ of type $(\mathbb{IR}, \mathbb{H})[\mathcal{SH}_\beta]$, where $\mathcal{SH}_\beta = \mathcal{CD}[\beta].\mathcal{SH}$, and $\mathcal{IRT}_\beta[\sigma] = (\mathrm{CR}_{\beta,\sigma}.\mathcal{IR}, H(\mathcal{TE}_{\beta,\sigma}))$;
   if there is no request then respective leaf repeats its previous value. If there is no previous value, the leaf is initialized to $\perp$.
3. For every $\beta \in C$, the shard tree χ_β (Sec. 3.1.5) is created by the function call $\chi_\beta \leftarrow$ CreateShardTree$(\mathcal{SH}_\beta, \mathcal{IRT}_\beta)$ (Sec. 3.1.5.1).

[2] Please refer to the Consensus chapter for details; notably a "request" may be induced for technical reasons

4. A temporary data structure $\mathcal{IH}$ of type $\mathbb{H}[C]$ is created such that $\mathcal{IH}[\beta] \leftarrow \chi_\beta(\lfloor\rfloor)$ for every $\beta \in C$, i.e. $\mathcal{IH}$ contains the root hashes of the shard trees.
5. The root of unicity tree is computed as $r \leftarrow$ CreateUnicityTree$(C, \mathcal{CD}, \mathcal{IH})$ (Sec. 3.1.6.1)
6. Unicity seal $C^{\mathsf{r}} = (\alpha, n, e, t, r_{-}, r; s)$ is created, where $t \leftarrow$ time() is the current time and s is a "signature" on all other fields. The form of s depends on the used consensus mechanism.
7. For every $\beta \in C$, the unicity tree certificate C_β^{uni} is created by the function call $C_\beta^{\mathsf{uni}} \leftarrow$ CreateUnicityTreeCert$(\beta, C, \mathcal{CD}, \mathcal{IH})$ (Sec. 2.7.3.1)
8. For every $\beta \in C$, and $\sigma \in \mathcal{SH}_\beta$:
   8.1 The shard tree certificate $C_{\beta,\sigma}^{\mathsf{shard}}$ (Sec. 2.7.2) is created by the function call $C_{\beta,\sigma}^{\mathsf{shard}} \leftarrow$ CreateShardTreeCert(σ, χ_β) (Sec. 2.7.2.2)
   8.2 The unicity certificate $UC_{\beta,\sigma} = (\mathcal{IRT}_\beta[\sigma].\mathcal{IR}, \mathcal{IRT}_\beta[\sigma].h_t, C_{\beta,\sigma}^{\mathsf{shard}}, C_\beta^{\mathsf{uni}}, C^{\mathsf{r}})$ (Sec. 2.7.5) and the certification response $\mathsf{CReS}_{\beta,\sigma} = \langle \alpha, \beta, \sigma; UC_{\beta,\sigma}, \mathcal{TE}_{\beta,\sigma} \rangle$ are composed (if the shard input have changed)
   8.3 The last unicity certificate field is updated by $\mathcal{SI}[\beta, \sigma]UC_{-} \leftarrow UC$ (if the shard input have changed).
9. The round number and the previous root hash of the unicity tree are updated by $n \leftarrow n + 1$ and $r_{-} \leftarrow r$.

When a shard's (identified by β, σ) epoch with at least one non-empty round ends, the following assignments are executed:

1. $\mathcal{SI}[\beta, \sigma].\mathsf{SR}_{-} \leftarrow \mathcal{SI}[\beta, \sigma].\mathsf{SR}$
2. $\mathcal{SI}[\beta, \sigma].\mathsf{SR} \leftarrow (0, 0, 0, 0, 0)$
3. $\mathcal{SI}[\beta, \sigma].e \leftarrow \mathcal{SI}[\beta, \sigma].e + 1$
4. The new validator set $\mathcal{SI}[\beta, \sigma].\mathcal{V}$ is chosen

4 BFT Core

4.1 Background

4.1.1 Definitions

Block is a set of requests, grouped together for mostly efficiency reasons, to be processed during one round. Unicity BFT Core does not produce an explicit blockchain – its certificates are persisted as proofs within the tokens.

UC is *Unicity Certificate*.

We call UC a **repeat UC** if it has incremented round number for a particular shard, but the certified hash has not changed compared to the UC of the previous round.

All aggregation layer nodes and BFT Core validators operate in **rounds**.

A round **extends** another finalized round by including its cryptographic hash as the hash of previous round.

The validators of a shard are synchronized based on input from the BFT Core. There are some fixed time-outs.

System has one BFT Core and an arbitrary number of partitions, which may be split into arbitrary number of shards.

Within a shard, there are k validators with identifier v, of which f might be faulty. For the BFT Core $k > 3f$. For a shard σ, $k_{\beta,\sigma} > 2f$. We assume that all faulty validators may be controlled by a coordinated, non-adaptive adversary. We assume trusted setup (Genesis) and authenticated data links (signed messages). We assume partially synchronous communication model where after unknown time GST message delivery time is upper-bounded by known Δ. We assume that in every shard, at least one non-faulty validator is able to persist its state.

A signature is denoted as s. Signed message with message name `name` is denoted as $\langle \texttt{name} \mid a, b, c; s \rangle$. Array of message fields is denoted as $\{f\}$.

Clients send **Unicity Service Requests** (requests).

4.1.2 Scope

Implementation details of BFT Core's atomic broadcast primitive (implementing the protocol `Ordering`) are not given. This is a modular component. Only safety-critical validation rules are provided.

4.1.3 Repeating Notation

n_r – round number of the BFT Core
$n_{\beta,\sigma}$ – round number of shard σ of partition β
$k_{\beta,\sigma}$ – number of aggregation layer nodes in shard σ of partition β, $k_{\beta,\sigma} = |\mathcal{V}_{\beta,\sigma}|$
v – aggregation layer node identifier, unique within the Unicity System instance; set of a shard's nodes is $\mathcal{V} = \{v_i\}_{i\in\{1,\ldots,k_\sigma\}}$
$v_l \leftarrow$ LEADERFUNC$(\cdot)$ – leader identifier for this round
h – SMT root hash
h' – previous SMT root hash

$ensure(\ldots)$ – function modeled after the Solidity language – if its argument evaluates to true then nothing happens; if it evaluates to false then execution stops and function returns 0. Unlike Solidity, should be complemented with returning and logging informative errors. Mostly used in message handlers for input validation.
$function(a, b \leftarrow c)$ – default value of function arguments, like in Python language. If 2nd argument is not specified by caller then parameter b obtains the value of expression c.

4.2 Aggregation Layer

4.2.1 Timing

An aggregation shard is synchronized using Input Records in returned UCs. For a shard, a UC can have the following options:

1. IR has not changed. Our shard can ignore this UC.
2. UC certifies an input from our shard, this input has never been certified before, and round number is incremented. This UC finalizes a round and starts a new round.
3. Round number is incremented, but state root hash remains the same (repeat UC). This UC starts another consensus attempt extending the same state as previous (likely failed) one.
4. UC is newer, certifying a future state. Indicates to a validator that it is behind the others and must roll back the pending proposal and initiate recovery.

If the latest UC certifies a state of this shard, then the certification response delivering UC determines the leader and starts a new round. While accepting incoming requests, the leader starts assembling his next block proposal, extending the latest finalized round (round with a valid UC).

When timer `t1` runs out the leader stops accepting new requests, finishes state updates and broadcasts a block proposal to followers and then sends Certification Request to the BFT Core. See Figure 4.

BFT Core has a timer `t2` for every shard within every partition; it is reset when a valid UC for this shard is issued. If this timer has run out, then a *repeat UC* is issued with incremented round number. This initiates a new consensus attempt for the shard. New round is executed with a different leader. Nodes can determine which UC is the latest based on round number. A block proposal, generated by the leader, includes a UC and this

UC must point to this leader. Round is finalized when its UC is embedded into the block. The retry mechanism is illustrated by Fig. 5.

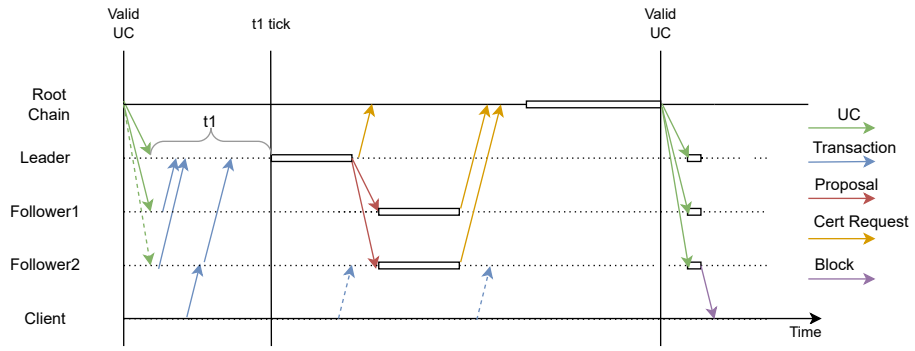

Figure 4. Successful Shard Round

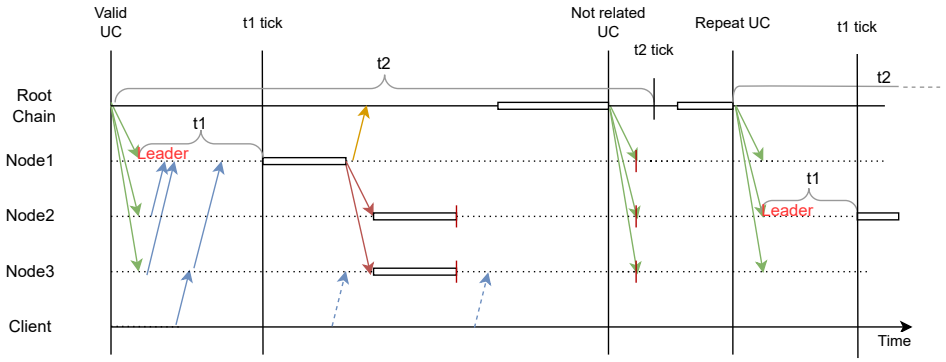

Figure 5. Shard Round attempt which did not produce a valid UC for this shard

4.2.2 Configuration and State

Configuration (managed by the Coordination Process) of every validator includes:

- Network Identifier (α).
- Partition Identifier (β).
- Shard Identifier (σ); present for multi-shard partitions.
- Aggregation Layer Node Identifier (ν). There are $k_{\beta,\sigma}$ nodes in shard σ of partition β.
- Timeout value `t1`: after a node sees a UC which appoints a leader, the leader waits for `t1` time units before stopping accepting new Unicity Service Requests and creating a block proposal.

Communication layer:

- secret key used to sign messages
- related public key; known to other validators and the BFT Core
- public keys of other validators within the shard
- communication addresses of other validators within the shard

- communication addresses of the BFT Core validators

Data layer:

- Unicity *Trust Base* ($\mathcal{T}$)
- other *partition* defining parameters; refer to Unicity Platform Specification, State of a Shard.

Variables:

- v_l: Current round leader's identifier; NULL if not known
- buf: buffer with pending Unicity Service Requests
- N: State Tree
- cp: State Tree checkpoint, helps the State Tree to roll back to previously certified state if a state extending attempt fails. Checkpoints can be released when a following round gets finalized.
- luc: latest UC. Importantly, this structure encapsulates the *state hash* of the last certified state, to be extended by the round finalization attempt.
- lte: latest Technical Record, certified by luc. Carries the required *round number*, leader identifier, epoch of the round finalization attempt.
- log: log of verified and executed (but not final) requests; respective changes in State Tree can be rolled back by reverting it to checkpoint cp.
- pr: Pending Certification Request waiting for UC; includes state tree hash and applied requests and round number as the time reference used for request validation. We are avoiding situation where there can be multiple pending requests and speculative validation; fresh UC invalidates all pending requests.
  There may be multiple parallel pending requests in future extensions.
- $\mathcal{B}$: the shared append-only log.

The variables are handled via state transitions like this:

1. *Initial state*. State tree is certified with 'luc'; log and pr are empty; cp points to the current state.
2. *After applying any request(s)*: There are changes in the state tree; executed requests are recorded in $proposal$ (that is, cp is the starting point and N and log are updated in sync). Processing of requests continues.
3. *Waiting for UC*. This state is reached on t1 click, after sending a BlockProposalMsg message (if being the leader) and sending a Certification Request. There are changes in the state tree on top of snapshot cp; executed requests were recorded in log; Now, root hash of the state tree and respective log are saved in pr, which extends luc. pr must be preserved as long as it is possible that it gets certified by a UC. No new requests are processed in this state.
4. *After receiving a CReS message with new UC*:
   - *UC certifies* pr: block is finalized and added to $\mathcal{B}$. OK to clear.
   - *UC is 'repeat UC'*: state is rolled back to cp; we assume simplified case that consensus for prev. request is not possible any more and clean pr.

Algorithm 1 State and Initialization

Constants:
α: Network Identifier
β: Partition Identifier
σ: Shard Identifier
ν: Node Identifier
$k_{\beta,\sigma}$: number of nodes in the shard σ of partition β
$\mathcal{T}$: Unicity Trust Base, may *evolve*
Variables:
$\nu_l \leftarrow$ NULL: leader node identifier of the current round; NULL if not known
$N \leftarrow \{\}$: SMT (Sparse Merkle Tree)
cp $\leftarrow \perp$: SMT Checkpoint
$luc \leftarrow$ NULL: latest valid UC
$lte \leftarrow$ NULL: latest technical record sent with latest UC
▷ <*round number to be certified*> = $lte.n$
▷ <*last certified hash*> = $luc.IR.h$
$buf \leftarrow \{\}$: input requests buffer
$log \leftarrow \{\}$: executed requests log for proposal creation
$pr \leftarrow$ NULL: pending proposal corresponding to a CR request waiting for UC
sr: statistical record data. See Statistical Record
ℓ_B: number of new SMT leaves added during round
ℓ_S: SMT memory usage in bytes
π_{CP}: optional consistency proof (hash-based or ZK-compressed)
$\mathcal{B}$: shared log

function START_NEW_ROUND(uc, te)
  ensure(VERIFYUNICITYCERT($uc, \mathcal{T}$))
  ensure($uc.h_t = h(te)$)
  ensure($te.n > luc.IR.n$)
  ensure($uc.IR.h' = luc.IR.h$) ▷ Double-checking
  cp $\leftarrow$ CHECKPOINT(N)
  RINIT()
  $log \leftarrow \{\}$
  $pr \leftarrow \{\}$
  $\nu_l \leftarrow te.\nu_l$
  $luc \leftarrow uc, lte \leftarrow te$
  RESET_TIMER(t1)
  if $\nu_l = \nu$ **then** ▷ Leader
    PROCESS(buf) ▷ Process for no longer than until t1 tick
    $buf \leftarrow \{\}$
  else ▷ Follower
    if SEND_INPUTFORWARDMSG(l, buf) **then**
      $buf \leftarrow \{\}$ ▷ Clean buffer on successful connection
    end if
  end if
end function

function LEADERFUNC(uc)
  return $\{\nu_i \mid i \leftarrow \text{integer}(H(uc)) \bmod k_{\beta,\sigma} + 1\}$ ▷ Simplest example
end function

- *UC certifies any round newer than the latest known UC*: rollback and recovery (independent state, consuming blocks until N is up-to-date with UC).

5. Loop to 1.

Please refer to Algorithm 1 for initialization.

4.2.3 Subcomponents

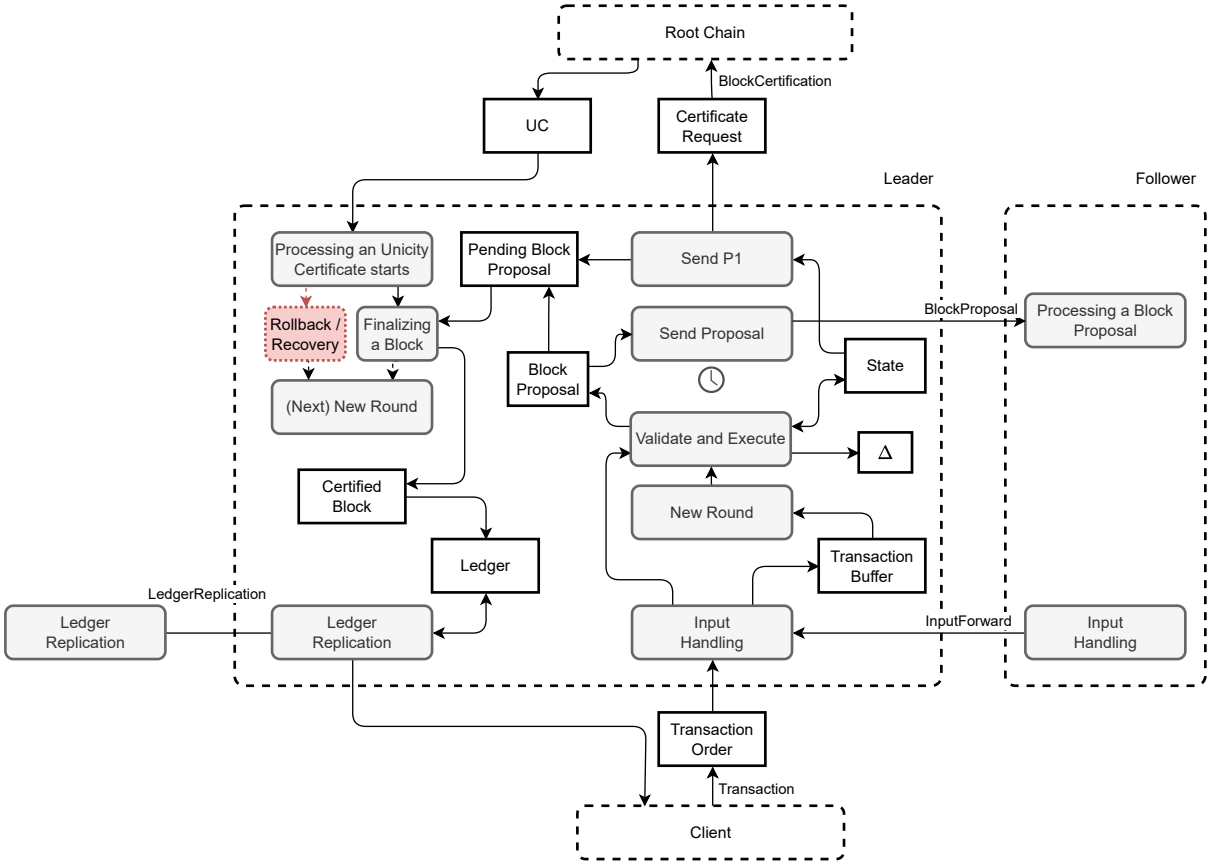

Figure 6. Data Flow of the Shard Leader Node

4.2.3.1 Input Handling

Input Handling prioritizes latency (fast finality). It is optimized for the case where there is enough processing capacity available and Unicity Service Requests do not have to be queued.

All validators accept Unicity Service Requests from clients. It is expected, that clients send Unicity Service Requests to many validators, as some may be byzantine. We assume that clients send Unicity Service Requests to the right shard; Unicity Service Requests sent to wrong shard can be discarded. Optionally, implement QoS / overload protection. There is no guarantee of execution – the validators may drop Unicity Service Requests to protect the system availability, or when working close to maximum capacity. Synchronized clients may send Unicity Service Requests directly to the expected leader.

Shard validators forward Unicity Service Requests, as they arrive, to respective shard leaders (node producing the next block proposal); while observing time-outs and discarding

expired Unicity Service Requests. There can be a light-weight partial validation, referred as *sanity check*, before continuing with the processing. If the leader is not known, or rejects messages, then keep Unicity Service Requests in a buffer and try again when the next leader is known and accepts Unicity Service Requests. If the validator is the current leader, then he processes available Unicity Service Requests immediately. At the moment when a leader can not include transactions into a proposal anymore, or have collected enough transactions to fill a block, it starts rejecting incoming Unicity Service Requests from other validators.

A validator should retain a Unicity Service Request if accepted from a client or other validator; until it is either expired or included into a finalized block. A validator may forget a Unicity Service Request if accepted by another validator. A validator should not forward a Unicity Service Request to a distinct validator more than once.

Validators may limit the number of times a Unicity Service Request is forwarded.

Please refer to Algorithm 2 for an example without optional functionality.

4.2.3.2 Block Proposal

Summary: On clock tick, stop immediate validation and execution of incoming Unicity Service Requests. Validate and execute Unicity Service Requests from the Request Buffer, updating the State Tree (N) and log for proposal creation. Executed requests from log go into Block Proposal, in the exactly same order they were validated and executed. Broadcast Block Proposal to Follower Nodes. Create and send Uniqueness Certificate Request, retaining necessary state in a Pending Block Proposal (pr) data structure.

A round must extend a previously certified round. If a party approves a block proposal, then it also approves the entire history. This ensures safety of the protocol.

Pending Block Proposal (pr) must be stored in durable way before Certification Request can be sent, by e.g. writing it to persistent storage. Losing all copies of pending block proposals, while obtaining a UC for this round, would be an unrecoverable error.

Please refer to Algorithm 3 for details.

Note that a block can be without any requests; however, this does not necessarily imply $h' = h$, as system-initiated "housekeeping" actions may have changed the state.

4.2.3.3 Validation and Execution

Sanity checking of Unicity Service Requests is quick and lightweight validation, with the main goal of protecting system resources by early detection of obvious garbage. All Unicity Service Requests will be fully verified later before actual execution. Thoroughness of sanity checking is a tuning parameter.

Validating Unicity Service Requests is performing their full verification, according to Unicity Platform Specification, section Unicity Service Request Validation, and performing partition specific additional checks. The requests must appear in the *proposal* in the same order. Requests without interdependencies (i.e., affecting distinct state) can be executed in parallel. Invalid requests are not executed and not included into produced proposal.

Algorithm 2 Input Handling

upon message <TransactionMsg | Q> **do**
  if SANITY_CHECK(Q) **then**
    if $v_l = v$ **then** ▷ This process is the leader
      PROCESS($\{Q\}$) ▷ Beware of parallel execution
    else if $v_l \neq$ NULL **then** ▷ We know someone else is the leader
      if ¬SEND_INPUTFORWARDMSG(v_l, Q) **then** ▷ Forward to v_l
        $buf \leftarrow buf \cup Q$ ▷ Store on failure
      end if
    else ▷ Buffer requests until leader is known
      $buf \leftarrow buf \cup Q$
    end if
  end if
end upon

upon message <InputForwardMsg | $reqs$> **do**
  if $v_l = v$ **then** ▷ This process is the leader
    defer PROCESS($reqs$)
    return "accepted"
  else
    return "reject"
  end if
end upon

on event next_leader_elected **do**
  PRUNE_EXPIRED(buf)
  if $v_l = v$ **then** ▷ This process is the leader
    PROCESS(buf) ▷ Beware of parallel execution
  else
    if SEND_INPUTFORWARDMSG(v_l, buf) **then** ▷ Forward to v_l
      $buf \leftarrow \{\}$ ▷ Forget on successful send
      ▷ . . . or keep in "forwarded" buffer and use when becoming the leader
    end if
  end if
end on

Algorithm 3 Producing a Block Proposal

on event t1 **do**
  if $v_l = v$ **then** ▷ This validator is the leader
    $v_l \leftarrow$ NULL
    RCOMPL() ▷ Request processing must have stopped by now
    $sr \leftarrow$ PRODUCESTATISTICS()
    SEND_BLOCKPROPOSALMSG($\alpha, \beta, \sigma, v, luc, lte, log, sr$) ▷ Sign and Broadcast
    DO_CERT_REQ(log, v, sr)
  end if
  $v_l \leftarrow$ NULL ▷ Leader stops accepting new txs
end on

Algorithm 4 Producing a Certification Request

function DO_CERT_REQ(txs, ν_l, sr)
  $h' \leftarrow luc.IR.h$, $n \leftarrow lte.n$, $e \leftarrow lte.e$, $t \leftarrow luc.C^r.t$
  $h \leftarrow$ STATEROOT(N)
  $\ell_B \leftarrow$ count of new SMT leaves added, $\ell_S \leftarrow$ SMT memory usage
  $pr \leftarrow (n, e, h', h, t, \nu_l, log, sr)$ ▷ Pending Block Proposal
  if STORE_IN_DURABLE_WAY(pr) **then**
    $IR \leftarrow (n, e, h', h, t)$
    $\pi_{CP} \leftarrow$ GENERATECONSISTENCYPROOF(h', h, batch) ▷ Generate consistency proof
    SEND_CR($\alpha, \beta, \sigma, \nu, IR, \ell_B, \ell_S, \pi_{CP}$) ▷ Sign and send
  else
    ▷ Do nothing as the current node can not guarantee data availability
    ▷ The round may get finalized though thanks to other partition nodes
  end if
end function
function DO_CERT_REQ_AGAINST(sr) ▷ Voting against the proposal
  $h' \leftarrow luc.IR.h$, $n \leftarrow lte.n$, $e \leftarrow lte.e$, $t \leftarrow luc.IR.C^r.t$
  $h \leftarrow \perp$
  $IR \leftarrow (n, e, h', h, t)$
  $\ell_B \leftarrow 0$, $\ell_S \leftarrow 0$ ▷ No additions when voting against
  SEND_CR($\alpha, \beta, \sigma, \nu, IR, \ell_B, \ell_S, \perp$) ▷ No consistency proof needed
end function

Algorithm 5 Validate and Execute Unicity Service Requests

function SANITY_CHECK($Q; t \leftarrow lte.n$) ▷ Requests will be fully validated later
  return Q.pk $\neq$ NULL ▷ Basic validation of request structure
end function

function VALIDATE($Q; n \leftarrow lte.n, t \leftarrow luc.C^r.t$)
  ▷ Omitted, see Platform Specification, Unicity Service Request Validation
end function

function PROCESS($reqs$) ▷ Should be implemented as processing queue
  for all $Q \in reqs$ **do**
    if VALIDATE(Q) **then**
      EXECUTE(N, Q) ▷ Can be executed in parallel
      $log \leftarrow log \cup Q$
    end if
  end for
end function

The expiration of Unicity Service Requests is checked relative to shard round number. Validation context includes also the time value t obtained from previous UC and to be recorded as current round's $UC.IR.t$.

Refer to Algorithm 5 for details.

4.2.3.4 Processing an Unicity Certificate and Finalizing a Round

Summary: On receiving a UC, round is finalized, and a new round is started.

More specifically,

1. UC is verified cryptographically according to the Framework Specification. Partition and shard identifiers are checked.
2. The time-stamp in UC is checked for sanity: it must not "jump around" and it must reasonably match the local time if it can be reliably determined. UC with a suspicious time-stamp must be logged, and processing continues because rejecting a UC may end with a deadlock of the shard.
3. UC consistency is checked:
4. UC is checked for equivocation, that is, for arbitrary uc and uc', the following must hold:

$$\begin{aligned}
uc.IR.n = uc'.IR.n &\Rightarrow uc.IR = uc'.IR \\
uc.IR.h' = uc'.IR.h' &\Rightarrow uc.IR.h = uc'.IR.h \\
&\quad \vee uc.IR.h' = uc.IR.h \vee uc'.IR.h' = uc'.IR.h \\
uc.IR.h = uc'.IR.h &\Rightarrow uc.IR.h' = uc'.IR.h' \\
&\quad \vee uc.IR.h' = uc.IR.h \vee uc'.IR.h' = uc'.IR.h \\
uc.IR.n = uc'.IR.n + 1 &\Rightarrow uc.IR.h' = uc'.IR.h \\
uc.IR.n < uc'.IR.n &\Rightarrow uc.C^{\mathrm{r}}.n < uc'.C^{\mathrm{r}}.n
\end{aligned}$$

   On failing any of these checks, an equivocation proof must be logged with all necessary evidence.
5. UC round number and epoch number can not decrement.
6. On unexpected case where there is no pending round certification request, recovery is initiated, unless the state is already up-to-date with the UC.
7. Alternatively, if UC certifies the pending round certification request then round is finalized.
8. Alternatively, if UC certifies the round whose pending proposal tried to extend ('repeat UC') then state is rolled back to the previous state.
9. Alternatively, recovery is initiated, after rollback. Note that recovery may end up with newer last known UC than the one being processed.
10. Finally, on valid UC (validation reached the 3 alternatives above), a new round is started.

Please refer to Algorithm 6 for details. Round Finalization is presented in Algorithm 8.

On arbitrary timeout / *lost connection*: re-establish connection to the BFT Core.

If a block can not be saved and made available during the finalization, then the round must be not closed. This ensures that a) the block can be restored based on saved proposal and UC during the recovery process, and b) the node can not extend the round with non-persisted block of inputs.

Algorithm 6 Processing a received Unicity Certificate

upon message <CReS | $\alpha, \beta, \sigma, te; UC$> **do**
  ensure(VerifyUnicityCert($uc, \mathcal{T}$))
  ensure($H(te) = UC.h_t$)
  if GOT_NEW_UC(UC) **then**
    START_NEW_ROUND(UC, te)
  end if
end upon
function GOT_NEW_UC(uc)
  ensure($uc.IR \neq luc.IR$) ▷ Ignore UC certifying the same
  ensure($uc.C^{\mathrm{r}}.\alpha = \alpha$)
  ensure($uc.C^{\mathrm{uni}}.\beta = \beta$)
  if ¬CheckSanity($uc.C^r.t$, $uc.C^r.n$, GetUTCDateTime()) **then**
    Log(uc) ▷ Rejecting a UC with strange time would break the shard
  end if
  ensure(NON_EQUIVOCATING_UCS(uc, luc))
  ensure($uc.IR.n > luc.IR.n$) ▷ Check late to catch equivocation
  if pr = Null **then** ▷ No pending Certification Request
    if $uc.IR.h \neq$ StateRoot(N) **then**
      RECOVERY(uc)
    end if
  else
    if $uc.IR.h = pr.h \wedge uc.IR.h' = pr.h'$ **then**
      FINALIZE_ROUND(pr, uc)
    else if $uc.IR.h = pr.h'$ **then**
      Revert(N, cp)
    else
      Revert(N, cp)
      RECOVERY(uc)
    end if
  end if
  return 1
end function

4.2.3.5 Processing a Block Proposal

Summary: Upon receiving a BlockProposalMsg message, create a rollback checkpoint and then validate the signature and header fields, execute Unicity Service Requests from the proposal and updating the State Tree (N). Executed requests go into Block Proposal. Create and send Uniqueness Certificate Request message, and retain a Pending Block Proposal data structure.

This procedure is performed by the non-leader validators. There are following steps (See Fig. 7):

1. Block proposal is checked: valid signature, correct partition and shard identifier, valid UC, the UC must be not older than the latest known by the validator. Sender must be the leader for the round started by included UC and match the leader identifier field.

Algorithm 7 Checking two UC-s for equivocation

function NON_EQUIVOCATING_UCS(uc, uc')
  ensure($uc.IR.n \geq uc'.IR.n$) ▷ To simplify, assume that uc is not older than uc'
  ensure($uc.C^{\mathrm{r}}.n_r > luc.C^{\mathrm{r}}.n_r$)
  if $uc.IR.n = uc'.IR.n$ **then**
    ensure($uc.IR = uc'.IR$) ▷ On all failures log uc and uc' as proof
  end if
  if $uc.IR.h' = uc'.IR.h' \wedge uc'.IR.h' \neq uc'.IR.h$ **then**
    ensure($uc.IR.h = uc'.IR.h$) ▷ A hash can be extended only with one hash
  end if
  if $uc.IR.h = uc'.IR.h$ **then** ▷ . . . and vice versa
    ensure($uc.IR.h' = uc'.IR.h'$)
  end if
  if $uc.IR.n = uc'.IR.n + 1$ **then**
    ensure($uc.IR.h' = uc'.IR.h$)
  end if
  return 1
end function

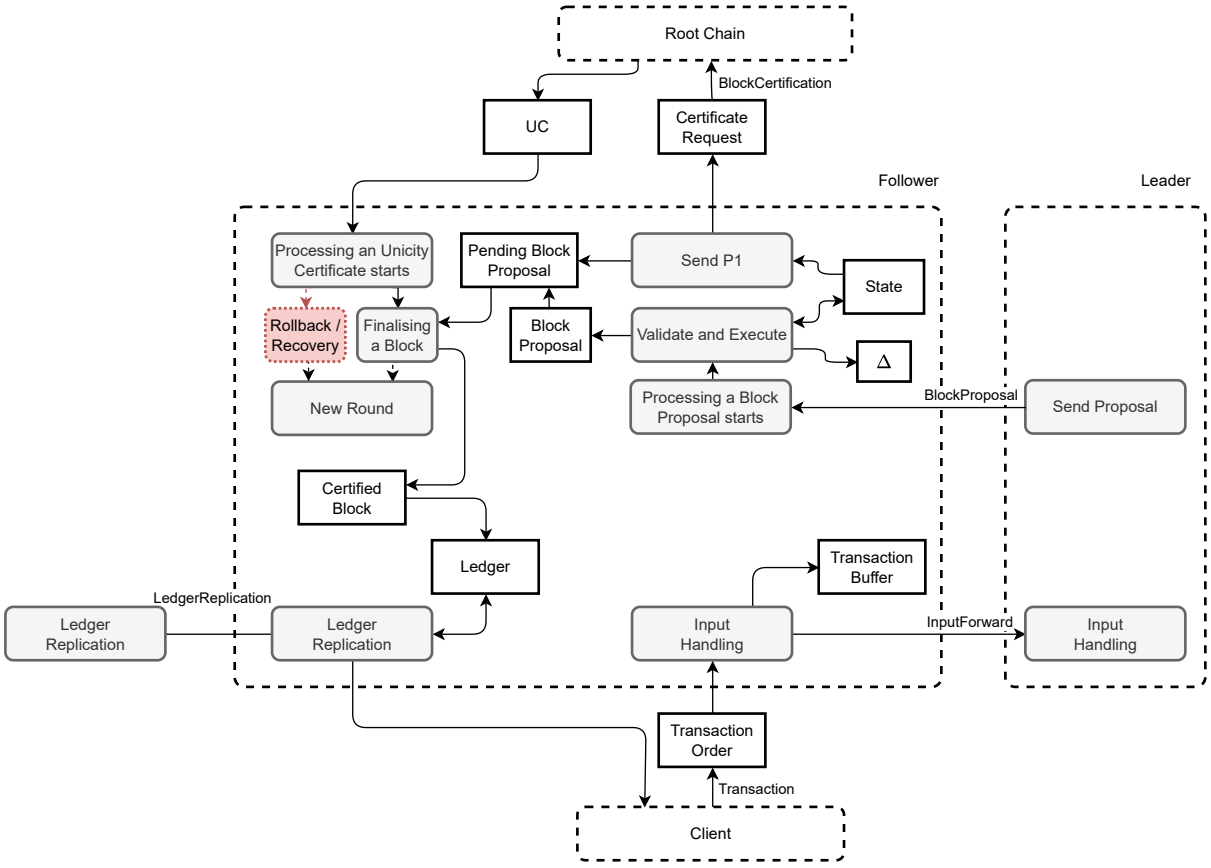

Figure 7. Data Flow of a non-leader aggregation shard node

Algorithm 8 Finalizing a Round

function FINALIZE_ROUND(pr, uc)
  $\mathcal{B} \leftarrow \mathcal{B} \cup (pr.txs; uc)$ ▷ Recording a new batch to shard's log
end function

2. If included UC is newer than latest UC then the new UC is processed; this rolls back possible pending change in state tree. If new UC is 'repeat UC' then update is reasonably fast; if recovery is necessary then likely it takes some time and there is no reason to finish the processing of current proposal.
3. If the state tree root is not equal to one extended by the processed proposal then processing is aborted and negative vote is delivered.
4. All Unicity Service Requests in proposal are validated; on encountering an invalid Unicity Service Request the processing is aborted and negative vote is delivered.
5. Transaction orders are executed by applying them to the state.
6. Pending certificate request data structure is created and persisted.
7. Certificate Request query (CR) is assembled and sent to the BFT Core—that is, positive vote for the proposal.

As an optimization, it is possible to vote against a proposal by sending a negative vote, where proposed state's hash is $\perp$. This helps the BFT Core to determine that convergence to consensus is not possible before waiting until time-out.

Please refer to Algorithm 9 for details.

4.2.3.6 Ledger Replication

Relatively independent subsystem for serving and replicating ledger data. Pseudocode of the service is provided in Algorithm 10.

On receiving log, the blocks are verified using embedded UC-s and cryptographic links. See Platform Specification, function VerifyBlock().

4.2.4 Recovery Procedure

If a validator is behind then it must use recovery procedure to sync its state with other validators, and obtain the latest UC for this shard, whose authoritative source is the BFT Core.

Summary: Missing log entries are fetched from other validators, validated, and applied to the state tree. A pending block proposal, if certified but not finalized, is applied and finalized.

It is assumed that the state tree is already rolled back by calling REVERT(N, cp) if it had requests of a not finalized block applied.

In more details:

1. Input UC is validated,
2. Missing log entries are fetched from other (random) validator(s),

Algorithm 9 Processing a received Block Proposal

upon message <BlockProposalMsg | $m = (\alpha, \beta, \sigma, v, uc, te, txs, sr; s)$> **do**
  ensure(VALID(m)) ▷ Consistent and authorized message
  ensure(VERIFYUNICITYCERT(uc, T))
  ensure($uc.h_t = h(te)$)
  ensure($m.v = te.v_l$) ▷ Signed by authorized leader
  ensure($m.\alpha = \alpha \wedge m.\beta = \beta \wedge m.\sigma = \sigma$)
  ensure($m.te.n \geq lte.n$)
  if ¬CHECKREQUESTSTATISTICS(sr) **then**
    DO_CERT_REQ_AGAINST(NULL) ▷ Invalid data, vote against
    return
  end if
  if $m.te.n > lte.n$ **then**
    ensure(GOT_NEW_UC($m.uc, m.te$)) ▷ Newer UC must be validated and processed
    if *processing of new UC took too much time (recovering?)* **then**
      return START_NEW_ROUND($m.uc, m.te$)
    else
      $luc \leftarrow m.uc, lte \leftarrow m.te$
    end if
  end if
  $h' \leftarrow m.uc.IR.h$
  if STATEROOT(N) $= h' \wedge \{ \forall\ Q \in m.reqs \mid$ VALIDATE(Q)$\}$ **then**
    cp $\leftarrow$ CHECKPOINT(N)
    RINIT()
    for all $Q \in m.reqs$ **do**
      EXECUTE(N, Q)
    end for
    RCOMPL()
    DO_CERT_REQ($m.reqs, m.v, sr$) ▷ Vote for
  else
    DO_CERT_REQ_AGAINST(sr) ▷ Vote against
  end if
end upon

Algorithm 10 Ledger Replication

upon message <LedgerReplication | $\alpha, \beta, \sigma, n_1, n_2 \leftarrow luc.IR.n; s$> **do**
  ▷ First authorization and sanity check,
  return $\{\mathcal{B}_{\alpha,\beta,\sigma,n} \mid n \in [n_1 \,..\, n_2]\}$
end upon

3. Each request block is verified: cryptographically using embedded UC, and for correct partition and shard ID;
4. Each request within the block is validated,
5. Requests are applied to the state tree,
6. Last known UC is updated if a block has newer one.

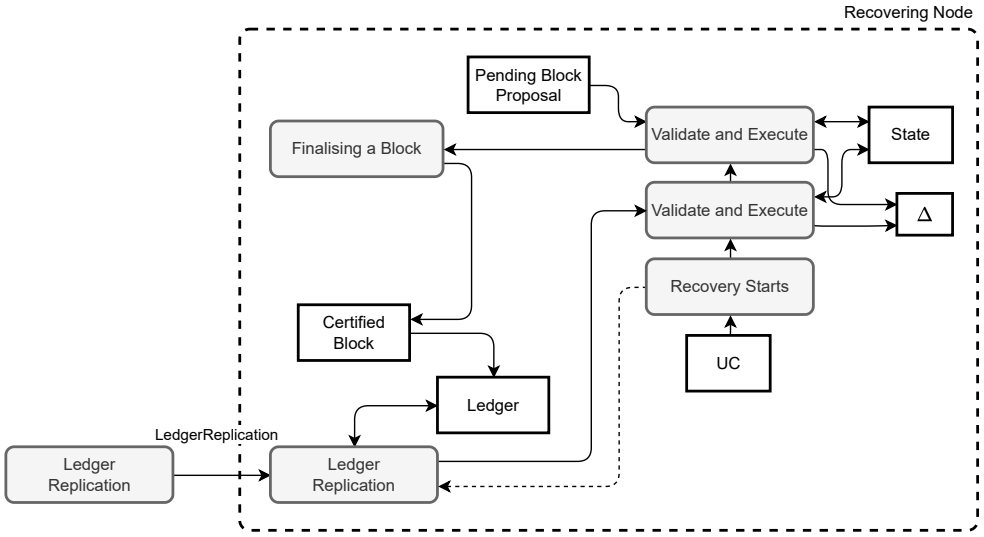

Figure 8. Data Flow of an out-of-sync (recovering) Shard Node

7. Then, if there is a pending block proposal which can be finalized using freshly obtained UC then it will be applied to current state and round is finalized.

Please refer to Algorithm 11 for full details. Recursive recovery is used to mark locations where last-resort failover/retry happens. More intelligent failover and back-off mechanism could be used, with gracious shut-down on unrecoverable situations.

4.2.5 Protocols – Shard Validators

4.2.5.1 Protocol TransactionMsg – Unicity Service Request Delivery

Users deliver their Unicity Service Requests to one or more validators (to account for byzantine validators censoring or re-ordering requests).

Message: $\langle \texttt{TransactionMsg} \mid Q \rangle$

4.2.5.2 Protocol CR – Round Certification Request

This section extends Sec. 3.3.1, Certification Request).

If h' is already 'extended' with UC then the latest UC is returned immediately via CReS message; otherwise validation and UC generation continues, UC is returned via CReS when available.

If h' is unknown to BFT Core then the latest UC is returned immediately via CReS.

Returned message has a technical data record, which may trigger a view change: next consensus attempt with incremented round number and different leader. An aggregation layer node can have only one pending CR per round number; subsequent messages are ignored. A message with higher round number is always preferred.

This message "subscribes" the validator to receive UC messages for a certain period, either a fixed number (e.g. 2 rounds), or until the shard have successfully proposed a following

Algorithm 11 Shard Node Recovery

```
function RECOVERY(uc)                                   ▷ Assuming that Revert() is done by caller
    ensure(VERIFYUNICITYCERT(uc, 𝒯))
    for all b ∈ SEND_LEDGERREPLICATIONREQUEST(luc.IR.n + 1) do ▷ To a random live validator
        if VERIFYBLOCK(b, 𝒯) then                        ▷ Assuming request blocks are ordered
            ensure(b.α = α ∧ b.β = β ∧ b.σ = σ)
            ensure(b.UC.IR.n > luc.IR.n)
            ensure(STATEROOT(N) = luc.IR.h = b.IR.h′)
            ensure({ ∀ T ∈ b.txs | VALIDATE(T, luc.IR.n + 1)})
            cp ← CHECKPOINT(N)
            RINIT()
            for all T ∈ b.txs do
                EXECUTE(N, T)
            end for
            RCOMPL()
            if STATEROOT(N) ≠ b.IR.h then
                REVERT(N, cp)
                return RECOVERY(uc)                     ▷ Failover
            end if
            ℬ ← ℬ ∪ b
            luc ← b.UC                                  ▷ Respective lte arrives with proposal
        else
            return RECOVERY(uc)                         ▷ Failover
        end if
    end for
    if uc.IR.h′ = STATEROOT(N) then                     ▷ Apply pending request if possible
        pr ← FETCH_PR_FROM_PERSISTENT_STORAGE(pr)
        if pr ≠ NULL ∧ pr.h′ = uc.IR.h′ ∧ pr.h = uc.IR.h then
            ensure(uc.IR.n = pr.n)
            ensure({ ∀ T ∈ pr.txs | VALIDATE(T, pr.n)})
            cp ← CHECKPOINT(N)
            RINIT()
            for all T ∈ pr.txs do
                EXECUTE(N, T)
            end for
            RCOMPL()
            if STATEROOT(N) = uc.IR.h then
                FINALIZE_BLOCK(pr, uc)
                luc ← uc
            else
                REVERT(N, cp)
            end if
        end if
    end if
end function
```

round, that is, there is another set of BFT Core validators which have received a quorum of CR messages and therefore “taken over” the subscription.

Message: $\langle \texttt{CR} \mid \alpha, \beta, \sigma, \nu, IR, sr; s \rangle$

If an aggregation layer node has reasons to suspect that BFT Core have generated a new UC, then he must try to fetch it by trying again. sr stands for Statistical Record; a technical data structure sent from aggregation layer nodes to the BFT Core.

4.2.5.3 Protocol RoundCertificationResponse (CReS)

This section extends Sec. 3.3.2, Certification Response).

CReS is asynchronous response (in the sense of data flow) to Certification Request (CR); there may be many CReS responses to one client request. TE stands for Technical Record, sent from the BFT Core to aggregation layer nodes. Valid if $UC.h_t = h(\text{TE})$.

Optional field RootTrustBaseEntry indicates, that UTB $\mathcal{T}$ have changed, e.g., have grown by the provided entry.

Message: $\langle \texttt{CReS} \mid \alpha, \beta, \sigma, UC, \text{TE}, [\text{RootTrustBaseEntry}] \rangle$

4.2.5.4 Protocol Subscription – subscribing to CReS messages

This message “subscribes” the validator to future CReS messages, without presenting a Certification Request in the form of CR message. Synchronous response is the latest UC for requestor.

Subscription ends when the shard have successfully proposed a following round, that is, there is another set of BFT Core validators which have received a quorum of CR messages and therefore serving a subscription.

Query: $\langle \texttt{SubscriptionMsg} \mid \alpha, \beta, \sigma, \nu; s \rangle$

Response: $\langle \texttt{CReS} \rangle$

4.2.5.5 Protocol InputForwardMsg – Input Forwarding

Forward a set of Unicity Service Requests.

Message: $\langle \texttt{InputForwardMsg} \mid \{Q\} \rangle$

4.2.5.6 Protocol BlockProposalMsg – Block Proposal

Leader broadcasts its input block proposal to other partition nodes.

Message: $\langle \texttt{BlockProposalMsg} \mid \alpha, \beta, \sigma, \nu_l, uc, te, txs, sr; s \rangle$
where $txs = \{T\}$

4.2.5.7 Protocol LedgerReplication – Ledger Replication

Let’s assume that we have a separate layer of components implementing the ledger storage. Entire ledger can be verified based on latest finalized and available round and every block of inputs can be verified based on the Unicity Trust Base.

This protocol is provided by every functional aggregation layer node and dedicated *archive nodes*; arbitrary parties can join the latter.

Query: $\langle \texttt{LedgerReplication} \mid (\alpha, \beta, \sigma n_1, [n_2] \rangle$

Response: $(\{B\})$

If 2nd number is missing then return everything till head. It is possible, that a reply misses some newer rounds, either because the queried node is behind or it prefers to return input blocks by smaller chunks.

4.2.5.8 Protocol GetTrustBase – Unicity Trust Base Distribution

The Unicity Trust Base is a chain of records, one per epoch, containing the BFT Core validator set configuration and forming a verifiable root of trust. This protocol allows nodes to obtain Trust Base entries for a specified epoch range.

Each Trust Base entry corresponds to one epoch and contains validator identities, stakes, quorum requirements, and cryptographic linkage to the previous epoch's entry through hashing and signatures. The chain structure enables verification: given an authentic base entry, subsequent entries can be verified by checking signatures and hash chains.

Nodes request Trust Base entries by specifying an epoch range. The response includes all entries from the starting epoch up to (and including) the ending epoch, or up to the latest available entry if no ending epoch is specified.

Query: $\langle \texttt{GetTrustBase} \mid \alpha, e_1, [e_2] \rangle$

where:

- α – network identifier of type $\mathbb{A}$
- e_1 – starting epoch number of type $\mathbb{N}_{64}$
- $[e_2]$ – optional ending epoch number of type $\mathbb{N}_{64}$

Response: $\langle \texttt{TrustBaseResponse} \mid \mathcal{T} \rangle$

where $\mathcal{T} = \langle T_{e_1}, T_{e_1+1}, \ldots, T_{e_2} \rangle$ is a sequence of Trust Base entries.

Each Trust Base entry T_e (for epoch e) is a tuple:

$$T_e = (\alpha, e, n_e, \{v, b_v\}_e, k_e, r, h_{\text{cr}}, h_{e-1}, s_{e-1})$$

as defined in Table 3.

If the ending epoch e_2 is omitted, the response includes all entries from e_1 up to the latest committed epoch. If the requesting node specifies an epoch range beyond what is currently available, the response includes entries up to the latest committed epoch only.

Verification: Upon receiving Trust Base entries, the requester must verify the chain:

1. Verify that the first entry T_{e_1} connects to an already-authenticated Trust Base entry (either the genesis entry or a previously verified entry)
2. For each subsequent entry $T_{e_{i+1}}$, verify:
   2.1 $h_{e_i} = H(T_{e_i})$ (hash chain linkage)

2.2 Signature s_{e_i} is valid over $T_{e_{i+1}}$ using validator set from epoch e_i

2.3 Epoch numbers increment correctly: $T_{e_{i+1}}.e = T_{e_i}.e + 1$

This protocol is provided by all active BFT Core validators. Validators maintain the complete Trust Base chain from genesis and serve requests for any epoch range. The protocol supports both initial synchronization (requesting from genesis) and incremental updates (requesting only recent epochs).

4.3 BFT Core State Machine

4.3.1 Summary

Leader-based BFT consensus based SMR. Roughly, BFT Core validators:

1. Validate incoming Certification Requests: signature correctness, they must extend shard's previous UC, and partition specific checks must pass.
2. Forward shard's requests to the BFT Core leader
3. BFT Core leader verifies shard's requests (incl. majority), produces UC tree, signs.
4. BFT Core leader sends requests (all Certification Requests including signatures) and trees and signature to BFT Core validators.
5. Followers verify shard's requests (incl. majority), create trees, sign.
6. Followers distribute their signatures to other BFT Core validators
7. On reaching $k - f$ (there are k validators in the BFT Core) unique signatures all BFT Core validators send `ack` to others
8. On reaching $k - f$ `ack`-s all BFT Core validators commit and return responses to the aggregation layer nodes.

4.3.2 Timing

BFT Core serves many partitions and shards, each implemented as a logical group of nodes. Operation cycles work like this:

1. Shards operate in parallel:
   - Shard validators send RoundCertificationRequest requests.
   - Once there is a quorum of requests for a shard, the BFT Core updates an entry in the array of input records.
2. *Eventually*, BFT Core starts Unicity Certificate generation. Fills data structure. Computes root. Signs. In distributed case this all takes some time.
3. Individual tree certificates and UCs are generated for participating partitions. Responses are returned to individual validators.

'Eventually' above is a compromise: 1) there is no sense to start a new round too fast, it is necessary to collect some input requests to certify; 2) input requests with quorum should be served as fast as possible to improve latency; 3) no need for further wait if all inputs are present; 4) some inputs might struggle with quorum; no need to wait for the long tail; 5) in order to generate 'repeat UCs' a round must be restarted after `t2` time units even when there are no requests.

Configuration parameters are: target round rate t_b and shard wait `t1`.
System parameters are: average root Certificate generation time t_r; average shard round processing time from receiving UC to sending Certification Request, including `t1` wait: t_p; with standard deviation σ_p. There are k shards.

Let m – how many BFT Core rounds fit into one shard round; $m \approx (t_r + t_p)/t_b$.

Let's denote cumulative (normal) distribution of RoundCertificationRequest query messages with $\Phi_\sigma(x)$. Assume $m = 1$, let's find a minimum of $(2 - \Phi_\sigma(t_b - (t_r + t_p))(t_b)$ by adjusting t_b and `t1`.

Practical rule for optimizing the average latency (Fig. 9):
Start BFT Core round when completed quorum ratio is $t_b/\Delta \approx t_b/m(t_b-(t_r+t_p))$. Note that t_b measures time from the moment when UC generation starts, thus it is circular and a rolling average must be used Median can be found by ignoring overflow from previous round and then waiting for completion of half of the quorums. Time from round start to next median is roughly $t_r + t_p$. Adjust `t1` if t_b needs to be changed.

Even more practically:

1. Maintain a rolling average of t_b and Δ
2. Do not count the pending overflow from previous round
3. After counting half of expected quorums $(k/2)$ start timer for measuring Δ and re-start timer for measuring t_b
4. After counting kt_b/Δ unique quorums stop timer Δ and start UC generation.

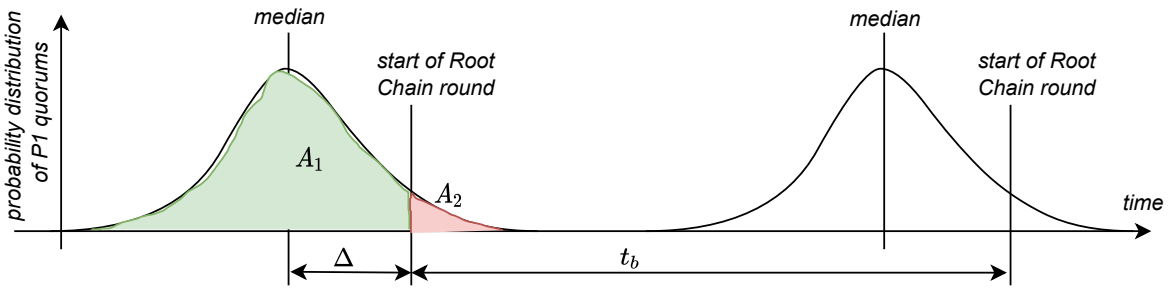

Figure 9. Optimal average latency is achieved when $\frac{A_1}{A_2} \approx \frac{t_b}{\Delta}$, where A represents the respective area (number of messages). $t_b - \Delta \approx t_r + t_p$.

If the distribution gets too large it makes sense to increase m, that is, have more than one core round per one average shard's round.

4.3.3 State

Please refer to Platform Configuration, State of the BFT Core, and Algorithm 12.

Configuration is provided by the Governance and other management processes. State must be persisted and synchronized (recoverable from other BFT Core validators).

Time-outs and timers:

- `t2` – BFT Core waits for `t2` time units before re-issuing a UC for a shard. This triggers a new round retrying certification of a block of inputs, with another leader. There is one timer instance per shard, referred as $\texttt{t2}_{\beta,\sigma}$ for the shard σ of the partition β.
- `t3` – Triggers Unicity Tree re-calculation and UC response generation (monolithic BFT Core only); Target Round Rate (t_b) – Distributed BFT Core specific parameter

Communication layer:

- Validator's secret key used to sign its messages
- related public key; known to other BFT Core validators via the Governance Records, and to other parties via the $\mathcal{T}$
- public keys of other BFT Core validators (used by the underlying communication layer)
- public keys of Shard Validators (usage captured by the opaque function VALID())
- communication addresses of other BFT Core validators
- $conn[][]$ – Connection contexts to return responses to validators which sent a RoundCertificationRequest request

4.3.4 Analysis

4.3.4.1 Safety

BFT Core enforces that each round is 'extended' by one round only. This excludes conflicting rounds (forks). BFT Core must not generate "equivocating" UCs (see Section 4.2.3.4 and Algorithm 7).

4.3.4.2 Liveness

Usual properties of a partially synchronous communication model apply.

4.3.4.3 Data Availability

If a validator issues a RoundCertificationRequest call then it must not lose the block proposal (request data) until the block is finalized and committed to persistent storage.

The mechanism of 'repeat UCs' presents a challenge: one block may receive multiple waves of extending attempts, initiated by subsequent 'repeat UCs'. There are following options:

1. Latest UC use is enforced strictly. If the BFT Core have issued a repeat UC, then all RoundCertificationRequest requests must be based on round number suggested by this response. Arriving and pending RoundCertificationRequest requests, referring to different round numbers, are dropped.
2. All RoundCertificationRequest requests extending the current state are considered; if there is a pending request from a validator then it is replaced by later request from this validator, with larger round number.
3. All RoundCertificationRequest requests extending the current state are considered, no matter the proposed round number, and included into pending request buffer; if

one proposed new state achieves majority then it wins; this state does not have to be the latest one.

On second and third option it is possible, that an aggregation layer node have issued multiple RoundCertificationRequest requests and eventually an older one gets certified. Thus, validators must retain all pending block proposals until one is committed.

These options provide somewhat better liveness of the protocol; in reality, if it is the case we can safely assume that time-out `t2` has too low value relative to system latencies. The rest of this specification assumes option 1.

Active aggregation layer nodes must retain their state until successfully—demonstrated by the ability to produce subsequent blocks—replaced by another set of validators after some epoch change. The long-term availability is provided by interested parties like RPC nodes, archive nodes, client middleware; these components must be able to replicate blocks with reasonable effort from the active validators.

4.3.5 Monolithic Implementation

This section (Algorithms 12, 13, 16) defines a monolithic, non-distributed BFT Core implementation. It serves distributed shards.

For a deployable implementation, please refer to Distributed BFT Core (Section 4.3.6); it provides the same functionality to shards without being a single point of failure, and can provide necessary level of decentralization.

Algorithm 12 Global Parameters and Variables

Configuration:
α: Network Instance identifier
$\mathcal{T}$: Unicity Trust Base
$C[]$: Partition Identifiers
$\mathcal{CD}[]$: Partition Description Records
State:
$n \leftarrow 0$: BFT Core's Round number
$e \leftarrow 0$: BFT Core's Epoch number
$r_{-} \leftarrow$ NULL: Previous round's Unicity Tree root hash
$\mathcal{SI}[][]$ - shard info of every $\sigma \in \mathcal{SH}_\beta$ of every $\beta \in C$
Variables:
$\forall\, \beta \in C, \sigma \in \mathcal{SH}_\beta$: $\mathcal{IRT}_\beta[\sigma] \leftarrow (\perp)$
χ_β: the shard tree for every $\beta \in C$ (Sec. 3.1.5)
υ: unicity tree (Sec. 2.7.3)
$changes[][] \leftarrow \perp$ Changes to IR, applied at the end of round, indexed by partition ID and shard number
$req[][][] \leftarrow \perp$: RoundCertificationRequest requests, indexed by partition ID, shard number and validator ID
$conn[][][] \leftarrow \perp$: Shard validator connections
Timers:
RESET_TIMER(`t3`)
$\forall\, \beta \in C,\ \forall\, \sigma \in \mathcal{SH}_\beta$: RESET_TIMER($\texttt{t2}_{\beta,\sigma}$)

4.3.5.1 Certification Request Processing

Certification Request (CR) processing starts with sanity checking of the message. Then, checking if a newer UC is available; if yes then it is returned immediately. This would initiate shard recovery if necessary, and start a new shard round.

If a request tries to extend an unknown state, then the latest UC is returned immediately.

Next, the request is retained in request buffer if it is the first valid message; if the message is a repeating message then processing stops.

Equal requests (comparing entire IR to make sure that other fields also match) from the same shard are counted. If a request achieves simple majority then respective IR gets added to the changes array waiting for certification, at position indexed by shard ID. If it is clear, that a shard can not converge to a majority agreement, then the slot in changes array is filled with IR from the certified IR array, with incremented round number and previous certified hash set the same as the certified hash. This produces a 'repeat UC' which, once delivered downstream, initiates a new shard consensus attempt.

See Algorithm 13.

4.3.5.2 Unicity Certificate Generation

Periodically, do the following:

1. In case a shard has not shown progress for a period `t2` since the last UC was delivered then respective slot is populated with the field content of previous certified IR array, with incremented round number. This produces a 'repeat UC'.
2. The pending changes in $changes$ array are applied to IR. If there are no new changes then the previous value is used.
3. Based on the array of Shard Input Records, build the Shard Trees.
4. Based on the roots of Shard Trees, build the Unicity Tree. Extract root hash value. Obtain wall clock time. Create the Unicity Certificate.
5. For every record in IR changes array, respond to the shard based on request context from the request buffer. Then, clean up req buffer, and reset respective `t2` timer.
6. Finally, reset $changes$, update the previous root hash used for linking[3], increment BFT Core round number and reset the timer `t3` which triggers the Unicity Certificate Generation (Algorithm 16).

4.3.6 Distributed Implementation

Distributed BFT Core is bisimilar to monolithic BFT Core in the sense that they provide the same service implementing the same business logic. Distributed BFT Core is Byzantine fault tolerant. It uses the SMR (State Machine Replication) concept.

The messaging between a shard and the BFT Core is illustrated by Figure 10, where a shard with two validators v_1 and v_2 requests a UC. BFT Core is also depicted with two validators, the next leader after reaching a quorum of shard requests is v_l^r.

[3]Note that linear hash-linking using r_{-} illustrates the idea that some sort of cryptographic linking is present; actual mechanism depends on the BFT Core implementation and used linking scheme.

Algorithm 13 CR Message Handling.

upon message <CR | $m = (\alpha, \beta, \sigma, \nu, IR, \ell_B, \ell_S; s; [\pi_{CP}]); context$> **do**
  ensure(VALID(m))
  ensure($m.\alpha = \alpha$) ▷ Right network id
  ensure($\beta \in C$) ▷ Valid part. id
  ensure($\sigma \in \mathcal{CD}[\beta].\mathcal{SH}$) ▷ Valid shard id
  ensure($\nu \in \mathcal{SI}[\beta, \sigma].\mathcal{V}$) ▷ Authorized validator
  $conn[\beta][\sigma][\nu] \leftarrow context$ ▷ Network connection for returning messages
  if $IR.n \neq \mathcal{SI}[\beta, \sigma].n + 1$ ▷ Round is behind/ahead
    $\vee$ $IR.e \neq \mathcal{SI}[\beta, \sigma].e$ ▷ Epoch can be incremented by RP only
    $\vee$ $IR.h' \neq \mathcal{SI}[\beta, \sigma].h$ ▷ Extending of unknown state
    $\vee$ $\neg \mathcal{CD}[\beta].\gamma_{CP}(IR.h', IR.h, \ell_B, \pi_{CP})$ ▷ Consistency proof verification
    $\vee$ $IR.t \neq \mathcal{SI}[\beta, \sigma].UC_{-}.C^{\mathrm{r}}.t$ **then** ▷ Time not set as expected
    SEND_CRES($context; \beta, \sigma, \mathcal{SI}[\beta, \sigma].UC_{-}$, GET_TE($\mathcal{SI}[\beta, \sigma]$))
    return
  end if
  if $req[\beta, \sigma][\nu]$ **then** ▷ Reject duplicate request
    return
  end if
  ensure(TX-SYSTEM-SPECIFIC-CHECKS($\mathcal{CD}[\beta], IR$))
  $req[\beta, \sigma][\nu] \leftarrow IR$ ▷ Add the new message
  $c \leftarrow \max_{r \in req[\beta][\sigma]} \sum_{p \in req[\beta][\sigma]} [r = p \wedge r.h \neq \perp]$ ▷ Number of the max. matching votes,
    ▷ Ignoring $\perp$, the "negative vote"
  $k \leftarrow |\mathcal{SI}[\beta, \sigma].\mathcal{V}|$ ▷ Number of validators
  if $c > k/2 \wedge \neg changes[\beta, \sigma]$ **then** ▷ Consensus
    $changes[\beta, \sigma] \leftarrow$ CERTIFY($IR, \mathcal{SI}[\beta, \sigma]$)
  else if $|req[\beta, \sigma]| - c \geq k/2$ **then** ▷ Consensus not possible
    ▷ If max.match + |yet missing votes| < threshold
    $changes[\beta, \sigma] \leftarrow$ REPEAT_CERT($\mathcal{SI}[\beta, \sigma]$) ▷ Produce 'repeat UC'
  end if
end upon

4.3.6.1 Summary of Execution

The summary follows the processing flow of a Certification Request by a BFT Core validator. The flow is illustrated in Fig. 11; loosely moving counter-clockwise.

Peer Node Selection In order to use the distributed BFT Core, aggregation layer nodes must choose an appropriate subset of BFT Core validators to communicate with. The set must be shared by all aggregation shard nodes during one shard round; on receiving a "repeat UC", the validators must communicate with a different subset. The number of validators in this set is a tuning parameter: balances between availability and overhead. 2-3 validators is a good starting point. This number (in the sense of delivering messages in parallel) does not have security implications, as produced quorum sets retain aggregation layer nodes' signatures which can be verified independently. UC responses must be checked for equivocation.

If an aggregation layer node haven't received a UC-s from chosen BFT Core validators

Algorithm 14 Certification helper functions

```
function CERTIFY(ir : 𝕀ℝ, si : 𝕊𝕀)
    si ← UPDATE_SHARD_INFO(ir, si)          ▷ Update caller's Shard Information record si
                ▷ According to Platform Specification, functional description of BFT Core
    te : 𝕋𝔼 ← GET_TE(si)
    return (ir, te)
end function
function REPEAT_CERT(si : 𝕊𝕀)
    si.n ← si.n + 1                         ▷ Changing the parent's data structure as well
    si.v_l ← LEADERFUNC(si.UC_, si.𝒱)
    rir : 𝕀ℝ ← (si.UC_.IR)                  ▷ Repeat the previous UC's IR
    te : 𝕋𝔼 ← GET_TE(si)
    return (rir, te)
end function
function GET_TE(si : 𝕊𝕀)
    return (si.n + 1, si.e, si.v_l, h(si.SR_, si.SR), h(si.VF_, si.VF))
end function
```

Algorithm 15 "SubscriptionMsg" subscribes to UC feed

```
upon message <SubscriptionMsg | (α, β, σ, v; s); context> do
    ensure(VALID((α, β, σ, v; s)))
    conn[β, σ][v] ← context     ▷ Network connection for returning subsequent messages
    return GET_UC(β, σ)
end upon
```

within $2 \times \texttt{t2}$ then it sends SubscriptionMsg requests to random other BFT Core validators. This ensures eventual synchronization if the firstly chosen validators happen to be faulty.

CR Validation No difference from Monolithic Implementation, please refer to Algorithm 13 for details. If available, or on invalid request, UC is returned immediately by the same BFT Core validator.

Shard Quorum Check Shard Quorum Check is the same as on Monolithic case (Alg. 13). If a quorum is achieved or considered impossible, then a message is assembled (IRChangeReq), which includes all CR messages, and forwarded to the next BFT Core leader, using the Atomic Broadcast submodule.

IR Change Request Validation The request must be validated analogously to the Algorithm 9.

Proposal Generation Leader collects all unique IR change requests and assembles the block proposal, which includes changed IR-s and a justification for each IR change request. Next, if a particular IR has not been changed during t2 timeout for this shard then the leader initiates "repeat UC" generation by including a specific record into the block proposal. There is no explicit justification, followers can validate timeouts based on their own timers.

In a proposal there can be up to one change request per shard.

Algorithm 16 Unicity Certificate Generation

on event t3 **do** ▷ Simplified, see section "Timing"
  for all $\beta \in C$ **do** ▷ Process partition timeouts
    for all $\sigma \in \mathcal{SH}_\beta$ **do**
      if $\neg changes[\beta, \sigma] \wedge$ EXPIRED_TIMER($t2_{\beta,\sigma}$) **then**
        $changes[\beta, \sigma] \leftarrow$ REPEAT_CERT($\mathcal{SI}[\beta, \sigma]$) ▷ Produce 'repeat UC'
      end if
    end for
  end for
  for all $(\beta, \sigma, ir, te) \in changes$ **do** ▷ Apply changes
    $\mathcal{IRT}_\beta[\sigma] \leftarrow (ir, h(te))$
  end for
  for all $\beta \in C$ **do**
    $\chi_\beta \leftarrow$ CreateShardTree($\mathcal{SH}_\beta, \mathcal{IRT}_\beta$) ▷ Sec. 3.1.5.1
    $\mathcal{IH}[\beta] \leftarrow \chi_\beta(\sqcup)$
  end for
  $r \leftarrow$ CreateUnicityTree($C, \mathcal{CD}, \mathcal{IH}$) ▷ Sec. 3.1.6.1
  $t \leftarrow$ GetUTCDateTime()
  $C^{\mathsf{r}} \leftarrow$ CreateUnicitySeal(α, n, e, t, r_-, r; sk_r)
  for all $(\beta, \sigma, ir, te) \in changes$ **do**
    $req[\beta][\sigma] \leftarrow []$
    $C^{\mathsf{shard}} \leftarrow$ CreateShardTreeCert(σ, χ_β)
    $C^{\mathsf{uni}} \leftarrow$ CreateUnicityTreeCert($\beta, C, \mathcal{CD}, \mathcal{IH}$)
    $uc \leftarrow (\mathcal{IRT}_\beta[\sigma].IR, \mathcal{IRT}_\beta[\sigma].h_t, C^{\mathsf{shard}}, C^{\mathsf{uni}}, C^{\mathsf{r}})$
    $\mathcal{SI}[\beta, \sigma].UC_- \leftarrow uc$
    for all $connection \in conn[\beta][\sigma]$ **do** ▷ To all validators of the shard, in parallel
      SEND_CRES($connection; \alpha, \beta, \sigma, uc, te$) ▷ Subscriptions expire – see protocol desc.
    end for
    RESET_TIMER($t2_{\beta,\sigma}$)
  end for
  $changes \leftarrow \{\}$
  $r_- \leftarrow r$
  $n \leftarrow n + 1$
  RESET_TIMER(t3)
  ▷ Note that IRT retains its current value for the next round
end on

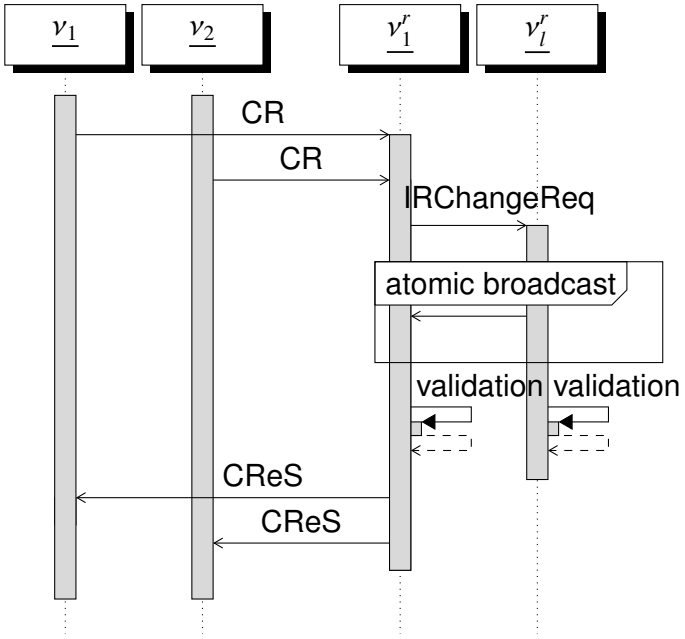

Figure 10. Message flow (simplified)

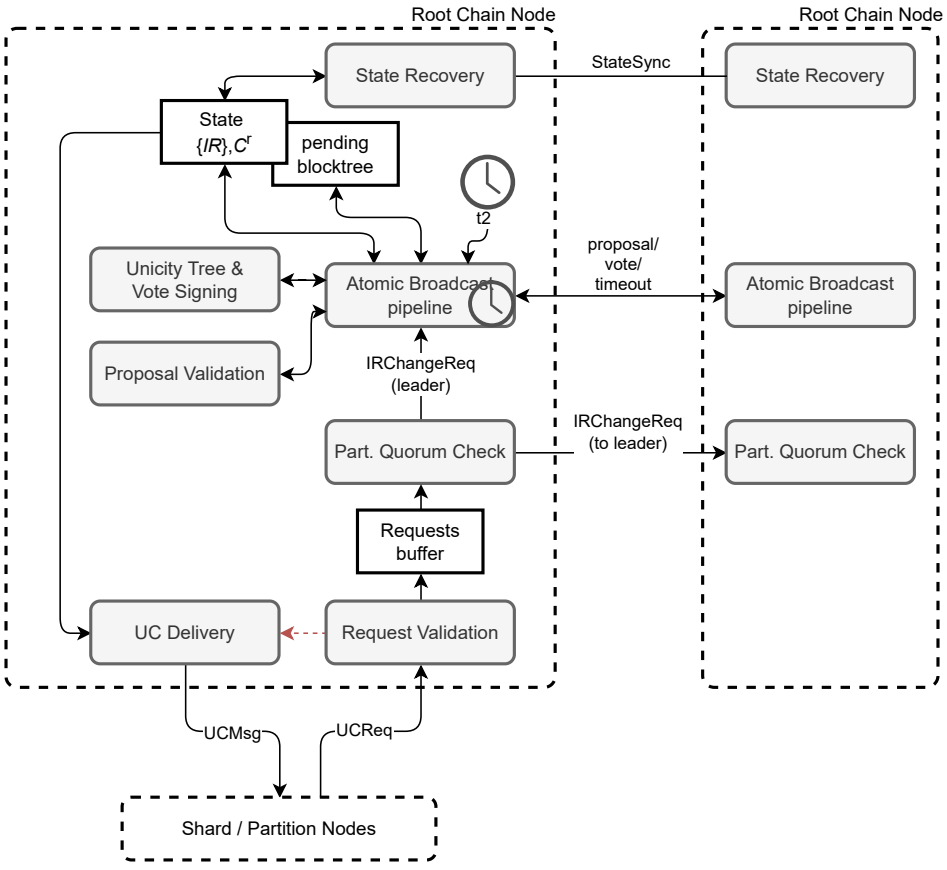

Figure 11. Data-flow of a Distributed BFT Core Validator

Due to the pipe-lined finality, a proposal should not include IR change requests for slots with a valid IR change request in immediately preceding round.

Finally, the leader signs the proposal and broadcasts it to other BFT Core validators.

Proposal Validation On receiving a proposal, validator validates it. There are consensus-specific checks. Every IR change request is validated based on its kind; base rules are presented by Algorithm 9. Summary of the cases:

Quorum achieved The justification must prove the achievement of consensus by an aggregation shard. More than half of the shard nodes must have coherent votes.

Quorum not possible The justification proves that there are enough conflicting votes to render the consensus impossible.

t2 timeout The message states, that its timer have reached the timeout; all validators can confirm the timeout based on their own clocks, allowing a little drift.

After validating the proposal and checking the Voting Rule, the validator signs its vote data structure and sends it to the next leader in pipeline.

On encountering unexpected state hash the recovery process is initiated (see Section 4.3.6.2).

State Signing On assembling a Quorum Certificate (QC) with enough votes and verifying the Commit Rule the leader modifies state: for every newly certified IR element it updates its last UC array.

UC Generation If a leader updates the last UC array element then it returns CReS responses to all pending aggregation shard nodes.

QC is included to HotStuff message pipeline, so it is broadcast to other validators (together with the new proposal produced by this validator). On seeing new QC and knowing recertified IRs, other validators update their last UC arrays (the state).

4.3.6.2 Proposal

Proposal is a signed set of IR change requests, supplemented with proofs–the necessary number of signed partition nodes' messages (*justification*). There are following options:

- Change requests with justifications.
- Repeat Certification Request where consensus is considered impossible.
- Repeat Certification Request on `t2` timeout of a shard.

State Synchronization A BFT Core validator must have up-to-date vector of Input Records of all shards. There is no persistent blockchain. Returned Unicity Certificates, certifying IRs, are possibly persisted by the underlying layers.

The state includes necessary meta-data like the round number. This is provided by including the Unicity Seal, which also authenticates the IR vector.

State may include atomic broadcast module specific data, e.g. uncommitted round information.

4.3.6.3 Atomic Broadcast Primitive

The Atomic Broadcast primitive is instantiated using an adaptation of HotStuff consensus protocol. The adaptation is optimized towards better latency on good conditions. Therefore, a "2-chain commit rule" is used. Changes and tweaks wrt. the original HotStuff paper are:

- "2-chain commit rule"
- Timeout Certificates for view change. This induces quadratic communication complexity on faulty leader, but enables the 2-chain rule instead of 3-chain.
- QC component votes go to the next leader directly and the next leader assembles QC.
- Use of aggregated signatures (instead of threshold signatures)

More formally, critical elements of the operation of a HotStuff-derived algorithm are specified by the following rules.

Let n denote round number, B – block of inputs, QC – Quorum Certificate, TC – Timeout Certificate.

Rule 1. Voting Rule
$B.n >$ last vote round
$B.n = B.QC.n + 1 \vee (B.n = TC.n + 1 \wedge B.QC.n \geq max($TC.tmo_high_qc_round$))$

Rule 2. Timeout Rule
$n \geq$ last vote round
$(n = QC.n + 1 \vee n = TC.n + 1) \wedge QC.n \geq$ 1-chain round

Rule 3. Commit Rule
It is safe to commit block B if there exist a sequential 2-chain $B \leftarrow QC \leftarrow B' \leftarrow QC$ such that $B'.n = B.n + 1$.

Please refer to the following papers for more details:

1. HotStuff: BFT Consensus in the Lens of Blockchain
2. DiemBFT v4: State Machine Replication in the Diem Blockchain

The concepts used in this specification map to the concepts used in the HotStuff and DiemBFT papers as follows:

State: Vector of Input Records
State Authenticator: State is identified by the root hash of Unicity Tree.
Block Proposal: List of changes to Input Records, with justifications; or a proposal to switch epochs.
Block: There are no (explicit) blocks. The set of Input Records can be seen as the cumulative state after applying all previous (virtual) blocks. Unicity Certificates are propagated downstream to shards where they could be saved as part of shard blocks. Validator implementation is encouraged to produce an audit log with all the block proposal payloads.
Blockchain: There is no such thing as the BFT Core Blockchain. However, Unicity Trust Base is somewhat blockchain-like: it gets a new entry added once per BFT Core epoch, and similarly to block headers, it can be interpreted as the Root of Trust.

Round Pipeline Committing the payload happens across many rounds due to the pipelined nature of HotStuff. In consecutive rounds, the flow is like this:

1. vector of IR change requests (payload of the proposal message)
2. updated IR-s (node block-tree) and Unicity Tree root (exec_state_id of a vote message)
3. committed Unicity Tree Root (commit_state_id of a vote message).

The output is commit_state_id from a Quorum Certificate which was formed by combining vote messages. QC is used to produce the Unicity Seal.

Pacemaker Pacemaker is a module responsible for advancing rounds, thereby providing liveness. Pacemaker sees votes from other validators and processes local time-out event.

Pacemaker either advances rounds on seeing a QC from the leader or on no progress, on seeing a TC. On local timeout or seeing $f + 1$ timeout messages a validator broadcasts signed TimeoutMsg message. TC is built from $2f + 1$ distinct TimeoutMsg messages. All messages must apply to currently known HighestQC.

The BFT Core should not tick faster than configured Target Round Rate. In order to throttle the speed, there is deterministic wait performed by leaders at every round.

The wait must be reasonably small to not trigger TimeoutMsg messages from other validators.

Leader Election Initially, a round-robin selection algorithm is used. In the roadmap there is stake weighted, unpredictable leader schedule.

Reputation is taken into account while producing per-epoch validator set assignments, for lowering the chance of inactive or unstable validator becoming a leader. One validator should not be the leader in two consecutive rounds.

Example: Take all validators. Remove one or more (fixed number) of the previous leaders. Remove all validators who did not participate in creation of the latest QC or TC. Pick one pseudo-randomly, and deterministically across all the validators, from the remainder.

4.4 Dynamic System

This section describes mechanisms making Unicity a dynamic system, by re-configuring partitions, shards and the BFT Core on the fly during execution. This introduces *epochs* – time periods where the configuration is stable. Configuration changes are executed at the switch of epoch. Partitions, shards and the BFT Core are not synchronized with each other, they have independent rounds and epochs.

Also, the changing nature of some global data structures is discussed below.

4.4.1 Configuration Changes

Table 1 provides a non-exhaustive list of configuration changes and the event synchronizing the change.

Table 1. Synchronizing configuration changes.

No	Change	RP epoch	Shard epoch
1.	Adding, removing aggregation layer nodes		✓
2.	Adding, removing BFT Core validators	✓	
3.	Communication address change of validators		
4.	Splitting (sharding) shards		✓
5.	Adding, removing partitions		✓
6.	RP's globally visible data structure changes (also increments versions of $\mathcal{T}$, C^{r}, see Sec 4.4.6.1)	✓	
7.	RP's operation rules	✓	
8.	Shard's data structure version changes resulting in changes in log content		✓
9.	Shard's data structure version changes resulting in changes in proofs		✓
10.	Changes in validation algorithms (ledger rules)		✓

Adding and removing of validators may incur changes in the respective quorum sizes. Communication layer changes are entirely handled by the communication layer. By "adding or removing a validator" we mean a change in actively participating validator identifier sets.

4.4.2 BFT Core Epoch Change

In order to facilitate a dynamic, responsive BFT Core, it is necessary to adjust its parameters on the fly. In particular, it is necessary to add new validators and retire some existing ones to maintain a healthy validator set, due to changing requirements and operating conditions.

The configuration can be changed once per epoch. The source information comes from an Governance Process whose output is change records called Validator Assignment Records (Table 4).

Any BFT Core protocol leader can initiate an epoch change, given it has received the change record. The procedure works as follows:

1. The leader produces a proposal where the usual payload is replaced by the Change Request justifying the increment of Epoch (Table 2);
2. Validators who approve the epoch change continue with execution flow and do not include usual payload until the epoch change proposal gets committed.
3. Next round after committing an epoch change is the first round of this epoch: Epoch in BlockData is incremented; and execution continues by the updated set of validators.

Fields 1, 2, 3 are copied from the Validator Assignment Records. The change record is a "justification": it is used as helper data for validators, but not included into the finalized record and resulting entry in the Unicity Trust Base. An implementation may choose to rely on alternative channels for distributing change records.

BFT Core's epoch change updates the Unicity Trust Base. On record-oriented Unicity Trust Base, the new record becomes part of Root Validator's state (see section 4.4.6.6); and the

Table 2. BFT Core Epoch Change Request.

No	Field	Notation	Type
1.	Epoch number	e	$\mathbb{N}_{64}$
2.	Validator identifiers and stakes	$\{v, b_v\}_e$	$\{(\mathbb{OCT}^*, \mathbb{N}_{64})\}$
3.	Quorum size (Voting power)	k_e	$\mathbb{N}_{64}$
4.	Hash of state summary	r	$\mathbb{H}$
5.	Hash of Change Record	h_{cr}	$\mathbb{H}$
6.	Hash of the previous record	h_{e-1}	$\mathbb{H}$
7.	(attached) Change Record		(Table 4)

new record gets propagated to Shard validators together with the next Unicity Certificate, as an extra field of the UCResp message.

Rule. BFT Core Epoch Change
For every proof, the Epoch Number in its Unicity Certificate's Unicity Seal must point to the entry in Unicity Trust Base which can be used for this proof's verification.

Validators must not execute invalid change records and approve proposals with invalid Change Requests. Instead, the latest valid change record must be used, whenever available; skipping over of some if necessary.

4.4.3 Shard Epoch Change

Shard Epoch Change is triggered by the BFT Core, by incrementing the Epoch number field in Technical Record, returned with a UC.

The next shard round after finalizing a round with UC with incremented Epoch value is processed according to the configuration of the next epoch. Leader and validators are selected according to next epoch configuration.

Rule. Shard Epoch Change
UC with incremented Epoch number in IR can be extended by a quorum of validators of the next epoch.

4.4.4 Controlling Shard Epochs

BFT Core triggers shard epoch changes. This happens in the following steps:

1. BFT Core validators obtain the Change Record for the next epoch of a shard.
2. If a BFT Core Leader includes an Input Record Change Request of a shard into a proposal and there is a pending epoch change of this shard, then the Change Record is included to the Request.
3. Presence of Change Record is the signal that Epoch should be incremented. Included Change Record decision is validated, and if valid then epoch number in Technical Record (TE) is incremented. All other IR, TE changes are validated and applied as well.
4. IR, TE-s get certified.

5. New CReS message is returned to the shard.
6. Shard's configuration is updated: quorum size (voting power needed for consensus), list of validators. This changes BFT Core's validation rules: the next Certification Request must be presented by a valid quorum of next epoch's aggregation shard nodes.

Rule. Shard Epoch Change Control
If a shard's state is certified by a UC with incremented Epoch number in certified TE, then the next shard round's requests get validated by the new epoch's configuration.

For the clarity of presentation, epoch handling is not present in the provided pseudocode. It supports the lower layer functionality of dynamic configuration changes.

4.4.5 Validator's life cycle

This section describes epoch changes from the viewpoint of an individual node, in particular, how a node joins and leaves shards.

Node accepts configuration updates through the Config API. The message is a set of change records, one per epoch. The records should cover the range from node's view of the current epoch to the epoch where node's status changes. The possible status changes (in respect to the current factual state of the node) are covered below.

It is assumed that the node have made itself already findable via the node discovery service.

4.4.5.1 Shard Validator, joining

Initially, a shard node is not an active validator. It receives a configuration message which indicates, that starting from epoch $e_{\beta,\sigma}$, this node is a validator in shard σ of partition β. Following procedure is executed:

1. Reset
   Clean up state and storage, unless already synchronized to the required shard
2. Start accepting proposals
   On incoming proposal message:
   - Extract the UC and update the last known UC if newer
3. Start accepting client requests
   On incoming client transaction:
   - If it is possible to determine the leader then forward client requests to the leader.
   - Reject otherwise.
4. Obtain the latest UC for the shard (β, σ) from the BFT Core.
5. Launch recovery if behind
6. Subscribe to log feed of the shard (β, σ).
   On arrival of a new block of inputs:
   - Verify the inputs, apply the inputs to state and store in ledger
   - If block's UC indicates that the expected epoch arrives:
     - Starting from the next round, this node is full validator

– Execution and message handling continues as a full validator

If the input block log streaming protocol is not available the procedure is as follows:

1. Reset
   Clean up state and storage, unless already synchronized to the required shard
2. Start accepting proposals
   On incoming proposal message:
   - Extract the UC and update the last known UC if newer
3. Start accepting client requests
   On incoming client transaction:
   - If it is possible to determine the leader then forward client requests to the leader.
   - Reject otherwise.
4. **loop:**
   - Obtain the latest UC from the BFT Core
   - Launch recovery if the node is behind
   - If the UC indicates that the epoch where node is a full validator have started:
     – this node is now a full validator,
     – exit loop.

The state graph is depicted at Figure 12.

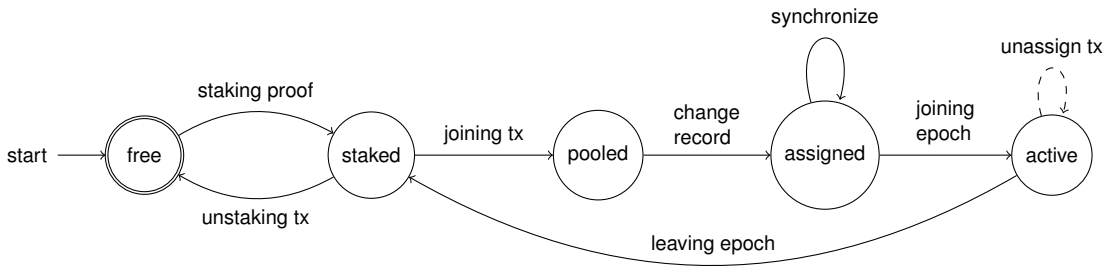

Figure 12. Node state graph (node's view)

4.4.5.2 Shard Node, leaving

If the node is an active validator and it receives indication that from an epoch e this node is not any more a member of shard (β, σ), then the following procedure is executed:

On receiving a UC indicating the expected epoch update:

1. Starting from the next round, this node is not a validator.
2. Stop accepting proposals.
3. Stop accepting client request messages.
4. OK to reset the state.

5. Continue serving the log replication protocol; it is mandatory to retain the storage until the next quorum have committed at least one round.
6. OK to reset the log.

The node (more specifically the node operator) can initiate the leaving process by sending an “unassign transaction” to the Governance Body. Processing may take arbitrary time, and the unassignment may happen for other reasons as well. Validators are encouraged to not drop off without executing the unassignment process.

Now, the node may submit a request to the Governance Body to be assigned again as an active node, possibly at another shard. The node is encouraged to keep its identity.

Rule. Every Shard Node must ensure the availability of generated ledger until the next validator set takes over.

4.4.5.3 Shard Node, ambiguous records

If there is a double assignment before leaving then the validator must continue at the current shard. When eventually the validator gets dismissed, it must join according to the latest valid change record.

4.4.5.4 BFT Core Node, joining

On receiving a change record:

1. Obtain the current BFT Core configuration via the GetUnicityTrustBase RPC call
2. Validate the updated Unicity Trust Base
3. Make itself findable via the node discovery service
4. Start listening to BFT Core proposal messages.
   On receiving a proposal where ProposalMsg.BlockData.Epoch is incremented and the epoch includes the node:
   - Use the BFT Core recovery protocol (GetStateMsg / StateMsg) to synchronize the state,
   - Process the proposal,
   - Start processing shard Certification Requests,
   - Continue as full BFT Core validator.

4.4.5.5 BFT Core Node, leaving

A BFT Core node may leave the active validator set after performing a successful epoch change, where this node is not a member of the new validator set anymore. The node is responsible for successful hand-over: by making its state available to the new joining nodes briefly after the beginning of the next epoch, and offering the Unicity Trust Base distribution service without limitations to all current and near-future validators.

A BFT Core node (more specifically, the operator of the node) can initiate the leaving process by sending a specific “unassign transaction” to the Governance Body. Processing

may take arbitrary time, and the unassignment may happen for other reasons as well. BFT Core validators are strongly encouraged to not drop off without executing the unassignment process.

Rule. BFT Core Node must keep running until the next epoch's Unicity Trust Base entry gets committed.

4.4.6 Dynamic Data Structures

4.4.6.1 Versioning

The content of Unicity Seal and Unicity Trust Base, globally used data structures across the Unicity Platform, depend on Distributed BFT Core implementation details. These data structures must be versioned to accommodate necessary iterative changes while the BFT Core implementation roadmap is being executed. That is, $\mathcal{T} = (v, \cdot)$ and $C^{\mathsf{r}} = (v, \cdot)$, where v is the version number and the rest depends on the version. In the following sections, we assume that a section shares the same version number and it is omitted for brevity.

Accordingly, the function VerifyUnicitySeal must be able to verify a recent subset of Unicity Seal versions, based on an authentic copy of the up-to-date Unicity Trust Base. This can be imagined as a wrapper, where the version number of input chooses the right implementation.

4.4.6.2 Evolving

Unicity Trust Base itself evolves (e.g., new records are added, while format/version stays the same) when the BFT Core validator set changes. Every evolved copy of Unicity Trust Base is cryptographically verifiable based on an older authentic copy of the Unicity Trust Base.

We denote the initial, authentic[4] Unicity Trust Base as $\mathcal{T}_{\mathsf{base}}$ and updated Unicity Trust Base as $\mathcal{T}$, and for each version of data structures define the function

$$\mathsf{VerifyUnicityTrustBase}_{\mathcal{T}_{\mathsf{base}}}(\mathcal{T}) \ .$$

For every supported version of proofs an implementation of VerifyUnicitySeal and the relevant Trust Base must be provided. Only the latest version of Unicity Trust Base can evolve.

At the launch, the system is bootstrapped to a genesis state where the content of Unicity Trust Base, together with Genesis Blocks, are created via some off-chain social consensus process. The relevant data structures are the same, while references to previous states and signatures created by previous states are hard-coded to zero values.

4.4.6.3 Monolithic, Static BFT Core

We start with one BFT Core validator, which does not change (and can not change its keys).

[4] Authenticity is guaranteed by e.g., off-band verification of the Genesis Block and embedding the newest possible Unicity Trust Base into the verifier code during release management process, analogously to *certificate pinning*.

- $\mathcal{T} = (\alpha, \mathsf{pk})$,
  where α is the system identifier and pk is the public key of the BFT Core.
- $C^{\mathsf{r}} = (\alpha, n_r, t_r, r_-, r; s)$,
  as defined in the Platform Specification; s is a digital signature created using BFT Core's secret key.

function VerifyUnicitySeal($r, C^{\mathsf{r}}, \mathcal{T}$)
  return ($\mathcal{T}.\alpha = C^{\mathsf{r}}.\alpha \wedge r = C^{\mathsf{r}}.r \wedge \mathsf{Ver}_{\mathcal{T}.\mathsf{pk}}(C^{\mathsf{r}}, C^{\mathsf{r}}.s)$) ▹ Calculated over all fields except signature
end function

function VerifyUnicityTrustBase($\mathcal{T}_{\mathsf{base}}, \mathcal{T}$)
  return ($\mathcal{T}_{\mathsf{base}} = \mathcal{T}$) ▹ No changes are allowed
end function

4.4.6.4 Monolithic, Dynamic BFT Core

In the case of dynamic BFT Core the validator(s) can change at epoch boundaries. The identifier (public key) of new validator is signed by the current one, and this signed record is appended to the Unicity Trust Base.

- $T_e = (\alpha, e, \mathsf{pk}_e; s)$
  is Unicity Trust Base Record, where $s = \mathsf{Sig}_{\mathsf{sk}_{e-1}}(T_e)$ is a cryptographic signature over verification record of epoch e (calculated over all fields except signature), signed by the previous epoch's secret key of the BFT Core.
- $\mathcal{T} = (T_j, T_{j+1}, \ldots, T_k)$,
  where j is the first epoch and k is the latest epoch covered. In the pseudocode below, we use notation $\mathcal{T}[j] := (T \in \mathcal{T} : \ T.e = j)$, that is, the element pointing to the epoch number j. System instance identifier of every epoch is invariant: $\forall\ i, j \colon T_i.\alpha = T_j.\alpha$.
- $C^{\mathsf{r}} = (\alpha, n_r, t_r, r_-, r; (s, e))$
  as defined in the Platform Specification; s is a cryptographic signature created using BFT Core's secret key of the epoch e.

function VerifyUnicitySeal($r, C^{\mathsf{r}}, \mathcal{T}$)
  if ($\mathcal{T}.\alpha \neq C^{\mathsf{r}}.\alpha \vee r \neq C^{\mathsf{r}}.r$ **then**
    return 0
  end if
  if $\mathcal{T}[C^{\mathsf{r}}.e] = \perp$ **then**
    return error ▹ No record in trust base for the epoch
  end if
  return ($\mathsf{Ver}_{T.\mathsf{pk}}(C^{\mathsf{r}}, C^{\mathsf{r}}.s) = 1$) ▹ Calculated over all fields except signature
end function

Note that the caller must ensure that a relevant record is present in Unicity Trust Base, that is, $\mathcal{T}[C^{\mathsf{r}}.e] \neq \perp$. Obtaining a fresh Unicity Trust Base is covered by the Chapter Unicity Anterior.

function VerifyUnicityTrustBase($\mathcal{T}_{\mathsf{base}}, \mathcal{T}$)
  $bmax = \max_{T \in \mathcal{T}_{\mathsf{base}}} T.e$ ▹ Last epoch in trust base
  $tmin = \min_{T \in \mathcal{T}} T.e$ ▹ First epoch number to be verified
  if $tmin > bmax + 1$ **then**

```
        return error                                          ▷ Not a continuous chain
    end if
    for T ∈ 𝒯 do
        if 𝒯_base[bmax].α ≠ T.α then
            return error                                      ▷ Non-invariant system id
        end if
    end for
    for j ∈ {tmin .. bmax + 1} do                             ▷ Verify based on available trust base
        if Ver_{𝒯_base[j−1].pk}(𝒯[j], 𝒯[j].s) = 0 then
            return 0
        end if
    end for
    for j ∈ {tmin + 1 .. max_{T∈𝒯} T.e} do                    ▷ Verify consistency of new
        if Ver_{𝒯[j−1].pk}(𝒯[j], 𝒯[j].s) = 0 then
            return 0
        end if
    end for
    return 1
end function
```

For efficiency, the user must cache the verification results. For example: 1) at the startup, the bundled trust base is checked for consistency, and 2) each time an evolved trust base is encountered, a) it is checked for equivocation, b) if the trust base is newer than the base and verified using the function VerifyTrustBase, then the base trust base is updated with new records from the new trust base (or substituted with the latest version if the implementation is accumulator-like).

This version is illustrative and not for implementation.

4.4.6.5 Distributed, Static BFT Core

Unicity Trust Base is a map of BFT Core validator identifiers to validator public keys, and a number q which specifies the required quorum size, i.e.,

- $\mathcal{T} = (\alpha, \{(i, \mathsf{pk}_i) \mid i \leftarrow 1 \ldots v_r\}, q)$ where $q \geq v_r - f$,
- $C^{\mathsf{r}} = (\alpha, n_r, t_r, r_-, r; s)$ where $s = \{(i, s_i) \mid i \in (1 \,..\, v_r)\}$ and $|s| \geq q$.

Unicity Trust Base does not change as the BFT Core configuration is static.

```
function VerifyUnicitySeal(r, C^r, 𝒯)
    if 𝒯.α ≠ C^r.α ∨ r ≠ C^r.r then
        return 0
    end if
    if ¬(∀ m, n ∈ 1 .. |C^r.s| : C^r.s[m].i ≠ C^r.s[n].i) then
        return 0                                              ▷ Duplicate signers
    end if
    if |C^r.s| ≤ 𝒯.q then
        return 0                                              ▷ No quorum
    end if
```

```
    for (i, s) ∈ C^r.s do
        if (Ver_{T.pk_i}(C^r, s) = 0) then        ▷ Calculated over all fields except signature
            return 0                               ▷ Invalid signature
        end if
    end for
    return 1                                       ▷ Success
end function
```

4.4.6.6 Distributed, Dynamic BFT Core

The Unicity Trust Base Record is defined by Table 3.

Table 3. Unicity Trust Base Record of Dynamic Distributed BFT Core.

No	Field	Notation	Type
1.	Network instance identifier	α	$\mathbb{A}$ (invariant)
2.	Epoch number	e	$\mathbb{N}_{64}$
3.	Epoch starting round	n_e	$\mathbb{N}_{64}$
4.	Validator identifiers and stakes	$\{v, b_v\}_e$	$\{(\mathbb{OCT}^*, \mathbb{N}_{64})\}$
5.	Quorum size (voting power)	k_e	$\mathbb{N}_{64}$
6.	Hash of state summary	r	$\mathbb{H}$
7.	Hash of related change record	h_{cr}	$\mathbb{H}$
8.	Hash of previous record	h_{e-1}	$\mathbb{H}$
9.	Signature of previous epoch validators	s_{e-1}	*version-dependent*

Fields 1, 2, 4 and 5 are copied from the referenced Governance Record. Initially, the stakes are fixed to 1. When appropriate orchestrating processes implementing (delegated) Proof of Stake mechanisms are in place, the stakes reflect the epoch's locked stake amounts of each particular validator.

Unicity Trust Base is a chain of records defined by Table 3.

Unicity Seal is a record signed by the validator set of the respective epoch as defined in 4.4.6.4, with an additional requirement of using the required multi-party signature scheme.

The verification functions are as defined in 4.4.6.4, where the signatures are interpreted in broader sense as multi-party signatures, created by respective quorums of validators.

4.4.6.7 Signature Aggregation

This is an optimization of Distributed BFT Core data structures, reducing the sizes of produced proofs and the trust base. It is based on a cryptographic primitive implementing the "non-interactive, accountable subgroup multi-signature", allowing identification of all parties whose (part-) signatures are aggregated into a final, aggregate signature. On the case of aggregatable signature schemes, the m-of-n aggregation of public keys is non-trivial though[5], thus, it may be implemented further down the roadmap.

Unicity Trust Base is a tuple of aggregate public key and a numeric parameter (q) specifying the necessary quorum size. Signature on Unicity Seal is an aggregate signature, produced

[5] See e.g., `https://eprint.iacr.org/2018/483`

by combining at least q partial signatures, and a bit-field identifying the signers. Partial signatures are created by individual BFT Core validators using their private keys.

A standardization attempt of the closest appropriate signature scheme is available from IETF — https://datatracker.ietf.org/doc/draft-irtf-cfrg-bls-signature/.

Specifically, threshold signature schemes are avoided because of 1) accountability requirement 2) complicated and security-critical key setup, and 3) missing support of non-equal voting powers.

4.5 BFT Core Data Structures (illustrative)

Below is an informal illustration on how Unicity data structures integrate to HotStuff consensus primitive; in ABNF format[6]. The structures document the distributed, dynamic BFT Core.

```
; Unicity Protocol Data Structures (ABNF)

; ===========================================================================
; 1.  BFT CORE (shared data structures)

  UC = IR TechRecordHash ShardTreeCertificate UnicityTreeCertificate UnicitySeal ; UC = (IR, h_t, C^shard, C^uni, C^r)
  IR = RoundNumber Epoch PreviousHash Hash Time ; IR = (n, e, h', h, t)
  ShardTreeCertificate = ShardIdentifier *SiblingHash ; C^shard = (σ; h^s_1, ..., h^s_|σ|)
  UnicityTreeCertificate = PartitionIdentifier PartitionDescriptionHash *HashStep
  HashStep = PartitionIdentifier SiblingHash ; C^uni = (β, dhash; (β_2, h_2), ..., (β_ℓ, h_ℓ)))

  UnicitySeal = Version NetworkIdentifier RootPartitionRoundNumber Epoch Timestamp PreviousHash Hash
       *Signatures ; C^r = (α, n_r, t_r, r_-, r; s)
                   ; where |Signatures| = quorumThreshold > 2f

; - Statistical and Technical Records -
  StatisticalRecord = NumRounds SumRoundSize SumStateSize MaxRoundSize MaxStateSize ; SR = (n_e, l̄_B, l̄_S, l̂_B, l̂_S)
  TechRecord = Round Epoch LeaderID SRHash ; TE = (n_r, e_r, ν_ℓ, h_sr)

  ShardInfo = Round PreviousHash PrevStatisticalRecord StatisticalRecord *ValidatorIdentifier
       LeaderIdentifier PrevUC ; (n, h_-, SR_-, SR, V, ν_ℓ, UC_-)

  ; Evolving trust base:
  UnicityTrustBase = Version *UnicityTrustBaseEntry
  UnicityTrustBaseEntry = NetworkIdentifier Epoch *(NodeID Pubkey Stake) QuorumThreshold StateHash
       ChangeRecordHash PreviousEntryHash *Signatures

; ===========================================================================
; BFT CORE (internal data structures)

; - Internal Consensus Structures -
  LedgerCommitInfo = UnicitySeal | UnicityTrustBaseEntry ; seal without signatures
  QC = VoteInfo LedgerCommitInfo *Signatures

  VoteInfo = RoundInfo
  RoundInfo = RoundNumber Epoch Timestamp ParentRoundNumber CurrentRootHash | UnicityTrustBaseEntryHash

; - BFT Core Messages -
  VoteMsg = VoteInfo LedgerCommitInfo HighQC Author Signature

  ProposalMsg = BlockData [LastRoundTc] Signature
  BlockData = Author Round Epoch Timestamp Payload AncestorQC
  Payload = *IRChangeReq | RPEpochChangeReq
  IRChangeReq = PartitionIdentifier CertReason *CR [EpochChangeJustification] SenderSignature
            ; presence of justification ==> epoch++
  RPEpochChangeReq = Epoch *(NodeID Pubkey Stake) QuorumThreshold StateHash ChangeRecordHash
       PreviousEntryHash SenderSignature [ChangeRecord]
```

[6] https://tools.ietf.org/html/rfc5234

```
  TimeoutMsg = Timeout Author Signature [LastTC]
     ; If HighQC is not from prev.  round then there must be TC of prev,
     ; justifying the incremented round number
  Timeout = Epoch Round HighQC ; HighQC - highest known Quorum Certificate to the validator
  TC = Timeout *Signatures ; 2f+1 Signatures ; TC - Timeout Certificate

  CertReason = 'quorum' | 'quorum-not-possible' | 't2-timeout' ; flags in appropriate encoding

; - Recovery Messages -
  GetStateMsg = NodeId ; id of the validator requesting the state
  StateMsg = *UC CommittedHead BlockNode UnicityTrustBaseEntry
  CommittedHead = BlockNode = RecoveryBlock
  RecoveryBlock = BlockData *InputData QC CommitQC
  InputData = PartitionIdentifier Shard IR Sdrh

; ============================================================================
; 3.  BFT CORE (protocols)

  CR = NetworkIdentifier PartitionIdentifier Shard NodeIdentifier IR BatchSize StateSize Signature
       [ConsistencyProof]
  ; Certification Request CR = ⟨α, β, σ, ν, IR, ℓ_B, ℓ_S; s; [π_CP]⟩
  ; Returned UC can be repeated cert for prevStateTreeHash which triggers next attempt using different
       leader
  ; a validator can have multiple pending requests extending the same hash; latest one is identified using
       IR.n

  CReS = NetworkIdentifier PartitionIdentifier ShardIdentifier UC TechRecord [UnicityTrustBaseEntry]
       ; Certification Response

  ; Subscription - Subscribe to CReS message feed in order to obtain the latest UC for synchronization
  SubscriptionMsg = NetworkIdentifier PartitionIdentifier Shard NodeIdentifier signature
       ; this request provides or updates validator connection parameters at
       ; transport layer so that Root can return CReS messages

  ; Trust Base Distribution protocol
  GetTrustBaseMsg = NetworkIdentifier EpochStart [EpochEnd]
  TrustBaseResponse = *UnicityTrustBaseEntry

; ============================================================================
; 4.  AGGREGATION LAYER

  StateIdentifier = Hash ; sid = H(pk, h_st)
  LeafData = TransactionHash ; value of key-value store
  SMTLeaf = StateIdentifier LeafData ; L = (sid, h_tx)

  InclusionCertificate = *InclusionStep ; C^inc = ⟨(p_1, d_1), (p_2, d_2), ..., (p_n, d_n)⟩
  InclusionStep = PathSegment (LeafData | SiblingHash)
     ; First step contains LeafData, subsequent steps contain sibling hashes
     ; Concatenated path segments equal the full StateIdentifier

  ConsistencyProof = Batch ProofSiblings | ZKProof
     ; π_CP - proves correct SMT manipulation (append-only)
     ; Hash-based:  batch and sibling hashes for verification
     ; ZK-compressed:  SNARK/STARK proof with public params

  Batch = *SMTLeaf ; Batch of new SMT leaves
  ProofSiblings = *DepthLayer ; Sibling hashes organized by tree depth
  DepthLayer = *SiblingHashPair
  SiblingHashPair = StateIdentifier Hash
  ZKProof = Bytes ; Opaque ZK proof (SNARK/STARK)

  InclusionProof = InclusionCertificate UC ; π^inc = (C^inc, UC)
     ; Complete proof from SMT leaf through UC to trust base
     ; Proves that a StateID EXISTS in the SMT
     ; SMT Leaf L = (sid, htx) can be extracted from InclusionCertificate:
     ;  sid = concatenation of all path segments p1 || p2 || ...  || pn
     ;  htx = d1 (leaf data from first pair)

  NonInclusionProof = InclusionCertificate UC ; π^exc = (C^inc, UC)
     ; Exclusion proof - proves that a StateID does NOT exist in the SMT
     ; Uses same structure as InclusionProof but with different validation
     ; Valid for queried sid if:
     ;  1.  VerifyInclusionProof returns TRUE
```

```
    ;  2.  p1 is a prefix of sid (queried StateID)
    ;  3.  d1 = empty (position is unoccupied)

; ============================================================================
; 5.  TOKEN / EXECUTION LAYER

  TokenState = OwnerPublicKey AuxiliaryData
    ; Token state S = (pk, aux)

  TransactionData = RecipientPublicKey BlindingMask RecipientAuxiliaryData [MintTransactionData]
    ; D = (pk′, x, aux′) unified for mint and transfer

  MintTransactionData = TokenId TokenType TokenData [CoinData] [MintReason]
    ; Additional fields for mint transactions (empty for transfers)

  CoinData = CoinId Balance ; for fungible tokens
  CoinId = Bytes32 ; 32-byte identifier for a specific coin type

  Transaction = CurrentStateHash TransactionData
    ; T = (h_st, D)

  TransactionHash = Hash ; h_tx = H(D)

  CertifiedTransaction = Transaction Signature TransactionHash InclusionProof
    ; (T, σ, h_tx, π^inc)

  UnicityServiceRequest = OwnerPublicKey CurrentStateHash TransactionHash Signature
    ; Q = (pk, h_st, h_tx, σ)
    ; Submitted to aggregation layer for state transition certification

  InclusionProofRequest = StateIdentifier ; sid = H(pk, h_st)
    ; Query if a StateID exists in the SMT

  InclusionProofResponse = Success [InclusionProof] [ErrorMessage]
    ; Response when StateID exists in the SMT

  NonInclusionProofRequest = StateIdentifier ; sid = H(pk, h_st)
    ; Query to prove a StateID does NOT exist in the SMT

  NonInclusionProofResponse = Success [NonInclusionProof] [ErrorMessage]
    ; Response proving StateID does not exist in the SMT
    ; Aggregator returns this when queried StateID is not present

  Token = Version CurrentTokenState GenesisCertifiedTransaction *CertifiedTransaction
    ; Complete token with full provenance chain
```

5 Governance

5.1 Introduction

By *governance*, we mean the operational management and re-configuration of a dynamic Unicity Platform instance. Governance processes manage the validators and other nodes, manage the lifecycle of partitions and govern sharding, manage incentives to the participants, and manage updates to the system.

Governance is an organizational process run by the Unicity Foundation, and potentially involving on-chain community voting ("coinvote") to make high-level governance decisions, which become binding to the Foundation. Governance parameters and algorithms can be changed only through the Governance.

5.1.1 Governance of the Dynamic Distributed Machine

The sections so far document a blockchain system which is entirely bootstrapped and managed by a centralized entity. This includes both administrative management and technical management, in the form of system administrators implementing requested changes. For example, if a validator wants to join the Unicity platform, the entity must give it explicitly a permit, and this is executed as a change made by system administrators. Analogously, in order to add a new partition, the change must be approved and executed by trusted entities. This may include restarting of the system.

This section is about implementing Unicity as a continuously running, decentralized and permission-less system. In the final form, it relies on algorithmic, on-chain governance implementing delegated proof of stake mechanisms and the economic security layer.

The launch as a permission-less, PoS controlled, decentralized blockchain is a fragile affair, due to initial instability (low number of validators, low locked stake, unpredictable usage patterns). Therefore, the system launches under the support and control of Unicity Foundation, and a gradual roadmap to full decentralization follows. While executing the roadmap, automated, on-chain governance processes, briefly described in the following sections, are taking over.

The governance is architected in a modular way. The Governance Body is an abstract entity, possibly implementing different mechanisms, like Proof of Authority, Proof of Stake, Delegated Proof of Stake.

The Governance Body implements a number of modular Governance Processes.

5.2 Data Flow

This section describes the “control plane” of Unicity Distributed Machine, complementing the “data plane” of user transaction processing.

5.2.1 Governance Body

The Governance Body is an abstract entity which manages the configuration of Unicity Network. It produces *change records*. Change record formats of different governance processes are documented in the following subsections.

5.2.1.1 How a Validator joins a Partition

Validators rely on configuration agents for performing any steps involving user-level communication with other partitions and shards. The process works like this:

- Configuration Agent detects a new Change Record it is involved with, learns the partition identifier and shard number
- Configuration Agent obtains the current epoch number from the shard
- Configuration Agent obtains the Change Records from current to the joining one (if not known)
- Configuration Agent activates the node process by making config API call providing necessary information
- Node continues as described in Section 4.4.5.1.

5.2.2 BFT Core

The BFT Core plays active role in enforcing and synchronizing the Governance. It initiates epoch changes and accepts UC Requests only from valid members of partitions/shards. In order to do so, it must have access to authentic change requests.

It is assumed, that the party who runs a BFT Core validator authenticates the change requests off-band.

5.3 Governance Mechanisms

5.3.1 Proof of Authority

In the Proof of Authority mechanism, a permissioned, *authorized* entity provides the governance information by sending signed transactions to the governance partition.

5.3.2 Proof of Stake

Proof of Stake mechanisms provide decentralization, permissionlessness and *Sybil attack*[7] protection.

Proof of Stake mechanism is based on the fact that there is a finite amount of certain backing asset, in Unicity’s case it is the native currency, ALPHA. When each validator

[7] An entity generates an arbitrary number of identities in order to obtain unfair representation strength in a distributed algorithm

presents a proof of locking a certain amount of ALPHA, it is guaranteed, that there is a real capital cost involved and hard economical limits of potential power grab.

Also, the staking mechanism provides a commitment and shared ownership structure for the Unicity Community.

Validator's voting power in consensus correlates to its staked amount. That is, votes have weights, and the quorum size is defined as the arithmetic sum of stakes of agreeing validators which is necessary to reach a consensus.

The leader election algorithm may use stake amounts as an input, so that the probability of becoming a leader correlates to the stake amount (assuming PoS).

5.3.3 Proof of Work

5.3.4 Tokenomics Toolbox

Tokenomics is the economic security layer of the Unicity Platform. It extends the security beyond traditional assumptions of distributed systems, e.g., a certain ratio of honest nodes.

Governance implements the tokenomics layer. In order to do so, there are certain tools and levers.

Incentives are payouts in ALPHA currency to incentivize validator participation in the Unicity System. There are incentives for staking (reward for capital cost) and incentives for providing secure and highly available validation service (reward for computation effort).

Reputation system improves the stability of the system, by incentivizing long-term identities and stable and high-quality behavior of validators.

Slashing is a mechanism enabled by staking: to disincentivize malicious behavior, it is possible to forfeit part or all of the stake as a punitive measure.

Stake Limits per validator allow the system to influence the number of participating physical validator machines; e.g., when there is an upper limit of staking rewards per machine, the stakers are incentivized to bring in more machine instances.

Liquid Staking is a mechanism where otherwise locked stake can be used for other purposes, for example used for staking in multiple partitions. This allows stakers to achieve better yield at the cost of higher risks.

Delegated Proof of Stake is an extension to PoS where external parties can delegate their state to specific validator nodes. Crucially, this democratizes the staking, so that non-sophisticated parties can participate, and also brings in external knowledge about the trustfulness of validators: delegators choose which validators they are placing their stakes on.

5.4 Governance Processes

Governance is divided logically to a set of independent processes. Following subsections detail the interface between Unicity Platform and the Governance: expectations on Governance Processes and their output data structures. This standardized interface allows Governance Body to be a plug-in replaceable component, for example to use either Proof of Authority or Delegated Proof of Stake implementations; or to change tokenomics mechanisms.

5.4.1 Validator Assignment

The process produces change records (called Validator Assignment Records), which assign validators to specific partitions and shards, as shown in Table 4.

Table 4. Change Record: Validator Assignment Record

No	Field	Notation	Type
1.	Network Identifier	α	$\mathbb{A}$
1.	Partition Identifier	β	$\mathbb{P}$
2.	Shard Identifier	σ	$\{0,1\}^{\leq \mathcal{SH}.k}$
3.	Epoch Number	e	$\mathbb{N}_{64}$
4.	Epoch Switching Condition	φ_e	$\mathbb{L}$
5.	Validator identifiers and stakes	$\{v, b_v\}_e$	$\{(\mathbb{OCT}^*, \mathbb{N}_{64})\} \cup \{\perp\}$
6.	Quorum Size (Voting power)	k_e	$\mathbb{N}_{16} \cup \{\perp\}$
7.	Hash of Previous Record	h_{e-1}	$\mathbb{H}$

The assignment of a particular validator is revoked by issuing a new record, which does not include the validator into the shard any more.

If a shard or entire partition stops working at the start of an epoch, the corresponding change record will have empty values for the validator identifiers and stakes as well as for the quorum size ($\{v, b_v\}_e = \perp \wedge k_e = \perp$). At all other times the values must not be empty.

The exact content of Epoch Switching Condition is left open to optimizations: it can be a suggested or mandatory time or round number, or an arbitrary predicate. For example, a Switching Condition may be a BFT Core round number range with soft enforcement: the responsible validators will not receive any fees for their work outside the expected epoch.

Initially, the Epoch Switching Condition is a fixed shard's round number: $\varphi_e = (n_{\beta,\sigma} > \texttt{constant})$

Quorum Size is measured in number of validators or total amount of stake behind validator votes to reach consensus. For the BFT Core, $k > 2/3 \sum b_v$. For shards, $k > 1/2 \sum b_v$.

Hash of Previous Record may be implemented inexplicitly if the underlying base partition provides hash linking of records.

5.4.2 Aggregation Layer Partition Lifecycle Management

This process creates, modifies and deletes partitions. The output is Partition Update Record with System Description Records and the configuration of consensus layer, as illustrated by Table 5.

The set of required fields depends on operation. Switching condition may be either aggregation shard or BFT Core round number, or aggregation shard epoch number switch.

5.4.3 Shard Management

This process creates and updates Sharding Schemes for partitions. The switch to a new sharding scheme is executed at epoch change; see Table 6.

Table 5. Change Record: Partition Update

No	Field	Notation	Type
1.	Network Identifier	α	$\mathbb{A}$
2.	Partition Identifier	β	$\mathbb{P}$
3.	Partition Description	$\mathcal{PD}[\beta]$	$\mathbb{PD}$
4.	Operation: {CREATE|CHANGE|DESTROY}	op	
5.	Hash of Genesis	h_g	$\mathbb{H}$ (optional)
6.	Switching Condition	φ_e	$\mathbb{L}$
7.	Number of Nodes	k	$\mathbb{N}_{32}$
8.	Target Round Rate	`t`	$\mathbb{N}_{32}$(ms)
9.	Time-outs	`t2`, `t3`	$\mathbb{N}_{32}$(ms)
10.	Hash of previous record	h_{e-1}	$\mathbb{H}$

Table 6. Change Record: Sharding Scheme Update

No	Field	Notation	Type
1.	Network Identifier	α	$\mathbb{A}$
2.	Partition Identifier	β	$\mathbb{P}$
3.	Sharding Scheme	$\mathcal{SH}$	$\mathbb{SH}$
4.	Switching Epoch Number	e	$\mathbb{N}_{64}$
5.	Hash of previous record	h_{e-1}	$\mathbb{H}$

The Input data for shard management process is collected by the BFT Core. It maintains a rolling statistics record for every shard for the ongoing shard epoch; and keeps the aggregates of the previous epoch. The rolling amount is modified based on reported measurements in UC Requests, as sent by the aggregation layer nodes.

Each shard is responsible for fetching the record certifying the previous epoch's aggregate statistics record, and submitting it as a transaction to the Governance Body. The aggregate fee record is certified by any UC of the following epoch.

A Bitstrings, Orderings, and Codes

A.1 Bitstrings and Orderings

By the *topological ordering* $\prec$ of $\{0,1\}^*$ we mean the irreflexive total ordering defined as follows: $c\|0\|x \prec c \prec c\|1\|y$ for all c, x, y in $\{0,1\}^*$.

For example, the subset $\{0,1\}^{\leq 2}$ is ordered as follows: $\{00 \prec 0 \prec 01 \prec \rfloor\!\lfloor \prec 10 \prec 1 \prec 11\}$.

A.2 Prefix-Free Codes

A *code* is a finite subset $\mathcal{C}$ of $\{0,1\}^*$. A code $\mathcal{C}$ is *prefix-free*, if no codeword $c \in \mathcal{C}$ is an initial segment of another codeword $c' \in \mathcal{C}$, i.e. if $c \in \mathcal{C}$, then $c\|c'' \notin \mathcal{C}$ for every bitstring $c'' \in \{0,1\}^*$.

For every code $\mathcal{C}$, the *closure* of $\mathcal{C}$ is the code $\overline{\mathcal{C}} = \{c \in \{0,1\}^* : \ \exists c'' : \ c\|c'' \in \mathcal{C}\}$, i.e. $\overline{\mathcal{C}}$ consists of all possible prefixes (including $\rfloor\!\lfloor$) of the codewords of $\mathcal{C}$.

A prefix-free code $\mathcal{C}$ is *irreducible*, if its closure $\overline{\mathcal{C}}$ satisfies the following property. For every $c \in \overline{\mathcal{C}}$, either $c \in \mathcal{C}$ or both $c\|0, c\|1 \in \overline{\mathcal{C}}$.

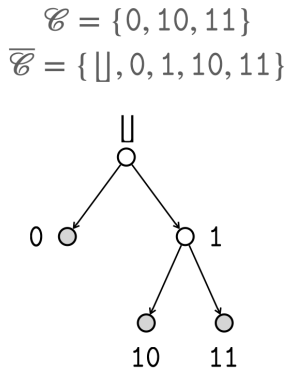

Figure 13. Irreducible prefix free code C and its closure $\overline{C}$.

B Encodings

B.1 CBOR

Most data structures on the Unicity platform are serialized in the Concise Binary Object Representation (CBOR), as defined in the IETF RFC 8949[8].

For data to be hashed or signed, the deterministic encoding rules (as defined in Sec. 4.2 of the RFC 8949) must be used.

B.2 Tuples

Tuples are encoded as CBOR arrays in the natural way: a tuple T consisting of n elements, denoted $(T_1, T_2, \ldots, T_n)$ or $\langle T_1, T_2, \ldots, T_n \rangle$ in the specification, is encoded as n-element CBOR array with T_1 the first elment, T_2 the second element, and so on.

An optional element with no value, denoted $\perp$ in the specification, is not omitted from the array, but encoded as the CBOR simple value `null`.

B.3 Bit-strings

Bit-strings are not directly supported in CBOR. On the Unicity platform, a bit-string is encoded as follows:

1. First, one 1-bit followed by zero to seven 0-bits are appended to the original bit-string so that the total number of bits is a multiple of 8.
2. Then, the padded result is encoded as a CBOR byte-string in the left-to-right, highest-to-lowest order.

For example, the 12-bit string '0101 1010 1111' is first padded to the 16-bit string '0101 1010 1111 1000', which is then encoded as the 3-byte sequence 0x425af8:

0x42	byte-string, length 2
0x5a	the bits '0101 1010'
0xf8	the bits '1111' and the padding '1000'

B.4 Hash Values

Hash values, although frequently defined as general bit-strings in cryptographic theory, in practice always have bit-lengths divisible by 8 and are encoded as simple CBOR byte-strings with no padding.

[8] `https://www.rfc-editor.org/rfc/rfc8949`

B.5 Unit Identifiers

Unit identifiers are encoded as CBOR byte-strings.

B.6 Time

Time is expressed as the number of milliseconds since 1970-01-01 00:00:00 UTC, encoded as an unsigned integer: the time value for 1970-01-01 00:00:00.000 UTC is 0, the value for 1970-01-01 00:00:01.000 UTC is 1000, etc. All days are considered to be exactly 86 400 seconds long, ignoring any leap seconds that have occurred in the past.

During a leap second, the part corresponding to full seconds (up from the fourth digit in decimal notation) is kept the same as during the previous second, but the part corresponding to fractions of a second (the three lowest digits in decimal notation) is reset to zero and counted up from there again. In other words, the value recorded for a time within a leap second is the same as the value recorded for the time exactly one second earlier.

B.7 Identifiers

Identifiers of Nodes (validators) are expressed as hashes of compressed ECDSA public keys; respective private key is controlled by the Node.

Identifiers of Transaction Systems are expressed as integers.

B.8 Cryptographic Algorithms

“ECDSA” denotes the Elliptic Curve Digital Signature Algorithm using secp256k1 curve

“SHA-256” denotes the SHA-2 hash algorithm with 256-bit output and “SHA-512” denotes the SHA-2 hash algorithm with 512-bit output; both are specified by NIST FIPS 180-4.

The list is non-exhaustive.

C Hash Trees

C.1 Plain Hash Trees

C.1.1 Function PLAIN_TREE_ROOT

Computes the root value of the plain hash tree with the given n values in its leaves.

Input: $L = \langle x_1, \ldots, x_n \rangle \in \mathbb{H}^n$, the list of the values in the n leaves of the tree

Output: $r \in \mathbb{H} \cup \{\bot\}$, the value in the root of the tree

Computation:

function PLAIN_TREE_ROOT(L)
  if $n = 0$ **then** ▷ $L = \langle\rangle$
    return $\bot$
  else if $n = 1$ **then** ▷ $L = \langle x_1 \rangle$
    return x_1
  else
    $m \leftarrow 2^{\lfloor \log_2(n-1) \rfloor}$ ▷ Canonical tree
    $L_{\text{left}} \leftarrow \langle x_1, \ldots, x_m \rangle$
    $L_{\text{right}} \leftarrow \langle x_{m+1}, \ldots, x_n \rangle$
    return $H(\text{PLAIN\_TREE\_ROOT}(L_{\text{left}}), \text{PLAIN\_TREE\_ROOT}(L_{\text{right}}))$
  end if
end function

Note that $2^{\lfloor \log_2(n-1) \rfloor}$ is the value of the highest 1-bit in the binary representation of $n - 1$, which may be the preferred way to compute m in some environments. Splitting the leaves this way results in a structure that allows the root of the tree to be computed incrementally, without having all the leaves in memory at once.

C.1.2 Function PLAIN_TREE_CHAIN

Computes the hash chain from the i-th leaf to the root of the plain hash tree with the given n values in its leaves.

Input:

1. $L = \langle x_1, \ldots, x_n \rangle \in \mathbb{H}^n$, the list of the values in the n leaves of the tree
2. $i \in \{1, \ldots, n\}$, the index of the starting leaf of the chain

Output: $C = \langle (b_1, y_1), \ldots, (b_\ell, y_\ell) \rangle \in (\mathbb{B} \times \mathbb{H})^\ell$, where y_j are the sibling hash values on the path from the i-th leaf to the root and b_j indicate whether the corresponding y_j is the right-

or left-hand sibling

Computation:

function PLAIN_TREE_CHAIN($L; i$)
  assert $1 \leq i \leq n$
  if $n = 1$ **then** ▷ $L = \langle x_1 \rangle$
    return $\langle\rangle$
  else
    $m \leftarrow 2^{\lfloor \log_2(n-1) \rfloor}$ ▷ Must match PLAIN_TREE_ROOT
    $L_{\text{left}} \leftarrow \langle x_1, \ldots, x_m \rangle$
    $L_{\text{right}} \leftarrow \langle x_{m+1}, \ldots, x_n \rangle$
    if $i \leq m$ **then**
      return PLAIN_TREE_CHAIN($L_{\text{left}}; i$)||(0, PLAIN_TREE_ROOT(L_{right}))
    else
      return PLAIN_TREE_CHAIN($L_{\text{right}}; i - m$)||(1, PLAIN_TREE_ROOT(L_{left}))
    end if
  end if
end function

C.1.3 Function PLAIN_TREE_OUTPUT

Computes the output hash of the chain C on the input x.

Input:

1. $C = \langle (b_1, y_1), \ldots, (b_\ell, y_\ell) \rangle \in (\mathbb{B} \times \mathbb{H})^\ell$, where y_j are the sibling hash values on the path from the i-th leaf to the root and b_j indicate whether the corresponding y_j is the right- or left-hand sibling
2. $x \in \mathbb{H}$, the input hash value

Output: $r \in \mathbb{H}$, output value of the hash chain

Computation:

function PLAIN_TREE_OUTPUT($C; x$)
  if $\ell = 0$ **then** ▷ $C = \langle\rangle$
    return x
  else
    assert $b_\ell \in \mathbb{B}$
    if $b_\ell = 0$ **then**
      return $H($PLAIN_TREE_OUTPUT$(\langle (b_1, y_1), \ldots, (b_{\ell-1}, y_{\ell-1}) \rangle; x), y_\ell)$
    else
      return $H(y_\ell,$ PLAIN_TREE_OUTPUT$(\langle (b_1, y_1), \ldots, (b_{\ell-1}, y_{\ell-1}) \rangle; x))$
    end if
  end if
end function

C.1.4 Inclusion Proofs

Plain hash trees can be used to provide and verify inclusion proofs. The process for this is as follows:

- To commit to the contents of a list $L = \langle x_1, \ldots, x_n \rangle$:
  - Compute $r \leftarrow$ PLAIN_TREE_ROOT(L).
  - Authenticate r somehow (sign it, post it to an immutable ledger, etc).
- To generate inclusion proof for $x_i \in L$:
  - Compute $C \leftarrow$ PLAIN_TREE_CHAIN$(L; i)$.
- To verify the inclusion proof $C = \langle (b_1, y_1), \ldots, (b_\ell, y_\ell) \rangle$ for x:
  - Check that PLAIN_TREE_OUTPUT$(C; x) = r$, where r is the previously authenticated root hash value.

C.2 Indexed Hash Trees

C.2.1 Function INDEX_TREE_ROOT

Computes the root value of the indexed hash tree with the given n key-value pairs in its leaves.

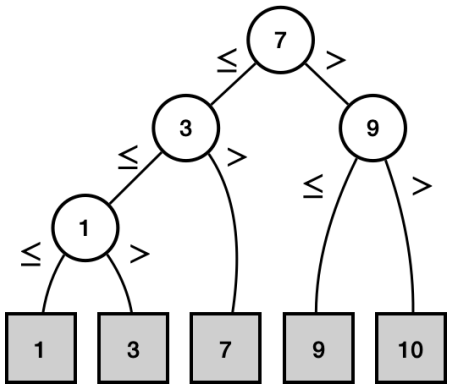

Figure 14. Keys of the nodes of an indexed hash tree.

Input: List $L = \langle (k_1, x_1), \ldots, (k_n, x_n) \rangle \in (\mathbb{K} \times \mathbb{H})^n$, the list of the key-value pairs in the n leaves of the tree; $\mathbb{K}$ must be a linearly ordered type and the input pairs must be strictly sorted in this order, i.e. $k_1 < \ldots < k_n$

Output: $r \in \mathbb{H} \cup \{\perp\}$, the value in the root of the tree

Computation:

```
function INDEX_TREE_ROOT(L)
    assert k_1 < ... < k_n
    if n = 0 then                                              ▷ L = ⟨⟩
        return ⊥
    else if n = 1 then                                         ▷ L = ⟨(k_1, x_1)⟩
        return H(1, k_1, x_1)
    else
        m ← ⌈n/2⌉                                              ▷ Most balanced tree
        L_left ← ⟨(k_1, x_1), ..., (k_m, x_m)⟩
        L_right ← ⟨(k_{m+1}, x_{m+1}), ..., (k_n, x_n)⟩
        return H(0, k_m, INDEX_TREE_ROOT(L_left), INDEX_TREE_ROOT(L_right))
    end if
end function
```

C.2.2 Function INDEX_TREE_CHAIN

Considers the indexed hash tree with the given n key-value pairs in its leaves. If there is a leaf containing the key k, computes the hash chain from that leaf to the root. If there is no such leaf, computes the hash chain from the leaf where k should be according to the ordering, which can be used as a proof of k's absence.

Input:

1. $L = \langle (k_1, x_1), \ldots, (k_n, x_n) \rangle \in (\mathbb{K} \times \mathbb{H})^n$, the list of the key-value pairs in the n leaves of the tree; $\mathbb{K}$ must be a linearly ordered type and the input pairs must be strictly sorted in this order, i.e. $k_1 < \ldots < k_n$
2. $k \in \mathbb{K}$, the key to compute the path for

Output: $C = \langle (k_1, y_1), \ldots, (k_\ell, y_\ell) \rangle \in (\mathbb{K} \times \mathbb{H})^\ell$, where k_j are the keys in the nodes on the path from the leaf to the root and y_j are the sibling hash values

Computation:

function INDEX_TREE_CHAIN($L; k$)
  assert $k_1 < \ldots < k_n$
  if $n \in \{0, 1\}$ **then** ▷ $L = \langle\rangle$ or $L = \langle (k_1, x_1) \rangle$
    return L
  else
    $m \leftarrow \lceil n/2 \rceil$ ▷ Must match INDEX_TREE_ROOT
    $L_{\text{left}} \leftarrow \langle (k_1, x_1), \ldots, (k_m, x_m) \rangle$
    $L_{\text{right}} \leftarrow \langle (k_{m+1}, x_{m+1}), \ldots, (k_n, x_n) \rangle$
    if $k \leq k_m$ **then**
      return INDEX_TREE_CHAIN($L_{\text{left}}; k$)$\|$(k_m, INDEX_TREE_ROOT(L_{right}))
    else
      return INDEX_TREE_CHAIN($L_{\text{right}}; k$)$\|$(k_m, INDEX_TREE_ROOT(L_{left}))
    end if
  end if
end function

C.2.3 Function INDEX_TREE_OUTPUT

Computes the output hash of the chain C on the input key k.

Input:

1. $C = \langle (k_1, y_1), \ldots, (k_\ell, y_\ell) \rangle \in (\mathbb{K} \times \mathbb{H})^\ell$, where k_j are the keys in the nodes on the path from the leaf to the root and y_j are the sibling hash values
2. $k \in \mathbb{K}$, the input key

Output: $r \in \mathbb{H} \cup \{\bot\}$, the value in the root of the tree

Computation:

function INDEX_TREE_OUTPUT($C; k$)
  if $\ell = 0$ **then** ▷ $C = \langle\rangle$
    return $\bot$
  else if $\ell = 1$ **then** ▷ $C = \langle (k_1, y_1) \rangle$

```
        return H(1, k_1, y_1)
    else
        if k ≤ k_ℓ then
            return H(0, k_ℓ, INDEX_TREE_OUTPUT(⟨(k_1, y_1), ..., (k_{ℓ-1}, y_{ℓ-1})⟩; k), y_ℓ)
        else
            return H(0, k_ℓ, y_ℓ, INDEX_TREE_OUTPUT(⟨(k_1, y_1), ..., (k_{ℓ-1}, y_{ℓ-1})⟩; k))
        end if
    end if
end function
```

C.2.4 Inclusion and Exclusion Proofs

Indexed hash trees can be used to provide and verify both inclusion and exclusion proofs. The process for this is as follows:

- To commit to the contents of a list $L = \langle (k_1, x_1), \ldots, (k_n, x_n) \rangle$ (where $k_1 < \ldots < k_n$):
  - Compute $r \leftarrow$ INDEX_TREE_ROOT(L).
  - Authenticate r somehow (sign it, post it to an immutable ledger, etc).
- To generate inclusion proof for $(k_i, x_i) \in L$:
  - Compute $C \leftarrow$ INDEX_TREE_CHAIN$(L; k_i)$.
- To verify the inclusion proof $C = \langle (k_1, y_1), \ldots, (k_\ell, y_\ell) \rangle$ for (k, x):
  - Check that INDEX_TREE_OUTPUT$(C; k) = r$, where r is the previously authenticated root hash value.
  - Check that $(k, x) = (k_1, y_1)$, where (k_1, y_1) is the first pair in the list C.
- To generate exclusion proof for $k \notin \{k_1, \ldots, k_n\}$:
  - Compute $C \leftarrow$ INDEX_TREE_CHAIN$(L; k)$.
- To verify the exclusion proof $C = \langle (k_1, y_1), \ldots, (k_\ell, y_\ell) \rangle$ for k:
  - Check that INDEX_TREE_OUTPUT$(C; k) = r$, where r is the previously authenticated root hash value.
  - Check that $k \neq k_1$, where (k_1, y_1) is the first pair in the list C.

C.3 Sparse Merkle Trees

A *Sparse Merkle Tree* (SMT) is a binary Merkle tree that maps keys from a large key space to values, where most positions in the key space are empty. The *path-compressed* form, documented below, retains only non-empty leaves and parent nodes with two children explicitly. The root node is an exception and may have zero, one, or two children.

C.3.1 Path Compression

In a path-compressed SMT, edges are labeled with path descriptions (bit-strings) that encode the path from child to parent.

- **Leaf node hash**: $h = H(p, d)$ where:
  - $p \in \{0, 1\}^*$ is the path label from this node to its parent

  - $d \in \mathbb{OCT}^*$ is the leaf data
- **Branch node hash**: $h = H(p, h_L, h_R)$ where:
  - $p \in \{0,1\}^*$ is the path label from this node to its parent
  - $h_L \in \mathbb{H}$ is the left child hash (or $\perp$ if empty)
  - $h_R \in \mathbb{H}$ is the right child hash (or $\perp$ if empty)
- **Root node**: Uses empty path ϵ (zero-length bit-string) in hash computation

C.3.2 Function SMT_LEAF_HASH

Computes the hash of an SMT leaf node.

Input:

1. $p \in \{0,1\}^*$ – path segment from this leaf to its parent
2. $d \in \mathbb{H}$ – leaf data (transaction hash)

Output: $h \in \mathbb{H}$ – hash of the leaf node

Computation:

function SMT_LEAF_HASH(p, d)
  return $H(p, d)$
end function

C.3.3 Function SMT_BRANCH_HASH

Computes the hash of an SMT branch node.

Input:

1. $p \in \{0,1\}^*$ – path segment from this node to its parent
2. $h_L \in \mathbb{H} \cup \{\perp\}$ – left child hash (or $\perp$ if empty)
3. $h_R \in \mathbb{H} \cup \{\perp\}$ – right child hash (or $\perp$ if empty)

Output: $h \in \mathbb{H} \cup \{\perp\}$ – hash of the branch node

Computation:

function SMT_BRANCH_HASH(p, h_L, h_R)
  if $h_L = \perp \wedge h_R = \perp$ **then**
    return $\perp$ ▷ Empty subtree
  else
    return $H(p, h_L, h_R)$
  end if
end function

Note: When hashing, $\perp$ values are encoded according to the serialization scheme (e.g., CBOR null).

C.3.4 Inclusion Certificate

An *Inclusion Certificate* for an SMT leaf with key sid is a sequence of pairs:

$$C^{\text{inc}} = \langle (p_1, d_1), (p_2, d_2), \ldots, (p_n, d_n) \rangle$$

where:

- (p_1, d_1) – the leaf's path segment and data
- (p_i, d_i) for $i > 1$ – path segment p_i to parent and sibling hash d_i
- The concatenated path $p_1 \| p_2 \| \ldots \| p_n$ = sid (full key)

C.3.5 Function SMT_VERIFY_INCLUSION

Verifies an inclusion certificate by reconstructing the root hash.

Input:

1. $k \in \mathbb{H}$ – key
2. $v \in \mathbb{H}$ – value
3. $C^{\text{inc}} = \langle (p_1, d_1), (p_2, d_2), \ldots, (p_n, d_n) \rangle$ – inclusion certificate
4. $h_{\text{root}} \in \mathbb{H}$ – expected root hash

Output: TRUE or FALSE

Computation:

function SMT_VERIFY_INCLUSION($k, v, C^{\text{inc}}, h_{\text{root}}$)
  if $d_1 \neq v$ **then**
    return FALSE ▷ Leaf value mismatch
  end if
  if $p_1 \| p_2 \| \ldots \| p_n \neq k$ **then**
    return FALSE ▷ Path doesn't match key
  end if
  $h \leftarrow H(p_1, d_1)$ ▷ Hash leaf node
  for $i \leftarrow 2$ to n **do**
    $b \leftarrow$ rightmost bit of p_{i-1}
    if $b = 0$ **then**
      $h \leftarrow H(p_i, h, d_i)$ ▷ Current is left child
    else
      $h \leftarrow H(p_i, d_i, h)$ ▷ Current is right child
    end if
  end for
  return $(h = h_{\text{root}})$
end function

The rightmost bit of path segment p_{i-1} determines whether the current node is the left child (bit is 0) or right child (bit is 1) of its parent.

C.3.6 Inclusion and Exclusion Proofs

Sparse Merkle Trees support both inclusion and exclusion (non-inclusion) proofs using the same certificate structure.

C.3.6.1 Inclusion Proof

To prove that a key-value pair (k, v) exists in an SMT:

- **Generation**: Extract the path from the leaf to the root, collecting sibling hashes at each level to form $C^{\text{inc}} = \langle (p_1, d_1), \ldots, (p_n, d_n) \rangle$
- **Verification**: Check that:
  1. SMT_VERIFY_INCLUSION$(k, v, C^{\text{inc}}, h_{\text{root}}) = \text{TRUE}$
  2. The leaf data $d_1 = h_{\text{tx}}$ matches the claimed value
  3. The concatenated path equals the claimed key: $p_1 \| \ldots \| p_n = \text{sid}$

C.3.6.2 Exclusion Proof (Non-Inclusion Proof)

To prove that a key k_q does NOT exist in an SMT:

- **Generation**: Find the position where k_q would be located if it existed. Generate a certificate $C^{\text{exc}} = \langle (p_1, d_1), \ldots, (p_n, d_n) \rangle$ showing this position (or its prefix) has empty value.
- **Verification**: Check that:
  1. The certificate reconstructs the correct root:
     SMT_VERIFY_INCLUSION$(k_p, d_1, C^{\text{exc}}, h_{\text{root}}) = \text{TRUE}$
     where $k_p = p_1 \| \ldots \| p_n$ is the path from the certificate
  2. The position is empty: $d_1 = \perp$ (no leaf exists at this location)

The exclusion proof shows the branch where sid_q would be if it existed, cryptographically proving its absence from the current tree state.

C.3.7 Consistency Proofs

A *Consistency Proof* for a Sparse Merkle Tree validates one round of operation by proving that the state transition from previous SMT root hash h' to current SMT root hash h was performed correctly: only new leaves were added, and no pre-existing leaves were removed or modified.

C.3.7.1 Structure

For a batch of state transitions processed in a single round, the consistency proof π_{CP} is a tuple:

$$\pi_{CP} = (B, \Pi)$$

where:

- $B = \langle (\text{sid}_1, h_{\text{tx},1}), (\text{sid}_2, h_{\text{tx},2}), \ldots, (\text{sid}_j, h_{\text{tx},j}) \rangle$ – batch of new SMT leaves, where each sid_i is a state identifier and $h_{\text{tx},i}$ is the corresponding transaction hash
- Π – proof data containing sibling hashes needed for verification

The proof data Π is organized by tree depth layers for efficient verification. For each newly inserted leaf, the sibling nodes on the path from the leaf to the root are collected. Siblings that are computable from other leaves in the batch are omitted.

C.3.7.2 Function SMT_GENERATE_CONSISTENCY_PROOF

Generates a consistency proof for a batch of new leaves inserted into an SMT[9].

Input:

1. B – batch of new leaves to insert
2. h' – SMT root hash before the round
3. SMT state before the round

Output: $\pi_{CP} = (B, \Pi)$ and updated SMT with root hash h

Process:

1. Insert all leaves from batch B into the SMT
2. Compute new root hash h
3. For each newly inserted leaf $(\text{sid}_i, h_{\text{tx},i})$:
   - 3.1 Collect sibling nodes on the path from leaf to root
   - 3.2 Discard siblings that are present in batch B or computable from it
4. Organize remaining siblings by depth layers into Π
5. Return (B, Π)

C.3.7.3 Function SMT_VERIFY_CONSISTENCY_PROOF

Verifies that a consistency proof correctly demonstrates the state transition from previous root hash h' to current root hash h.

Input:

1. $\pi_{CP} = (B, \Pi)$ – consistency proof where $B = \langle(\text{sid}_1, h_{\text{tx},1}), \ldots, (\text{sid}_j, h_{\text{tx},j})\rangle$
2. h' – previous SMT root hash (before the round)
3. h – current SMT root hash (after the round)

Output: TRUE or FALSE

Computation:

function SMT_VERIFY_CONSISTENCY_PROOF(π_{CP}, h', h)
  $(B, \Pi) \leftarrow \pi_{CP}$
  $B_\emptyset \leftarrow \langle(\text{sid}_1, \emptyset), (\text{sid}_2, \emptyset), \ldots, (\text{sid}_j, \emptyset)\rangle$ ▷ Empty leaves
  $h_\emptyset \leftarrow$ SMT_COMPUTE_TREE_ROOT($\Pi, B_\emptyset$) ▷ Compute root with empty leaves
  if $h_\emptyset \neq h'$ **then return** FALSE
  end if ▷ Verify previous state
  $h_B \leftarrow$ SMT_COMPUTE_TREE_ROOT(Π, B) ▷ Compute root with actual leaves
  if $h_B \neq h$ **then return** FALSE
  end if ▷ Verify current state
  return TRUE
end function

[9]Complete algorithm: github.com/../ndsmt.py

C.3.7.4 Function SMT_COMPUTE_TREE_ROOT

Computes the SMT root hash from a batch of leaves and proof siblings.

Input:

1. Π – proof data organized by depth layers
2. B – batch of leaves $\langle(\text{sid}_1, v_1), \ldots, (\text{sid}_j, v_j)\rangle$

Output: Root hash h_{root}

Computation:

function SMT_COMPUTE_TREE_ROOT(Π, B)
  $L \leftarrow B$ ▷ Current layer, initially leaves
  for $\ell \leftarrow 0$ to $d - 1$ **do** ▷ For each tree depth layer
    $L' \leftarrow []$ ▷ Next layer
    $m \leftarrow 0; n \leftarrow 0$ ▷ Indices into L and $\Pi[\ell]$
    while $m < |L|$ **do**
      $(\text{sid}_k, v_k) \leftarrow L[m]$
      $\text{sid}_p \leftarrow \lfloor \text{sid}_k/2 \rfloor$ ▷ Parent identifier
      isRight $\leftarrow \text{sid}_k \bmod 2$
      $\text{sid}_s \leftarrow 2 \cdot \text{sid}_p + (1 - \text{isRight})$ ▷ Sibling identifier
      if $\neg$isRight $\wedge\ m + 1 < |L| \wedge L[m+1].\text{sid} = \text{sid}_s$ **then**
        $v_s \leftarrow L[m+1].v$ ▷ Sibling is next in layer
        $m \leftarrow m + 1$ ▷ Skip sibling
      else if $n < |\Pi[\ell]| \wedge \Pi[\ell][n].\text{sid} = \text{sid}_s$ **then**
        $v_s \leftarrow \Pi[\ell][n].v$ ▷ Sibling from proof
        $n \leftarrow n + 1$
      else
        $v_s \leftarrow \emptyset$ ▷ Sibling is empty
      end if
      $v_p \leftarrow H(p_\ell, v_k, v_s)$ **if** isRight **else** $H(p_\ell, v_s, v_k)$
      $L' \leftarrow L' \| [(\text{sid}_p, v_p)]$
      $m \leftarrow m + 1$
    end while
    $L \leftarrow L'$
  end for
  if $|L| \neq 1$ **then return** $\perp$
  end if ▷ Must have exactly one root
  return $L[0].v$ ▷ Root hash
end function

where p_ℓ is the path segment for layer ℓ.

C.3.8 ZK-Compressed Consistency Proofs

For high-throughput scenarios with large batches, the hash-based consistency proof size of $O(j \cdot d)$ (where j is the batch size and d is the tree depth) may become a bandwidth bottleneck. Cryptographic zero-knowledge proofs (ZK-SNARKs or ZK-STARKs) can compress

the consistency proof to constant ($O(1)$) or logarithmic ($O(\log j)$) size while maintaining tight security guarantees.

C.3.8.1 ZK Proof Structure

A ZK-compressed consistency proof proves successful execution of the smt_verify_consistency_proof algorithm (Appendix C.3.7.3) with the following parameters:

- **Public inputs (instance)**: Values that must match the Input Record in the Certification Request:
  - h' – previous SMT root hash (matches $\mathcal{IR}.h'$)
  - h – current SMT root hash (matches $\mathcal{IR}.h$)
  - j – number of leaves in batch (matches ℓ_B in the Certification Request)
- **Private inputs (witness)**: proof (B, Π)

C.3.8.2 Verification

When receiving a ZK-compressed consistency proof, the verifier must:

1. Verify that the proof is proving the right statement (hash-based consistency proof verification algorithm)
2. Verify the ZK proof, obtaining authentic public inputs h', h, and j
3. Confirm these values match the claimed Input Record

Index