Alphabill Yellowpaper

Ahto Buldas, Märt Saarepera, Risto Laanoja, Ahto Truu

Specification

Version TestNet V1 (main/58345947)

November 29, 2024

Contents

Notation

Typographic Conventions

Names of types are set in "blackboard bold": $\mathbb{A}, \mathbb{B}, \ldots$

Common Constructions

$\perp$	nothing; missing or uninitialized value, or the indication of error
$a \in \mathbb{A}$	variable or constant a is of type $\mathbb{A}$
$\mathbb{A}[\mathbb{B}]$	dictionaries with elements of type $\mathbb{A}$ indexed by indices of type $\mathbb{B}$ (or partial functions from $\mathbb{B}$ to $\mathbb{A}$)
$a[b] = \perp$	dictionary a has no element with index b (or partial function a is not defined on argument value b)
$f\colon \mathbb{B} \to \mathbb{A}$	f is a total function from $\mathbb{B}$ to $\mathbb{A}$
$\mathbb{A}^*$	finite arrays of elements of type $\mathbb{A}$, including the empty array
$\mathbb{A}^k$	arrays of exactly k elements of type $\mathbb{A}$
$\mathbb{A}^{\leq k}$	arrays of at most k elements of type $\mathbb{A}$
$\mathbb{A}^{\geq k}$	arrays of at least k elements of type $\mathbb{A}$
$a[i]$	if a is of type $\mathbb{A}^*$, then $a[i]$ denotes the i-th element of a; numbering of elements starts from 1
$\lvert a \rvert$	if a is of type $\mathbb{A}^*$, then $\lvert a \rvert$ denotes the number of elements of type $\mathbb{A}$ in a; $a \in \mathbb{A}^{\lvert a \rvert}$
$a \parallel b$	concatenation of lists or bitstrings a and b
$\wedge$	logical AND operation
$\vee$	logical OR operation

Concrete Types

$\mathbb{N}_k$	k-bit unsigned integers $(0 \ldots 2^k - 1)$
$\{0,1\}^*$	finite bitstrings, including the zero-length string denoted by $\lfloor\rfloor$
$\mathbb{OCT}^*$	finite octet strings (sequences of 8-bit bytes)
$\mathbb{CHR}^*$	finite text strings (sequences of Unicode code points represented in the UTF-8 encoding)

1 General Description

1.1 Purpose

Alphabill Framework provides interoperability of all block-chained transactions systems of certain general type.

Transaction systems that fit to Alphabill Framework have:

- **units** u, each unit having an identifier ι and unit data D;
- **transactions** that create and delete units or change the data of the units.

The data of most types of units includes the owner predicate φ that is used to validate the next transaction manipulating that unit. Replacing the owner predicate of a unit effectively transfers its ownership.

Alphabill Framework:

- defines a *language* for describing the functionality of transaction systems: state and transactions (syntax and semantics),
- provides *libraries* and *toolkits* for developing block-chained transaction systems in Alphabill Framework,
- based on descriptions of transaction systems, *registers* and *assigns identifiers* β to partitions implementing them,
- provides *unicity certificate service* for the shards of registered partitions: unique state root hash h_β and transaction block root hash $h_{B\beta}$ and summary value V_β for every pair (n, β), where n is the sequence number of a (successful) round of the shard.

1.2 Alphabill Architecture

Based on the Alphabill Framework, new Transaction Systems are defined. Transaction systems are parameterized and instantiated as Partitions.

A Partition may be decomposed into arbitrary number of Shards in order to meet performance needs (Fig. 1).

All Shards and Partitions are implemented as a distributed machine in order to meet decentralization and availability needs.

The security of Alphabill system is intrinsic to the architecture.

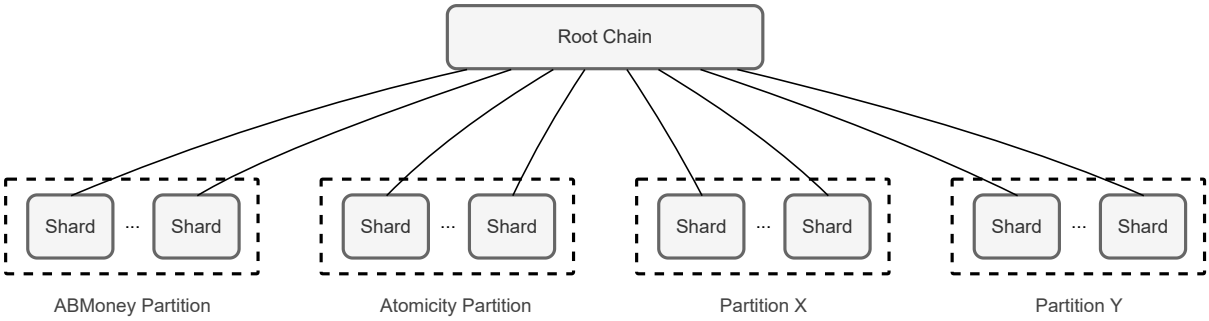

Figure 1. Alphabill functional model

2 Framework Data Structures

2.1 Parameters, Types, Constants

2.1.1 Parameters

tidlen – unit type identifier length of type $\mathbb{N}_8$ (per partition)
uidlen – unit identifier length of type $\mathbb{N}_8$ (per partition)

2.1.2 Types

$\mathbb{A} = \mathbb{N}_{16}$ – network identifiers
$\mathbb{P} = \mathbb{N}_{32}$ – partition identifiers
$\mathbb{IT} = \mathbb{OCT}^{\text{tidlen}}$ – unit type identifiers
$\mathbb{IU} = \mathbb{OCT}^{\text{uidlen}}$ – unit identifiers
$\mathbb{I} = \mathbb{IU} \times \mathbb{IT} = \mathbb{OCT}^{\text{uidlen+tidlen}}$ – extended identifiers, combining the type and the unit identifiers
$\mathbb{L}$ – predicates (the conditions defining unit ownership, token manipulation rights, etc.)
$\mathbb{H}$ – hash value type, output of hash function H of type $\mathbb{OCT}^* \to \mathbb{H}$
$\mathbb{UB}$ – unicity trust base type
$\mathbb{SP}$ – unit (state) proof type
$\mathbb{XP}$ – transaction (execution) proof type
$\mathbb{ST} = \mathbb{N}_8$ – system type identifiers
$\mathbb{T} = \mathbb{N}_{16}$ – transaction type identifiers
$\mathbb{AT}$ – abstract type of transaction attributes (formally the union of types of transaction attributes over all valid transaction types, $\bigcup \mathbb{AT}_\tau$ over all $\tau \in \mathbb{T}$)
$\mathbb{PT}$ – abstract type of transaction authorization proofs (formally the union of types of authorization proofs over all valid transaction types, $\bigcup \mathbb{PT}_\tau$ over all $\tau \in \mathbb{T}$)
$\mathbb{MC}$ – client-side transaction metadata type
$\mathbb{MS}$ – server-side transaction metadata type
$\mathbb{TO}$ – abstract transaction order type, including the abstract transaction attributes, client-side metadata, and the abstract authorization proofs
$\mathbb{TR}$ – abstract transaction record type, including the abstract transaction attributes, client-side metadata, the abstract authorization proofs, and server-side metadata
$\mathbb{SD}$ – system description type
$\mathbb{PD}$ – partition description type
$\mathbb{IR}$ – shard input record type
$\mathbb{US}$ – unicity seal type
$\mathbb{SH}$ – sharding scheme type

$\mathbb{CS}$ – state tree certificate type
$\mathbb{CU}$ – unit tree certificate type

2.1.3 Constants

$0_{\mathbb{I}}$ – zero identifier of type $\mathbb{I}$
$0_{\mathbb{H}}$ – zero-hash of type $\mathbb{H}$

2.2 Unit Identifiers

The structure of extended identifiers in a partition's state tree is defined by two parameters:

tidlen – the type identifier length (in bytes)
uidlen – the unit identifier length (in bytes)

The extended identifiers are the concatenation of the unit identifier part and the type identifier part, with the unit identifier in the uidlen first bytes and the type identifier in the tidlen last bytes of the extended identifier.

We also define the following convenience functions:

ExtrUnit : $\mathbb{I} \to \mathbb{IU}$ that extracts the unit identifier part from an extended identifier
ExtrType : $\mathbb{I} \to \mathbb{IT}$ that extracts the type identifier part from an extended identifier
NodeID : $\mathbb{IT} \times \mathbb{IU} \to \mathbb{I}$ that combines the type and unit identifiers into an extended identifier

For ordering, identifiers are compared lexicographically.

2.3 Sharding Schemes

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

If $\mathcal{SH} = \{\sqcup\}$, 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 $f_{\mathcal{SH}} \colon \mathbb{I} \to \mathcal{SH}$. The shard $f_{\mathcal{SH}}(\iota)$ responsible for handling the unit ι is the shard whose identifier σ_i is a prefix of ι. With $\mathcal{SH}$ an irreducible prefix-free code, there is exactly one such σ_i.

2.4 Networks

In practice, there will be several instances of Alphabill networks, each consisting of its own Root Chain 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 Alphabill 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.

2.5 System Type and System Type Descriptor

Every partition (an instance of a transaction system) registered in the Alphabill Framework with partition identifier β has a system type that is either:

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

2.5.1 System Type Descriptor

A *system type descriptor* is a tuple $\mathcal{SD} = (\mathcal{U}, \mathbb{D}, \mathbb{V}, V_0, V_S, F_S, \gamma) \in \mathbb{SD}$, where:

- $\mathcal{U} \in (\mathbb{OCT}^{\text{tidlen}})^*$ is the list of known unit type identifiers;
- $\mathbb{D}$ is the abstract unit data type (union of unit data types for all known unit types);
- $\mathbb{V}$ is the summary value type;
- $V_0 \in \mathbb{V}$ is the summary value of the data related to the unit with zero-identifier $0_{\mathbb{I}}$;
- $V_S : \mathbb{D} \to \mathbb{V}$ is the data summary function;
- $F_S : (\mathbb{V} \cup \{\bot\}) \times \mathbb{V} \times \mathbb{V} \to \mathbb{V}$ is the node summary function;
- $\gamma \colon \mathbb{V} \times \mathbb{V} \to \mathbb{B}$ is the summary check predicate.

2.5.2 Standard System Types

Money system (st $= 1$) – Defined in the Money Partition Type specification

User token system (st $= 2$) – Defined in the User Token Partition Type specification

Atomicity system (st $= 3$) – Defined in the Atomicity Partition Type specification

Orchestration system (st $= 4$) – Defined in the Orchestration Partition specification

2.5.3 Partition Description Record

A *partition description record* is a tuple $\mathcal{PD} = (\alpha, \beta, \text{st}, \mathcal{SD}, \text{tidlen}, \text{uidlen}, \mathcal{SH}, \mathcal{V}, \iota_{FC}, F_C) \in \mathbb{PD}$, where:

- $\alpha \in \mathbb{A}$ is the network identifier;
- $\beta \in \mathbb{P}$ is the partition identifier;
- st $\in \mathbb{ST}$ is the system type identifier (for standard types, st > 0);
- $\mathcal{SD} \in \mathbb{SD} \cup \{\bot\}$ is the system type descriptor (exists if st $= 0$);
- tidlen $\in \mathbb{N}_8$ is the unit type identifier length;
- uidlen $\in \mathbb{N}_8$ is the unit identifier length;
- $\mathcal{SH} \in \mathbb{SH}$ is the sharding scheme;

- $\mathcal{V} \in \mathbb{V}[\mathbb{SH}]$ lists the summary trust base values for each shard;
- $\iota_{FC} \in \mathbb{I}$ is the identifier of the bill in the money partition that represents the fee credits users have in this transaction system;
- $F_C \colon \mathbb{TO} \times \mathbb{S} \to \mathbb{N}_{64}$ is the transaction cost function.

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

2.6 Unit Type Identifiers

Each transaction system has a defined list of unit types that it supports. Each unit type has an identifier of type $\mathbb{IT} = \mathbb{N}_{\text{tidlen}}$. The identifier values $16 \dots 31$ are reserved for common unit types supported across many transaction systems (such as fee credit records, Sec. 4.3.1) and should not be assigned to unit types specific to just one transaction system.

2.7 Transaction Type Identifiers

Each transaction system has a defined list of transaction types that it supports. Each transaction type has an identifier of type $\mathbb{T} = \mathbb{N}_{16}$. The identifier values $16 \dots 31$ are reserved for common transaction types supported across many transaction systems (such as handling fee credits) and should not be assigned to transaction types specific to just one transaction system.

2.8 Transaction Orders and Records

A *transaction order* is a tuple $T = \langle \alpha, \beta, \iota, \tau, A, M_C, P, s_f \rangle$, with $M_C = (T_0, f_m, \iota_f, \rho)$, where

- $\alpha \in \mathbb{A}$ is the network identifier;
- $\beta \in \mathbb{P}$ is the partition identifier;
- $\iota \in \mathbb{I}$ is the unit identifier;
- $\tau \in \mathbb{T}$ is the transaction type identifier;
- $A \in \mathbb{AT}_\tau$ are the transaction attributes (a tuple whose contents are defined by the transaction type);
- $M_C \in \mathbb{MC}$ is the client metadata for the transaction, where
  - $T_0 \in \mathbb{N}_{64}$ is the transaction timeout;
  - $f_m \in \mathbb{N}_{64}$ is the maximum fee the user is willing to pay for the execution of this transaction;
  - $\iota_f \in \mathbb{I} \cup \{\bot\}$ is the optional identifier of the fee credit record;
  - $\rho \in \mathbb{OCT}^{\leq 32} \cup \{\bot\}$ is the optional reference number;
- $P \in \mathbb{PT}_\tau$ are the transaction authorization proofs (a tuple whose contents are defined by the transaction type);
- $s_f \in \mathbb{OCT}^* \cup \{\bot\}$ is the optional fee authorization proof.

Each transaction order T has an associated *set of target units* targets(T). In most cases targets(T) = $\{T.\iota\}$, but some transaction orders target multiple units. Such cases are highlighted in the sections defining those transaction types.

A *transaction record* is a transaction order with server-side metadata added to it. More formally, it is a tuple $T' = \langle \alpha, \beta, \iota, \tau, A, M_C, P, s_f, M_S \rangle$, with $M_S = (f_a, r, R)$, where

- α, β, ι, τ, A, M_C, P, s_f are as defined above;
- $M_S \in \mathbb{MS}$ is the service metadata for the transaction, where
  - $f_a \in \mathbb{N}_{64}$ is the actual fee charged for the processing of this transaction;
  - $r \in \mathbb{B}$ indicates whether the transaction was executed successfully; currently only successful transactions (with $r = 1$) are recorded in blocks; however, in the future also unsuccessful transactions (with $r = 0$) may be recorded and charged for;
  - $R \in \mathbb{RT}_\tau$ are the processing result details (a tuple whose contents are defined by the transaction type).

2.9 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. 2.10) proving e.g. execution of a transaction, or existence of a unit state.

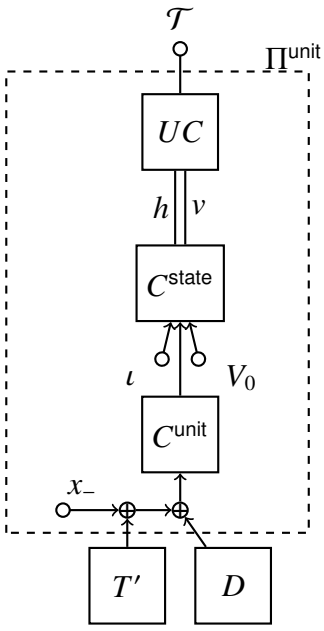

Figure 2. Chain of certificates forming a Unit Proof

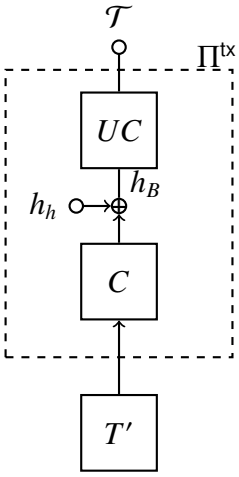

Figure 3. Chain of certificates forming a Transaction Proof

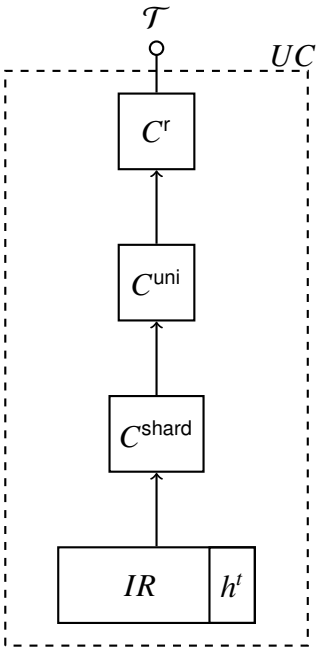

Figure 4. Unicity Certificate is a chain of certificates and a certified input data record IR

See Fig. 2, 3 and the exploded view of Unicity Certificate, Fig. 4. Transaction record T' and targeted unit's (identified by ι) state D—immediately after executing the transaction—are

certified by a chain of certificates. The final link, Unicity Seal (C^{r}), is validated based on the global root of trust: Unicity Trust Base $\mathcal{T}$.

2.9.1 Unit Tree Certificate

Unit Tree Certificate C^{unit} consists of the following components:

1. An initial tuple (t, s)
2. A list of tuples $(b_1, y_1), \ldots, (b_m, y_m)$

Here $s \in \mathbb{H}$ is the hash of the unit's state, computed as $s = H(D)$, and $t \in \mathbb{H}$ is the hash of the transaction that brought the unit to this state, computed as $t = H_d(T')$, where T' is the transaction record.

The $y_i \in \mathbb{H}$ are the sibling hash values on the path from the state's leaf to the root in the hash tree aggregating the state change log of the unit within one round, and $b_i \in \{0, 1\}$ indicates whether y_i is a right- or left-hand sibling.

2.9.1.1 Creation: CreateUnitTreeCert

Input:

1. ι – unit identifier to generate the certificate for
2. i – index of the intermediate state within the round to generate the certificate for
3. N – state tree (Sec. 4.2.2)

Output: Unit tree certificate C^{unit}

Computation:

$n \leftarrow |N[\iota].S|$
for $j \leftarrow 1$ to n **do**
  $s_j \leftarrow H(N[\iota].S_j.D)$
  $y_j \leftarrow H_0(N[\iota].S_j.x, s_j)$
end for
return $((N[\iota].S_i.t, s_i), \textsc{plain\_tree\_chain}(\langle y_1, \ldots, y_n \rangle, i))$

2.9.1.2 Computation: CompUnitTreeCert

Input:

1. x_- – hash value of type $\mathbb{H}$
2. $C^{\mathrm{unit}} = \langle (t, s); (b_1, y_1), \ldots, (b_m, y_m) \rangle$ – unit tree certificate of type $\mathbb{CU}$

Output: The unit tree root hash value of type $\mathbb{H}$

Computation:

$z \leftarrow H(H_0(x_-, t), s)$ ▹ H_0 defined in Sec. 4.6
return $\textsc{plain\_tree\_output}(\langle (b_1, y_1), \ldots, (b_m, y_m) \rangle; z)$ ▹ Sec. C.1.3

2.9.2 State Tree Certificate

State tree certificate C^{state} for (ι, z_0, V_0) consists of the following components:

1. An initial tuple $(h_L, V_L; h_R, V_R)$
2. A list of tuples $(\iota_1, z_1, V_1; h_1^s, V_1^s), \ldots, (\iota_m, z_m, V_m; h_m^s, V_m^s)$ such that $\iota_i \neq \iota$ for every $i \in \{1, \ldots, m\}$; if $\iota_i = \iota$ for some i, then the certificate must be considered invalid

2.9.2.1 Creation: CreateStateTreeCert

Input:

1. ι – extended identifier of type $\mathbb{I}$
2. N – state tree
3. ι_r – root node identifier
4. PD – partition description of type $\mathbb{PD}$

Output: State tree certificate C^{state}

Computation:

$C \leftarrow ()$
$\iota' \leftarrow \iota_r$
while $\iota' \notin \{\iota, 0_{\mathbb{I}}\}$ **do**
  $V \leftarrow \text{PD}.F_S(N[\iota'].D)$
  if $\iota < \iota'$ **then**
    $\iota_R \leftarrow N[\iota'].\iota_R$
    $C \leftarrow (\iota', N[\iota'].h_s, V; N[\iota_R].h, N[\iota_R].V) \| C$
    $\iota' \leftarrow N[\iota'].\iota_L$
  else
    $\iota_L \leftarrow N[\iota'].\iota_L$
    $C \leftarrow (\iota', N[\iota'].h_s, V; N[\iota_L].h, N[\iota_L].V) \| C$
    $\iota' \leftarrow N[\iota'].\iota_R$
  end if
end while
if $\iota' = \iota$ **then**
  $\iota_L \leftarrow N[\iota'].\iota_L$; $\iota_R \leftarrow N[\iota'].\iota_R$
  $C \leftarrow (N[\iota_L].h, N[\iota_L].V; N[\iota_R].h, N[\iota_R].V) \| C$
end if
return C

2.9.2.2 Computation: CompStateTreeCert

Input:

1. ι – unit identifier of type $\mathbb{I}$
2. z_0 – unit tree root hash of type $\mathbb{H}$
3. V_0 – unit summary value of type PD.$\mathbb{V}$ (type is not needed for the computation)
4. $C^{\text{state}} = \langle (h_L, V_L; h_R, V_R); (\iota_1, z_1, V_1; h_1^s, V_1^s), \ldots, (\iota_m, z_m, V_m; h_m^s, V_m^s) \rangle$ – state tree certificate of type $\mathbb{CS}$

5. PD – partition description of type $\mathbb{PD}$

Output: A pair (h, V), where h is the state tree root hash of type $\mathbb{H}$ and V is the state tree summary value of type PD.$\mathbb{V}$

Computation:

$V \leftarrow \text{PD}.F_S(V_0, V_L, V_R)$
$h \leftarrow H(\iota, z_0, V; h_L, V_L; h_R, V_R)$
for $i \leftarrow 1$ to m **do**
  if $\iota < \iota_i$ **then**
    $V' \leftarrow \text{PD}.F_S(V_i, V, V_i^s)$
    $h \leftarrow H(\iota_i, z_i, V'; h, V; h_i^s, V_i^s)$
  else
    $V' \leftarrow \text{PD}.F_S(V_i, V_i^s, V)$
    $h \leftarrow H(\iota_i, z_i, V'; h_i^s, V_i^s; h, V)$
  end if
  $V \leftarrow V'$
end for
return (h, V)

2.9.3 Shard Tree Certificate

Shard Tree Certificate is a tuple $C^{\text{shard}} = (\sigma; h_1^s, \ldots, h_{|\sigma|}^s)$, where:

1. $\sigma = \sigma_1\sigma_2 \ldots \sigma_{|\sigma|}$ is a shard identifier of type $\{0,1\}^*$
2. $h_1^s, \ldots, h_{|\sigma|}^s$ – sibling hashes of type $\mathbb{H}$

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

2.9.3.1 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 $C^{\text{shard}} = (\sigma; h_1^s, \ldots, h_{|\sigma|}^s)$

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 $(\bigsqcup, \bot)$.

2.9.3.2 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$((\lfloor\rfloor, \perp), IR, h_t)$ returns $H(IR \,\|\, h_t)$.

2.9.4 Unicity Tree Certificate

Unicity Tree Certificate 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 $\mathbb{P}$
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

2.9.4.1 Creation: CreateUnicityTreeCert

Input:

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

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

Computation:

$n \leftarrow |\mathcal{P}|$ ▷ Number of partitions
$P \leftarrow \text{sorted}(\mathcal{P})$ ▷ Sorted list
for $i \leftarrow 1 \ldots n$ **do**
  $x_i \leftarrow H(\mathcal{IH}[P_i] \,\|\, H(\mathcal{PD}[P_i]))$
end for
$\langle(\beta_1, h_1), (\beta_2, h_2), \ldots, (\beta_\ell, h_\ell)\rangle \leftarrow$ INDEX_TREE_CHAIN$((P_1, x_1), (P_2, x_2), \ldots, (P_n, x_n), \beta)$
return $(\beta, H(\mathcal{PD}[\beta]); (\beta_2, h_2), \ldots, (\beta_\ell, h_\ell))$ ▷ Drop redundant first hash step

2.9.4.2 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

2.9.5 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 – Root Chain Round number
3. e_r – Root Chain Epoch number
4. t_r – Round creation time (wall clock value specified and verified by the Root Chain), 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 Root Chain 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.

2.9.6 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.

2.9.6.1 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$(r, C^{\text{r}}, \mathcal{T}) = 1$

2.10 Proofs

2.10.1 Unit State Proof

Unit State Proof, also called *Unit Proof* is a tuple $\Pi^{\text{unit}} = (\iota, x_-, C^{\text{unit}}, V_0, C^{\text{state}}, UC)$, where:

1. ι – extended identifier of the unit, of type $\mathbb{I}$
2. x_- – previous state hash of type $\mathbb{H} \cup \{\perp\}$
3. C^{unit} – unit tree certificate (Sec. 2.9.1)
4. V_0 – data summary of type PD.$\mathbb{V}$
5. C^{state} – state tree certificate (Sec. 2.9.2)
6. UC – unicity certificate

2.10.1.1 Creation: CreateUnitProof

Input:

1. ι – unit identifier to generate proof for
2. i – index of the intermediate state within the round to generate proof for
3. N – state tree
4. ι_r – root node of the state tree
5. UC – Unicity Certificate
6. PD – partition description record

Output: Unit State Proof Π^{unit} of type $\mathbb{SP}$

Computation:

assert $1 \leq i \leq |N[\iota].S|$
if $i > 1$ **then**
  $x_- \leftarrow N[\iota].S_{i-1}.x$ ▹ Existing unit was updated by a transaction
else if $N[\iota].S_1.t = \perp$ **then**
  $x_- \leftarrow N[\iota].S_1.x$ ▹ Initial state was copied from previous round
else
  $x_- \leftarrow \perp$ ▹ Unit was created, no previous state
end if
$C^{\text{unit}} \leftarrow$ CreateUnitTreeCert(ι, i, N) ▹ Sec. 2.9.1.1
$C^{\text{state}} \leftarrow$ CreateStateTreeCert$(\iota, N, \iota_r, \text{PD})$ ▹ Sec. 2.9.2.1
return $(\iota, x_-, C^{\text{unit}}, N[\iota].V, C^{\text{state}}, UC)$

2.10.1.2 Verification: VerifyUnitProof

VerifyUnitProof – unit proof verification function of type $\mathbb{SP} \times \mathbb{UB} \times \mathbb{PD} \to \mathbb{B}$

Input:

1. $\Pi^{\text{unit}} = (\iota, x_{-}, C^{\text{unit}}, V_0, C^{\text{state}}, UC)$ – unit proof
2. $\mathcal{T}$ – trust base
3. PD – partition description of type $\mathbb{PD}$

Output: True or False

Computation:

$z \leftarrow \text{CompUnitTreeCert}(x_{-}, C^{\text{unit}})$
$(h, v) \leftarrow \text{CompStateTreeCert}(\iota, z, V_0; C^{\text{state}}, \text{PD})$
return $\text{VerifyUnicityCert}(UC, \mathcal{T}) \ \wedge \ UC.C^{\text{shard}}.\text{dhash} = H(\text{PD})$
$\wedge \ \gamma(v, \text{PD}.\mathcal{V}[f_{\text{PD}.\mathcal{SH}}(\iota)]) \ \wedge \ UC.IR.h = h \ \wedge \ UC.IR.v = v$

Note that a unit state proof Π can be used to prove and verify several different claims about a unit. Using the notation that the state tree certificate C^{state} in the unit proof is $\langle (h_L, V_L; h_R, V_R); (\iota_1, z_1, V_1; h_1^s, V_1^s), \ldots, (\iota_m, z_m, V_m; h_m^s, V_m^s) \rangle$ and the unit tree certificate C^{unit} is $\langle (t, s); (b_1, y_1), \ldots, (b_m, y_m) \rangle$, we can express the following conditions:

1. At some point during the round n, the unit ι had the data D:

$$\begin{aligned} &\text{VerifyUnitProof}(\Pi, \mathcal{T}, \text{PD}) = 1 \ \wedge \\ &\Pi.\iota = \iota \ \wedge \ \Pi.UC.IR.n = n \ \wedge \\ &\Pi.C^{\text{unit}}.s = H(D). \end{aligned}$$

2. At some point during the round n, the unit ι had the transaction in the record T' applied to it:

$$\begin{aligned} &\text{VerifyUnitProof}(\Pi, \mathcal{T}, \text{PD}) = 1 \ \wedge \\ &\Pi.\iota = \iota \ \wedge \ \Pi.UC.IR.n = n \ \wedge \\ &\Pi.C^{\text{unit}}.t = H(T'). \end{aligned}$$

3. At some point during the round n, the unit ι had the transaction in the record T' applied to it and this set the unit's data to D (essentially the conjunction of the two previous conditions):

$$\begin{aligned} &\text{VerifyUnitProof}(\Pi, \mathcal{T}, \text{PD}) = 1 \ \wedge \\ &\Pi.\iota = \iota \ \wedge \ \Pi.UC.IR.n = n \ \wedge \\ &\Pi.C^{\text{unit}}.t = H(T') \ \wedge \ \Pi.C^{\text{unit}}.s = H(D). \end{aligned}$$

4. The state of the unit ι did not change from the beginning of round n_1 to the end of

round n_2:

$$\begin{aligned}
&\mathsf{VerifyUnitProof}(\Pi_1, \mathcal{T}, \mathsf{PD}) = 1 \wedge \\
&\Pi_1.\iota = \iota \ \wedge \ \Pi_1.UC.IR.n = n_1 \wedge \\
&\Pi_1.C^{\mathsf{unit}}.b_1 = \Pi_1.C^{\mathsf{unit}}.b_2 = \ldots = \Pi_1.C^{\mathsf{unit}}.b_m = 0 \wedge \\
&\mathsf{VerifyUnitProof}(\Pi_2, \mathcal{T}, \mathsf{PD}) = 1 \wedge \\
&\Pi_2.\iota = \iota \ \wedge \ \Pi_2.UC.IR.n = n_2 \wedge \\
&\Pi_2.C^{\mathsf{unit}}.b_1 = \Pi_2.C^{\mathsf{unit}}.b_2 = \ldots = \Pi_2.C^{\mathsf{unit}}.b_m = 1 \wedge \\
&\Pi_1.x_- = \Pi_2.x_- \ \wedge \ \Pi_2.C^{\mathsf{unit}}.t = \bot.
\end{aligned}$$

5. The unit ι did not exist at the end of round n:

$$\begin{aligned}
&\mathsf{VerifyUnitProof}(\Pi, \mathcal{T}, \mathsf{PD}) = 1 \ \wedge \ \Pi.UC.IR.n = n \wedge \\
&(\iota < \Pi.\iota \ \wedge \ \Pi.h_L = 0_{\mathbb{H}} \ \vee \ \iota > \Pi.\iota \ \wedge \ \Pi.h_R = 0_{\mathbb{H}}) \wedge \\
&(\iota < \Pi.C^{\mathsf{state}}.\iota_1 \ \wedge \ \Pi.\iota < \Pi.C^{\mathsf{state}}.\iota_1 \ \vee \ \iota > \Pi.C^{\mathsf{state}}.\iota_1 \ \wedge \ \Pi.\iota > \Pi.C^{\mathsf{state}}.\iota_1) \wedge \\
&(\iota < \Pi.C^{\mathsf{state}}.\iota_2 \ \wedge \ \Pi.\iota < \Pi.C^{\mathsf{state}}.\iota_2 \ \vee \ \iota > \Pi.C^{\mathsf{state}}.\iota_2 \ \wedge \ \Pi.\iota > \Pi.C^{\mathsf{state}}.\iota_2) \wedge \\
&\ldots \\
&(\iota < \Pi.C^{\mathsf{state}}.\iota_m \ \wedge \ \Pi.\iota < \Pi.C^{\mathsf{state}}.\iota_m \ \vee \ \iota > \Pi.C^{\mathsf{state}}.\iota_m \ \wedge \ \Pi.\iota > \Pi.C^{\mathsf{state}}.\iota_m).
\end{aligned}$$

2.10.2 Transaction Execution Proof

Transaction Execution Proof (also *Transaction Proof*) for a transaction record T' is a tuple $\Pi^{\mathsf{tx}} = \langle h_h, C, UC \rangle$, where:

1. h_h – hash of block header fields
2. C – block tree hash chain
3. UC – unicity certificate

2.10.2.1 Creation: CreateTxProof

Input:

1. $B = \langle (\beta, \sigma, h_-, \nu); T'_1, \ldots, T'_k; UC \rangle$ – block
2. i – index of the transaction to generate proof for

Output: Transaction Execution Proof Π^{tx} of type $\mathbb{XP}$

Computation:

assert $1 \leq i \leq k$
$h_h \leftarrow H(\beta, \sigma, h_-, \nu)$
$C \leftarrow$ PLAIN_TREE_CHAIN$(\langle H(T'_1), \ldots, H(T'_k) \rangle, i)$ ▹ Sec. C.1.2
return (h_h, C, UC)

2.10.2.2 Verification: VerifyTxProof

VerifyTxProof – transaction proof verification function of type $\mathbb{TR} \times \mathbb{XP} \times \mathbb{UB} \times \mathbb{PD} \to \mathbb{B}$

Input:

1. T' – transaction record
2. $\Pi^{\text{tx}} = (h_h, C, UC)$ – transaction proof
3. $\mathcal{T}$ – trust base
4. PD – partition description of type $\mathbb{PD}$

Output: TRUE or FALSE

Computation:
return VerifyInc($(T', \Pi^{\text{tx}}), \mathcal{T}$, PD) $\wedge$ $(T'.M_S.r = 1)$

2.10.2.3 Verify Inclusion: VerifyInc

Verifies if a transaction is included into a block.

Input:

1. T' – transaction record
2. $\Pi^{\text{tx}} = (h_h, C, UC)$ – transaction proof
3. $\mathcal{T}$ – trust base
4. PD – partition description of type $\mathbb{PD}$

Output: TRUE or FALSE

Computation:
$h \leftarrow$ PLAIN_TREE_OUTPUT$(C, H(T'))$ ▹ Sec. C.1.3
$h \leftarrow H(h_h, UC.IR.h', UC.IR.h, h)$
return VerifyUnicityCert$(UC, \mathcal{T})$ $\wedge$ $UC.C^{\text{uni}}$.dhash $= H$(PD) $\wedge$ $UC.IR.h_B = h$

3 Root Partition

3.1 Data Structures of the Root Partition

3.1.1 Shard Input Record

Shard input record (IR) of a shard of a partition (of type $\mathbb{IR}$) is a tuple $(n, e, h', h, v, t, h_B, f_B)$, 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 root hash of type $\mathbb{H}$
4. h – current round's root hash of type $\mathbb{H}$
5. v – summary value of the current round; type $\mathbb{V}^*$, where $\mathbb{V}^* = \cup_{\beta \in \mathcal{P}} \mathcal{PD}[\beta].\mathbb{V}$
6. t – reference time for transaction validation of type $\mathbb{N}_{64}$
7. h_B – hash of the block B computed over all fields except certificates, type $\mathbb{H}$; computation is specified by the function BLOCK_HASH()
8. f_B – sum of the actual fees over all transaction records in the block, of type $\mathbb{N}_{64}$

3.1.2 Statistical Record

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

1. n_e – number of non-empty blocks (recording a state change), of type $\mathbb{N}_{64}$
2. $\overline{f}_B$ – total block fees, of type $\mathbb{N}_{64}$
3. $\overline{\ell}_B$ – the sum of all block sizes, of type $\mathbb{N}_{64}$
4. $\overline{\ell}_S$ – the sum of all state sizes, of type $\mathbb{N}_{64}$
5. $\hat{f}_B$ – maximum block fee, of type $\mathbb{N}_{64}$
6. $\hat{\ell}_B$ – maximum block size, of type $\mathbb{N}_{64}$
7. $\hat{\ell}_S$ – maximum state size, 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 Fee Record

Fee Record of type $\mathbb{VF}$ is a map VF : $\mathcal{V} \to f$, where:

1. $\mathcal{V}$ – the set of validator identifiers of the shard, each of type $\{0,1\}^*$
2. f – fee amount associated with a validator identifier, of type $\mathbb{N}_{64}$

There is one cumulative Fee Record of the validators of the current epoch and one invariant Fee Record of the validators of the previous epoch of every shard of every public partition.

3.1.4 Technical Record

Technical record of type $\mathbb{TE}$ is a tuple TE = $(n_r, e_r, \nu_\ell, h_{\text{sr}}, h_{\text{vf}})$, 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 $\mathbb{H}$
5. h_{vf} – hash of validator fee records; type $\mathbb{H}$

Technical record is delivered with an Unicity Certificate. When a shard is extending the Unicity Certificate with a next block production 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 block production attempt.

3.1.5 Shard Info

Shard info of type $\mathbb{SI}$ is a tuple $(n, e, h_{-}, \text{SR}_{-}, \text{SR}, \mathcal{V}, \mathcal{VF}_{-}, \mathcal{VF}, \nu_\ell, UC_{-})$, 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) $(0,0,0,0,0,0,0)$
6. $\mathcal{V}$ – validators of the shard, a set of identifiers, each of type $\{0,1\}^*$
7. $\mathcal{VF}_{-}$ – per validator total, invariant fees of the previous epoch of type $\mathbb{N}_{64}[\{0,1\}^*]$, where $\mathcal{VF}_{-}[\nu]$ is the total amount of fees taken by the validator with identifier ν during the previous epoch
8. $\mathcal{VF}$ – per validator summary fees of the current epoch of type $\mathbb{N}_{64}[\{0,1\}^*]$, where $\mathcal{VF}_{-}[\nu]$ is the monotonically increasing total amount of fees taken by the validator with identifier ν during the current epoch
9. ν_ℓ – leader identifier of type $\{0,1\}^*$ (it is assumed that $\nu_\ell \in \mathcal{V}$)
10. UC_{-} – last created unicity certificate

3.1.6 Shard Tree

For a partition β, let $\mathcal{SH}_\beta = \mathcal{PD}[\beta].\mathcal{SH}$, and $\mathcal{IRT}_\beta$ be of type $(\mathbb{IR}, \mathbb{H})[\mathcal{SH}_\beta]$, i.e. for every $\beta \in \mathcal{P}$ 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 $\mathcal{IRT}_\beta[\sigma] = 0_\mathbb{H}$.

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 $\chi_\beta(\sqcup)$ 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. 2.9.3.

3.1.6.1 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($\sqcup$)

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 $\{(\sqcup, H(\mathcal{IRT}[\sigma])\}$, i.e. it only has a single record for the root hash $\chi(\sqcup) = H(\mathcal{IRT}[\sigma])$.

3.2 Unicity Tree

Unicity Tree is an indexed Merkle tree.

3.2.1 **Creation:** CreateUnicityTree

Input:

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

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

Computation:

$n \leftarrow |\mathcal{P}|$ ▷ Number of partitions
$P \leftarrow \text{sorted}(\mathcal{P})$ ▷ Sorted list
for $i \leftarrow 1 \ldots n$ **do**
  $x_i \leftarrow H(\mathcal{IH}[P_i] \,||\, H(\mathcal{PD}[P_i]))$
end for
return INDEX_TREE_ROOT$((P_1, x_1), (P_2, x_2), \ldots, (P_n, x_n))$ ▷ Sec. C.2.1

3.3 State of the Root Partition

State of the root partition is a tuple $(\alpha, n, e, r_-, \mathcal{T}, \mathcal{P}, \mathcal{PD}, \mathcal{SI})$, where:

1. α – network identifier of type α
2. n – root partition's round number of type $\mathbb{N}_{64}$
3. e – root partition'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. $\mathcal{P}$ – set of partition identifiers, with elements of type $\mathbb{P}$
7. $\mathcal{PD}$ – partition descriptions of type $\mathbb{PD}[\mathbb{P}]$
8. $\mathcal{SI}$ – shard info of type $\mathbb{SI}[\mathbb{P}, \{0,1\}^*]$

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

3.4 Messages of the Root Partition

3.4.1 Certification Request

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

1. α – network identifier of type $\mathbb{A}$
2. β – partition identifier of type $\mathbb{P}$
3. σ – shard identifier of type $\mathbb{SH}$
4. ν – validator identifier of type $\{0,1\}^*$
5. $\mathcal{IR}$ – shard input record of type $\mathbb{IR}$
6. ℓ_B – block size in bytes of type $\mathbb{N}_{64}$
7. ℓ_S – state size in bytes of type $\mathbb{N}_{64}$
8. s – signature authenticating the message

The block size ℓ_B is computed as described in Sec. 4.7.6 and the state size ℓ_S is computed as described in Sec. 4.2.1.

Certification requests are sent to the root partition by the validators of shards.

3.4.2 Certification Response

Certification Response (CReS) of a shard of a partition 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 $\mathbb{P}$
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 root partition to validators of a shard of a partition as a response to a certification request message. This is an asynchronous message: response is not immediate, and there can be multiple subsequent responses.

3.5 Functional Description of the Root Partition

During every round, the root partition receives certification requests from the shards of partitions $\beta \in \mathcal{P}$.

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 root partition during the round, and the majority voting may fail (see the Consensus specification for details).

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

1. $\text{CR}.\alpha = \alpha$ – request came from the same network instance
2. $\text{CR}.\beta \in \mathcal{P}$ and $\sigma \in \mathcal{PD}[\text{CR}.\beta].\mathcal{SH}$ – request came from a legitimate shard of a registered partition
3. $\text{CR}.\nu \in \mathcal{SI}[\text{CR}.\beta, \text{CR}.\sigma].\mathcal{V}$ – request came from an authorized validator
4. $\text{CR}.\mathcal{IR}.n = \mathcal{SI}[\beta, \sigma].n + 1$ – round number is correctly incremented
5. $\text{CR}.\mathcal{IR}.e = \mathcal{SI}[\beta, \sigma].e$ – epoch number matches
6. $\text{CR}.\mathcal{IR}.h' = \mathcal{SI}[\beta, \sigma].h_-$ – previous round's root hash in IR matches the recorded one
7. $(\text{CR}.\mathcal{IR}.h = \text{CR}.\mathcal{IR}.h') = (\text{CR}.\mathcal{IR}.h_B = 0_{\mathbb{H}})$ – if current round's root hash coincides with the previous round's root hash, then the block must be empty, and its hash has to be the zero-hash; and vice versa
8. $\mathcal{PD}[\beta].\gamma(\text{CR}.\mathcal{IR}.v, \mathcal{PD}[\beta].\mathcal{V}[\sigma])$ – summary value check (of the shard) succeeds
9. $\text{CR}.\mathcal{IR}.t = \mathcal{SI}[\beta, \sigma].UC_-.IR.C^{\text{r}}.t$ – time reference is equal to the time field of the previous unicity seal

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

When the Root Partition's round n is completed, then:

1. For every $\beta \in \mathcal{P}$, and $\sigma \in \mathcal{SH}_\beta$ with accepted certificate request [1]:
   - 1.1 $\mathcal{SI}[\beta,\sigma].n \leftarrow \mathsf{CR}.\mathcal{IR}.n$
   - 1.2 $\mathcal{SI}[\beta,\sigma].h_{-} \leftarrow \mathsf{CR}.\mathcal{IR}.h$
   - 1.3 $\mathcal{SI}[\beta,\sigma].\mathsf{SR}.n_e \leftarrow \mathcal{SI}[\beta,\sigma].\mathsf{SR}.n_e + 1$
   - 1.4 $\mathcal{SI}[\beta,\sigma].\mathsf{SR}.\overline{f}_B \leftarrow \mathcal{SI}[\beta,\sigma].\mathsf{SR}.\overline{f}_B + \mathsf{CR}.\mathcal{IR}.f_B$
   - 1.5 $\mathcal{SI}[\beta,\sigma].\mathsf{SR}.\overline{\ell}_B \leftarrow \mathcal{SI}[\beta,\sigma].\mathsf{SR}.\overline{\ell}_B + \mathsf{CR}.\mathcal{IR}.\ell_B$
   - 1.6 $\mathcal{SI}[\beta,\sigma].\mathsf{SR}.\overline{\ell}_S \leftarrow \mathcal{SI}[\beta,\sigma].\mathsf{SR}.\overline{\ell}_S + \mathsf{CR}.\mathcal{IR}.\ell_S$
   - 1.7 $\mathcal{SI}[\beta,\sigma].\mathsf{SR}.\hat{f}_B \leftarrow \max\{\mathcal{SI}[\beta,\sigma].\mathsf{SR}.\hat{f}_B, \mathsf{CR}.\mathcal{IR}.f_B\}$
   - 1.8 $\mathcal{SI}[\beta,\sigma].\mathsf{SR}.\hat{\ell}_B \leftarrow \max\{\mathcal{SI}[\beta,\sigma].\mathsf{SR}.\hat{\ell}_B, \mathsf{CR}.\mathcal{IR}.\ell_B\}$
   - 1.9 $\mathcal{SI}[\beta,\sigma].\mathsf{SR}.\hat{\ell}_S \leftarrow \max\{\mathcal{SI}[\beta,\sigma].\mathsf{SR}.\hat{\ell}_S, \mathsf{CR}.\mathcal{IR}.\ell_S\}$
   - 1.10 Leader fees:
     $\mathcal{SI}[\beta,\sigma].\mathcal{VF}[\mathcal{SI}[\beta,\sigma].\nu_\ell] \leftarrow \mathcal{SI}[\beta,\sigma].\mathcal{VF}[\mathcal{SI}[\beta,\sigma].\nu_\ell] + \mathsf{CR}.\mathcal{IR}.f_B$,
     where it is assumed that $\mathcal{VF}[\bot] = 0$, even if $\mathcal{VF} = \emptyset$
   - 1.11 The leader for next round:
     $\mathcal{SI}[\beta,\sigma].\nu_\ell \leftarrow \textsc{leaderfunc}(UC_{-}, \mathcal{SI}[\beta,\sigma].\mathcal{V})$
   - 1.12 Technical record:
     $\mathcal{TE}_{\beta,\sigma} \leftarrow (\mathcal{SI}[\beta,\sigma].n + 1, \mathcal{SI}[\beta,\sigma].e, \mathcal{SI}[\beta,\sigma].\nu_\ell, h_{\mathsf{sr}}, h_{\mathsf{vf}})$,
     where $h_{\mathsf{sr}} = H(\mathcal{SI}[\beta,\sigma].\mathsf{SR}_{-}, \mathcal{SI}[\beta,\sigma].\mathsf{SR})$,
     and $h_{\mathsf{vf}} = H(\mathcal{SI}[\beta,\sigma].\mathcal{VF}_{-}, \mathcal{SI}[\beta,\sigma].\mathcal{VF})$,
     where the hash $H(\mathcal{VF}_{-}, \mathcal{VF})$ is computed assuming that the records indexed by validator identifiers ν are in the increasing lexicographic order.
2. For every registered partition $\beta \in \mathcal{P}$, the collected certification requests $\mathsf{CR}_{\beta,\sigma}$ are converted to a temporary data structure $\mathcal{IRT}_\beta$ of type $(\mathbb{IR}, \mathbb{H})[\mathcal{SH}_\beta]$, where $\mathcal{SH}_\beta = \mathcal{PD}[\beta].\mathcal{SH}$, and $\mathcal{IRT}_\beta[\sigma] = (\mathsf{CR}_{\beta,\sigma}.\mathcal{IR}, H(\mathcal{TE}_{\beta,\sigma}))$;
   if there is no request then respective leaf repeats its previous value. If there is no previous value, the leaf is initialized to $0_{\mathbb{H}}$.
3. For every $\beta \in \mathcal{P}$, the shard tree χ_β (Sec. 3.1.6) is created by the function call $\chi_\beta \leftarrow$ CreateShardTree($\mathcal{SH}_\beta, \mathcal{IRT}_\beta$) (Sec. 3.1.6.1).
4. A temporary data structure $\mathcal{IH}$ of type $\mathbb{H}[\mathcal{P}]$ is created such that $\mathcal{IH}[\beta] \leftarrow \chi_\beta(\bigsqcup)$ for every $\beta \in \mathcal{P}$, i.e. $\mathcal{IH}$ contains the root hashes of the shard trees.
5. The root of unicity tree is computed as $r \leftarrow$ CreateUnicityTree($\mathcal{P}, \mathcal{PD}, \mathcal{IH}$) (Sec. 3.2.1)
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 $\beta \in \mathcal{P}$, the unicity tree certificate C^{uni}_β is created by the function call $C^{\mathsf{uni}}_\beta \leftarrow$ CreateUnicityTreeCert($\beta, \mathcal{P}, \mathcal{PD}, \mathcal{IH}$) (Sec. 2.9.4.1)
8. For every $\beta \in \mathcal{P}$, and $\sigma \in \mathcal{SH}_\beta$:
   - 8.1 The shard tree certificate $C^{\mathsf{shard}}_{\beta,\sigma}$ (Sec. 2.9.3) is created by the function call $C^{\mathsf{shard}}_{\beta,\sigma} \leftarrow$ CreateShardTreeCert(σ, χ_β) (Sec. 2.9.3.1)

[1] Please refer to the Consensus chapter for details; notably a “request” may be induced for technical reasons

8.2 The unicity certificate $UC_{\beta,\sigma} = (\mathcal{IRT}_\beta[\sigma].\mathcal{IR}, \mathcal{IRT}_\beta[\sigma].h_t, C^{\text{shard}}_{\beta,\sigma}, C^{\text{uni}}_{\beta}, C^{\text{r}})$ (Sec. 2.9.6) and the certification response $\text{CReS}_{\beta,\sigma} = \langle \alpha, \beta, \sigma; UC_{\beta,\sigma}, \mathcal{TE}_{\beta,\sigma} \rangle$ 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 produced block ends, the following assignments are executed:

1. $\mathcal{SI}[\beta,\sigma].\text{SR}_{-} \leftarrow \mathcal{SI}[\beta,\sigma].\text{SR}$
2. $\mathcal{SI}[\beta,\sigma].\text{SR} \leftarrow (0,0,0,0,0,0,0)$
3. $\mathcal{SI}[\beta,\sigma].\mathcal{VF}_{-} \leftarrow \mathcal{VF}$ and $\mathcal{SI}[\beta,\sigma].\mathcal{VF} \leftarrow \emptyset$ – update the validator fees structure
4. $\mathcal{SI}[\beta,\sigma].e \leftarrow \mathcal{SI}[\beta,\sigma].e + 1$
5. The new validator set $\mathcal{SI}[\beta,\sigma].\mathcal{V}$ is chosen

4 Base Partition Type

4.1 Parameters

Every partition in the Alphabill network is completely defined by the following parameters:

1. α – network identifier of type $\mathbb{A}$
2. β – partition identifier of type $\mathbb{P}$
3. PD[β] $=$ (tidlen, uidlen, st, $\mathcal{SD}, \mathcal{SH}, \mathcal{V}, F_C, \iota_{FC}$) – partition description of type $\mathbb{PD}$. When specifying a partition with identifier β, we use the shorthand notation $\mathcal{SH} =$ PD[β].$\mathcal{SH}$
4. PrndSh – function of type $\mathbb{IU} \times \mathbb{OCT}^* \to \mathbb{IU}$ such that $f_{\mathcal{SH}}(\text{PrndSh}(\iota, X)) = f_{\mathcal{SH}}(\iota)$ for every ι and X
5. S_0 – initial state of type $\mathbb{S} = \mathbb{P} \times \{0,1\}^* \times \mathbb{SH} \times \mathbb{N}_{64} \times \mathbb{I} \times \mathbb{ND}[\mathbb{I}] \times \mathbb{UB} \times \mathbb{SD}[\mathbb{P}]$, where $\mathbb{ND} = ((\mathbb{H} \cup \{\bot\}) \times \mathbb{H} \times \mathbb{D})^* \times \mathbb{H} \times \mathbb{D} \times \mathbb{V} \times \mathbb{H} \times \mathbb{I} \times \mathbb{I}$ is the node type
6. RInit – round initialization procedure of type $\mathbb{S} \to \mathbb{S}$
7. RCompl – round completion procedure of type $\mathbb{S} \to \mathbb{S}$
8. $\mathbb{T}$ – transaction identifier type (a finite set)
9. For every $\tau \in \mathbb{T}$:
   - 9.1 $\mathbb{AT}_\tau$ – attributes type
   - 9.2 $\mathbb{PT}_\tau$ – transaction authorization proofs type
   - 9.3 $\mathbb{TO}_\tau = (\mathbb{P} \times \mathbb{T} \times \mathbb{I} \times \mathbb{AT}_\tau \times \mathbb{MC}) \times \mathbb{PT}_\tau \times (\mathbb{OCT}^* \cup \{\bot\})$ – derived transaction order type
   - 9.4 $\mathbb{TR}_\tau = \mathbb{TO}_\tau \times \mathbb{MS}$ – derived transaction record type
   - 9.5 ψ_τ – predicate of type $\mathbb{TO}_\tau \times \mathbb{S} \to \mathbb{B}$
   - 9.6 Action$_\tau$ – function of type $\mathbb{TO}_\tau \times \mathbb{S} \to \mathbb{S}$

Desirable features of the PrndSh function are:

1. *Collision resistance* – infeasibility of finding $X \neq X'$ and ι such that PrndSh$(\iota, X) =$ PrndSh(ι, X')
2. *Uniformity* – for any ι, and sufficiently large n, if $X \leftarrow \mathbb{OCT}^n$ is uniformly distributed in $\mathbb{OCT}^n$, then the probability distribution PrndSh(ι, X) is indistinguishable from the uniform distribution on the set $\{\iota' \in \mathbb{IU} \colon f_{\mathcal{SH}}(\iota') = f_{\mathcal{SH}}(\iota)\}$

For interoperability between different implementations, PrndSh is defined as

$$\text{PrndSh}(\iota, X) = \sigma_1\sigma_2 \ldots \sigma_\ell\chi_{\ell+1}\chi_{\ell+2} \ldots \chi_{8\cdot\text{uidlen}} \ ,$$

where $\sigma_1\sigma_2\ldots\sigma_\ell$ is the binary representation of $f_{\mathcal{SH}}(\iota)$ and $\chi_1\chi_2\ldots\chi_{8\cdot\text{uidlen}}$ is the output of a collision-resistant hash function $H^*\colon \mathbb{OCT}^* \to \{0,1\}^{8\cdot\text{uidlen}}$. In other words, PrndSh$(\iota, X)$ is $H^*(X)$ with the leftmost bits replaced with those of $f_{\mathcal{SH}}(\iota)$.

The function H^* in turn should be constructed from a collision-resistant hash function H with k-bit outputs ($H\colon \mathbb{OCT}^* \to \{0,1\}^k$) by taking $H^*(X) = h_1^0 h_2^0 \ldots h_k^0 h_1^1 h_2^1 \ldots h_k^1 \ldots h_1^m h_2^m \ldots h_i^m$, where $8\cdot\text{uidlen} = m\cdot k + i$ with $1 \le i \le k$ and $h_1^j h_2^j \ldots h_k^j$ is the binary representation of $H(X, j)$. In other words, $H^*(X)$ is the $8\cdot\text{uidlen}$ leftmost bits of the concatenation $H(X,0)\|H(X,1)\|\ldots$

4.2 Shard

4.2.1 State of a Shard

State of a shard of a partition is a tuple $(\alpha, \beta, \sigma, n, e, \iota_r, N, \mathcal{T}, \mathcal{PD})$, where:

1. α – network identifier of type $\mathbb{A}$
2. β – partition identifier of type $\mathbb{P}$
3. σ – shard identifier of type $\{0,1\}^{\le 8\cdot\text{PD}[\beta].\text{uidlen}}$
4. n – round number of type $\mathbb{N}_{64}$
5. e – epoch number of type $\mathbb{N}_{64}$
6. ι_r – root node identifier of type $\mathbb{I}$
7. N – state tree of type $\mathbb{ND}[\mathbb{I}]$, i.e. a node $N[\iota]$ of type $\mathbb{ND}$ is assigned to some identifiers ι of type $\mathbb{I}$
8. $\mathcal{T}$ – unicity trust base of type $\mathbb{UB}$
9. $\mathcal{PD}$ – partition descriptions of type $\mathbb{PD}[\mathbb{P}]$ for all registered partitions (including PD[β])

State size is $\sum_\iota |N[\iota].D|$, where the summation is over all ι with $N[\iota] \neq \bot$ and $|N[\iota].D|$ is the size (in bytes) of the same representation of $N[\iota].D$ that is used to compute the hash $H(D_i)$ in Sec. 4.2.2.2.

4.2.2 State Tree

State tree (N) is an AVL tree of State Tree Nodes (Sec. 4.2.2.1).

4.2.2.1 Node of the State Tree

Node $N[\iota]$ of type $\mathbb{ND}$ is a tuple $(S, h_s, D, V, h, \iota_L, \iota_R)$, with $S = (S_1, S_2, \ldots, S_n)$ and $S_i = (t_i, x_i, D_i)$, where:

1. S – log of state changes of the unit during the current round, with each record S_i consisting of
   1.1 t_i – the hash of the record of the transaction that brought the unit to the state described in S_i, of type $\mathbb{H} \cup \{\bot\}$
   1.2 x_i – the new head hash of the unit ledger, of type $\mathbb{H}$
   1.3 D_i – the new unit data of type $\mathbb{D}$
2. h_s – root value of the hash tree built on the state log S
3. D – current unit data of type $\mathbb{D}$

4. V – summary value of the subtree rooted at this node, of type $\mathbb{V}$
5. h – summary hash of the subtree rooted at this node, of type $\mathbb{H}$
6. ι_L – left child node identifier of type $\mathbb{I}$
7. ι_R – right child node identifier of type $\mathbb{I}$

4.2.2.2 Invariants of the State Tree

Definitions for the node with identifier $0_{\mathbb{I}}$:

$$N[0_{\mathbb{I}}].V = V_0$$
$$N[0_{\mathbb{I}}].h = 0_{\mathbb{H}}$$

For $\iota \neq 0_{\mathbb{I}}$ and $N[\iota] = (S, h_s, D, V, h, \iota_L, \iota_R) \neq \bot$:

$x_i = H_0(x_{i-1}, t_i)$ for all $i \in \{2, \ldots, |S|\}$ (Sec. 4.6)

$h_s = \textsc{plain\_tree\_root}(\langle z_1, \ldots, z_{|S|}\rangle)$, where $z_i = H(x_i, H(D_i))$

$D = D_{|S|}$

$V = F_S(V_s(D), N[\iota_L].V, N[\iota_R].V)$

$h = H(\iota, h_s, V; N[\iota_L].h, N[\iota_L].V; N[\iota_R].h, N[\iota_R].V)$

4.2.2.3 Unit Manipulation Functions

Actions on state tree nodes should be defined through the following helper functions:

1. AddItem(ι, D) – Adds a unit with identifier ι and data D; $N[\iota] \leftarrow (\bot, \langle\rangle, \bot, D, V, h, 0_{\mathbb{I}}, 0_{\mathbb{I}})$, where $V = V_S(D)$, and $h = H(\iota, \bot, V; 0_{\mathbb{H}}, V_0; 0_{\mathbb{H}}, V_0)$
2. DelItem(ι) – Deletes the unit with identifier ι

4.3 Transaction Fees

Fees provide the incentive for the validators to process transactions and thus effectively run the partitions. All fees in all Alphabill partitions are handled in the Alphabill native currency. The fees are expected to be low compared to the values of the transactions themselves and therefore a more lightweight credit balance based system is used to handle them.

The general process for fee payments consists of three phases:

- To prepare to transact on an application partition, the user first executes a special “transfer to fee credit” transaction on the money partition and presents a proof of the transaction to the application partition in order to obtain or top up a fee credit balance.
- Executing transactions on the application partition, the user gradually spends their fee credit.
- For each block, the application partition reports the sum of earned transaction fees to the root chain; based on that information, a governance process periodically issues payment orders on the money partition to pay out the fees earned to the application partition validators.

4.3.1 Fee Credit Records

To facilitate the above process, each application partition maintains a fee credit record for each user who has obtained a fee credit balance on that partition. Fee credit records are stored in the state tree of the application partition as nodes of a dedicated type with the node data $D = (b, \varphi, \ell, c, t)$, where:

1. b – current balance, of type $\mathbb{N}_{64}$; the value is represented in fixed point format with 8 fractional decimal digits; this means that a balance of 1 ALPHA is stored as $b = 100000000$ and $b = 123$ represents 0.00000123 ALPHA
2. φ – owner predicate of type $\mathbb{L}$
3. ℓ – lock status of the record, of type $\mathbb{N}_{64}$; allows locking of the record at the beginning of a multistep protocol that needs the transaction counter to remain unmodified by other transactions during the protocol execution; $\ell = 0$ means the record is not locked, any other value means it's locked; note that locking a record does not prevent spending the credit on the record to process other transactions, it only applies to actions directly targeting the record, like adding or reclaiming fee credits
4. c – transaction counter, of type $\mathbb{N}_{64}$; incremented with each "add fee credit", "close fee credit", "lock fee credit", and "unlock fee credit" operation with this record; note that spending fee credit when executing other transactions does not affect this value
5. t – the minimum lifetime of this record, expressed as the round number of type $\mathbb{N}_{64}$; when the balance goes to zero, the record may be "garbage collected" only after this round

On the other hand, the fee credits transferred by users to the partition β and not yet paid out to the validators of the partition are tracked as a special bill ι_β in the money partition. Such special bill is maintained also for the money partition itself.

4.3.2 Fee Credit Manipulation Functions

Actions on fee credit records should be defined through the following helper functions:

1. AddCredit$(\iota_f, v, \varphi_f, t)$ – Calls AddItem$(\iota_f, (v, \varphi_f, 0, 0, t))$; in other words, adds a new credit record $(v, \varphi_f, 0, 0, t)$ with the identifier ι_f
2. DelCredit(ι_f) – Calls DelItem(ι_f); in other words, deletes the credit record ι_f
3. IncrCredit(ι_f, v, t) – Sets $N[\iota_f].D.b \leftarrow N[\iota_f].D.b + v$, $N[\iota_f].D.c \leftarrow N[\iota_f].D.c + 1$, $N[\iota_f].D.\ell \leftarrow 0$, $N[\iota_f].D.t \leftarrow \max(N[\iota_f].D.t, t)$; note that $N[\iota_f].D.\varphi$ remains unchanged in this operation
4. DecrCredit(ι_f, v) – Sets $N[\iota_f].D.b \leftarrow N[\iota_f].D.b - v$; note that $N[\iota_f].D.\varphi$, $N[\iota_f].D.c$, $N[\iota_f].D.\ell$, and $N[\iota_f].D.t$ remain unchanged in this operation

4.4 Valid Transaction Orders

Let $S = (\alpha, \beta, \sigma, n, e, \iota_r, N, \mathcal{T}, \mathcal{PD})$ be a state where $N[\iota] = (S, h_s, D, V, h, \iota_L, \iota_R)$.

Transaction order $T = \langle \alpha, \beta, \iota, \tau, A, M_C, P, s_f \rangle$, with $M_C = (T_0, f_m, \iota_f, \rho)$, is *valid* if the following conditions hold:

1. $T.\alpha = S.\alpha$ – transaction is sent to this network

2. $T.\beta = S.\beta$ – transaction is sent to this partition
3. $f_{\mathcal{SH}}(T.\iota) = S.\sigma$ – target unit is in this shard
4. $S.n < T_0$ – transaction has not expired
5. ExtrType(ι_f) = fcr $\wedge$ $N[\iota_f] \neq \bot$ – the fee payer has credit in this shard
6. VerifyFeeAuth($N[\iota_f].D.\varphi, T, T.s_f$) – fee authorization proof satisfies the owner predicate of the fee credit record
7. $f_m \leq N[\iota_f].D.b$ – the maximum permitted transaction cost does not exceed the fee credit balance
8. $\mathcal{PD}.F_C(T, S) \leq f_m$ – the actual transaction cost does not exceed the maximum permitted by the user
9. $\psi_\tau(T, S)$ – type-specific validity condition holds

The "transfer to fee credit" and "reclaim fee credit" transactions in the money partition (see 5.2.7.1 and 5.2.7.4) and the "add fee credit", "close free credit", "lock fee credit", and "unlock fee credit" transactions in all application partitions (see 5.2.7.2, 5.2.7.35.2.7.5, and 5.2.7.6) are special cases: fees are handled intrinsically in those transactions; therefore, no separate fee authorization data (ι_f and s_f) should be present and the conditions 5 to 7 above do not apply.

4.4.1 Validation Helper Predicates

VerifyFeeAuth – fee authorization predicate evaluation function of type $\mathbb{L}\times\mathbb{TO}\times\mathbb{OCT}^* \to \mathbb{B}$; for predicate φ, the transaction order $T = \langle\alpha, \beta, \iota, \tau, A, M_C, P, s_f\rangle$, and the authorization proof s_f, the result is defined as $\varphi((\alpha, \beta, \iota, \tau, A, M_C, P), N[T.\iota], s_f)$; in other words, the predicate φ receives the tuple $(\alpha, \beta, \iota, \tau, A, M_C, P)$, the current state of the unit, and the proof s_f as inputs.

VerifyTxAuth – transaction authorization predicate evaluation function of type $\mathbb{L} \times \mathbb{TO} \times \mathbb{OCT}^* \to \mathbb{B}$; for predicate φ, the transaction order $T = \langle\alpha, \beta, \iota, \tau, A, M_C, P, s_f\rangle$, and the authorization proof s, the result is defined as $\varphi((\alpha, \beta, \iota, \tau, A, M_C), N[T.\iota], s)$; in other words, the predicate φ receives the tuple $(\alpha, \beta, \iota, \tau, A, M_C)$, the current state of the unit, and the proof s as inputs

4.5 Execution Round

4.5.1 Round Initialization: RInit

The round initialization procedure consists of the following steps:

1. Prune the state change history for all units that were targeted by transactions in the previous round:
   1.1 Find all such units: $\mathcal{I} \leftarrow \{\iota : N[\iota] \neq \bot \wedge |N[\iota].S| > 1\}$
   1.2 For all $\iota \in \mathcal{I}$:
      1.2.1 $x \leftarrow N[\iota].S_{|N[\iota].S|}.x$
      1.2.2 $N[\iota].S \leftarrow \langle(\bot, x, N[\iota].D)\rangle$
2. Delete all unlocked fee credit records with zero remaining balance and expired lifetime:

2.1 Find all such records:
$\mathcal{I} \leftarrow \{\iota : \text{ExtrType}(\iota) = \text{fcr} \wedge N[\iota] \neq \perp \wedge N[\iota].D.b = 0 \wedge N[\iota].D.\ell = 0 \wedge N[\iota].D.t < S.n\}$

2.2 For all $\iota \in \mathcal{I}$: DelItem(ι)

3. RInit_β – execute the partition specific initialization steps

4.5.2 Executing Transactions

Execution of the transaction order $T = \langle \alpha, \beta, \iota, \tau, A, M_C, P, s_f \rangle$, with $M_C = (T_0, f_m, \iota_f, \rho)$, consists of the following steps:

1. $M_S \leftarrow (S.\mathcal{PD}[\beta].F_C(T, S), 1, \perp)$ – initialize the transaction processing metadata (these initial values may be overwritten by Action_τ)
2. Action_τ – execute the type-specific actions
3. Append the transaction record and the new state to the change logs of all units affected by the transaction; for each $\iota \in \text{targets}(T)$:
   3.1 $t \leftarrow H_d(T \| M_S)$ – compute the hash of the transaction record
   3.2 If $|N[\iota].S| = 0$: – this is a freshly created unit
      3.2.1 $x \leftarrow H_0(\perp, t)$ – initialize the unit ledger
   3.3 If $|N[\iota].S| > 0$: – this is a pre-existing unit
      3.3.1 $x \leftarrow N[\iota].S_{|N[\iota].S|}.x$ – get the current head hash of the unit ledger
      3.3.2 $x \leftarrow H_0(x, t)$ – compute the new head hash of the unit ledger
   3.4 $N[\iota].S \leftarrow N[\iota].S \| (t, x, N[\iota].D)$ – append to the change log
4. DecrCredit($T.M_C.\iota_f, M_S.f_a$) – decrease the balance of the corresponding fee credit record

The “transfer to fee credit” and “reclaim fee credit” transactions in the money partition (see 5.2.7.1 and 5.2.7.4) and the “add fee credit”, “close free credit”, “lock fee credit”, and “unlock fee credit” transactions in all application partitions (see 5.2.7.2, 5.2.7.35.2.7.5, and 5.2.7.6) are special cases: fees are handled intrinsically in those transactions; therefore, step 4 above is skipped when processing those transactions.

4.5.3 Round Completion: RCompl

The round completion procedure consists of the following steps:

1. RCompl_β – execute the partition specific completion steps

4.6 Unit Ledger

Unit ledger is a list $R_1, R_2, \ldots, R_k$ of unit records.

Unit record is a tuple $R_i = (T'_i, C_i^{\text{unit}}, C_i^{\text{state}}, UC_i)$, where

1. T'_i – optional transaction record of type $\mathbb{TR} \cup \{\perp\}$,
2. C_i^{unit} – unit tree certificate,
3. C_i^{state} – state tree certificate,

4. UC_i – unicity certificate.

The unit tree certificate C_i^{unit} is computed from the current unit data D_ι and the ledger state hash x_ι of the unit ι. The certificate contents also depend on the values D'_ι, x'_ι of other states of the same unit.

The state tree certificate is computed from the identifier ι and the summary hash and summary value h_ι, V_ι of the unit ι. The certificate contents also depend on the summaries $h_{\iota'}$, $V_{\iota'}$ of other units.

The ledger state hash x_i is computed as

$$x_i = \begin{cases} H_0(x_{i-1}, H_d(T'_i)) & \text{if } i > 0 \\ \perp & \text{if } i = 0 \end{cases}$$

where

$$H_d(X) = \begin{cases} H(X) & \text{if } X \neq \perp \\ \perp & \text{if } X = \perp \end{cases}$$

and

$$H_0(X, Y) = \begin{cases} H(X, Y) & \text{if } Y \neq \perp \\ X & \text{if } Y = \perp \end{cases}$$

The structure of a unit ledger is depicted in Fig. 5

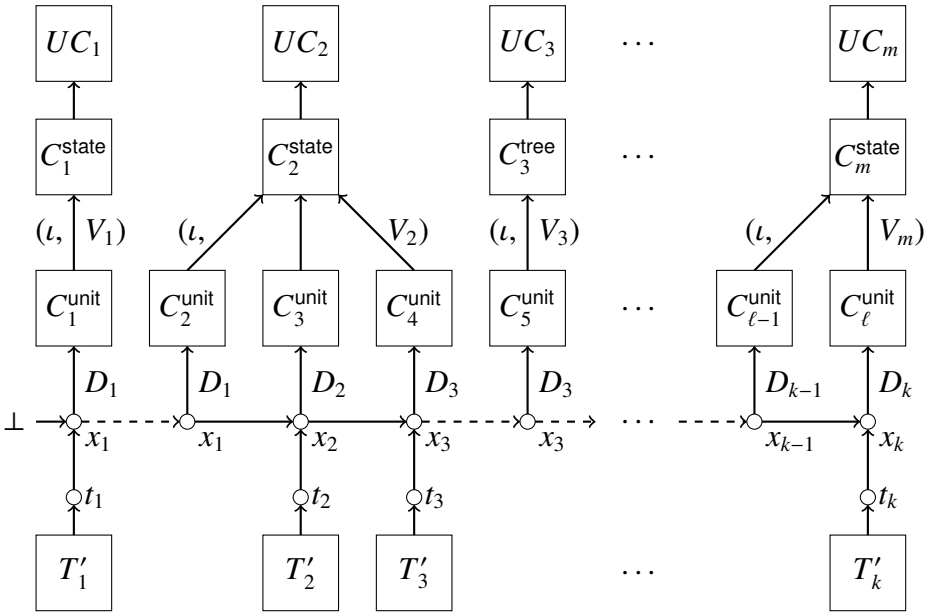

Figure 5. The structure of a unit ledger. In round 1, the unit was created by transaction T'_1. In round 2, the unit started in the state copied from round 1 and was then updated by transactions T'_2 and T'_3. The three states in which the unit was during round 2 have distinct unit tree certificates, but they share a common state tree certificate and a common unicity certificate. In round 3, the unit remained in the state copied from round 2 with no transactions. In round m, the unit was brought to its current state by transaction T'_k.

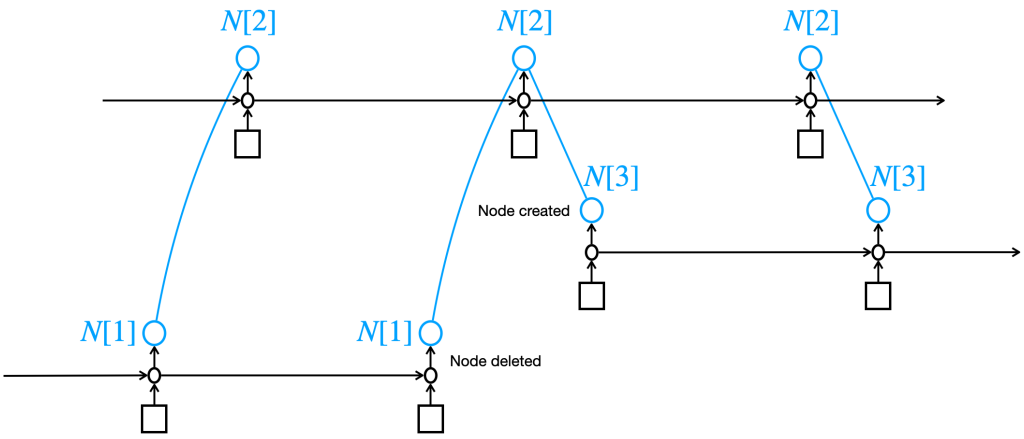

Figure 6. Evolution of the state tree and unit ledgers.

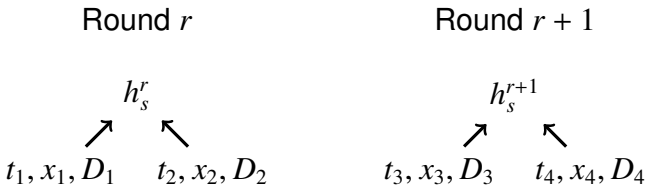

Figure 7. Evolution of a unit's state and the unit trees in two rounds.

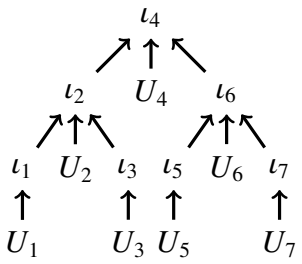

Figure 8. Embedding of unit trees in the state tree: ι_i are unit nodes in the state tree (these form a binary hash tree), U_i are the unit trees linked to each unit node.

4.7 Blocks

4.7.1 Block of a Shard

Block is a tuple $B = \langle(\beta, \sigma, h_-, \nu_{\text{prop}}); T'_1, \ldots, T'_k; UC\rangle$, where:

1. β – partition identifier
2. σ – shard identifier
3. h_- – hash of the previous block
4. ν_{prop} – block proposer's identifier
5. $T'_1, \ldots, T'_k$ – transaction records of the block
6. UC – unicity certificate

Existence of a transaction in a block is the proof of processing of this transaction during the block's round ($UC.IR.n$) and this changes the state: $k > 0 \Rightarrow UC.IR.h \neq UC.IR.h' \wedge$ BLOCK_HASH$(B) \neq 0_{\mathbb{H}}$. Inclusion implies transaction execution only if $T'.M_s.r = 1$.
If $k = 0$ and $UC.IR.h' = UC.IR.h$ then the block is *empty block* and BLOCK_HASH$(B) = 0_{\mathbb{H}}$. If $k = 0$ but the round initialization and finalization functions RInit, RCompl change the state then the block still carries information about the state change; we call the block *emptyish* and define the block hash so that it is not empty: $UC.IR.h \neq UC.IR.h' \Rightarrow$ BLOCK_HASH$(B) \neq 0_{\mathbb{H}}$.

Block's network is identified by $B.UC.C^{\mathsf{r}}.\alpha$.

4.7.2 Genesis Block of a Shard

Genesis block is a tuple $B_0 = \langle \alpha, \beta, \sigma, n, e, \iota_r, N, \mathcal{T}, \mathcal{PD} \rangle$ where:

1. α – network identifier of type $\mathbb{A}$
2. β – partition identifier of type $\mathbb{P}$
3. σ – shard identifier of type $\{0,1\}^{\leq 8 \cdot \mathsf{PD}[\beta].\mathsf{uidlen}}$
4. n – round number of type $\mathbb{N}_{64}$
5. e – epoch number of type $\mathbb{N}_{64}$
6. ι_r – root node identifier of type $\mathbb{I}$
7. N – state tree of type $\mathbb{ND}[\mathbb{I}]$, i.e. a node $N[\iota]$ of type $\mathbb{ND}$ is assigned to some identifiers ι of type $\mathbb{I}$
8. $\mathcal{T}$ – unicity trust base of type $\mathbb{UB}$
9. $\mathcal{PD}$ – partition descriptions of type $\mathbb{PD}[\mathbb{P}]$ for all registered partitions (including PD[β])

4.7.3 Block Creation: CreateBlock

Input:

1. State $(\alpha, \beta, \sigma, n, e, \iota_r, N, \mathcal{T}, \mathcal{PD})$
2. Sequence of transaction orders $T_1, \ldots, T_m$
3. Block hash of the previous block h_-, obtained as $h_- \leftarrow B'.UC.IR.h_B =$ BLOCK_HASH(B'), where B' is the preceding non-empty block.

Output: Block $B = \langle (\beta, \sigma, h_-, \nu); T'_1, \ldots, T'_k; UC \rangle$, where $T'_1, \ldots, T'_k$ are transaction records of successfully validated transactions in the same order as they appear in $(T_1, \ldots, T_m)$.

The procedure changes the state.

Computation:

Execute the round initialization procedure RInit
$k \leftarrow 0$
for $i \leftarrow 1 \ldots m$ **do**
  if T_i is valid in terms of Sec. 4.4 **then**
    Execute T_i as described in Sec. 4.5.2
    $k \leftarrow k + 1$
    $T'_k \leftarrow T_i || M_S$, where M_S is created during execution

Add T'_k to the block
end if
end for
Execute the round completion procedure RCompl
Record Block Proposer identifier ν_{prop} (defined and verified by the underlying consensus mechanism)
Send certification request $\langle \alpha, \beta, \sigma, \nu; \mathcal{IR}, \ell_B, \ell_S \rangle$ to the root partition, where ℓ_B is the block size computed as described in Sec. 4.7.6 and ℓ_S is computed as described in Sec. 4.2.1.
Obtain UC that certifies the block
return $B = \langle (\beta, \sigma, h_-, \nu_{\text{prop}}); T'_1, \ldots, T'_k; UC \rangle$

4.7.4 Block Verification: VerifyBlock

Input:

1. Block $B = \langle (\beta, \sigma, h_-, \nu); T'_1, \ldots, T'_k; UC \rangle$
2. Unicity trust base $\mathcal{T}$

Output: True or False

This function also checks if the network which created the block matches with the network instance identifier encoded in $\mathcal{T}$.

Computation:
$x \leftarrow$ BLOCK_HASH(B)
return (VerifyUnicityCert$(UC, \mathcal{T}) = 1 \ \wedge \ UC.IR.h_B = x$)

4.7.5 Block Hash: BLOCK_HASH

Hash of a block is computed as hash of (hash of block header fields || state change || tree hash of transactions).

Input: Block $B = \langle (\beta, \sigma, h_-, \nu); T'_1, \ldots, T'_k; UC \rangle$

Output: Hash of type $\mathbb{H}$

Computation:
function BLOCK_HASH(B)
if $k = 0$ **then**
if $UC.IR.h' = UC.IR.h$ **then** ▷ Empty block
return $0_{\mathbb{H}}$
else ▷ "Emptyish" with state change
return $H(H(\beta, \sigma, h_-, \nu), (UC.IR.h', UC.IR.h), 0_{\mathbb{H}})$
end if
else
for $i \leftarrow 1$ to k **do**
$h_i \leftarrow H(T'_i)$
end for
return $H(H(\beta, \sigma, h_-, \nu), (UC.IR.h', UC.IR.h),$ PLAIN_TREE_ROOT$(\langle h_1, \ldots, h_k \rangle))$
end if
end function

4.7.6 Block Size

The size of the block $B = \langle(\beta, \sigma, h_-, v); T'_1, \ldots, T'_k; UC\rangle$ is $\sum_{i=1}^{k} |T'_i|$, where $|T'_i|$ is the size (in bytes) of the same representation of T'_i that is used to compute the hash $h_i = H(T'_i)$ (Sec. 4.7.5).

5 Money Partition Type

5.1 Motivation and General Description

5.1.1 Pure Bill Money Schemes

Pure bill-type money schemes only have transfer type transactions (Fig. 9) that change the owner predicates of bills.

Pure bill schemes enable massively parallel decompositions of the money system, but also have shortcomings. Similar to physical cash, it is not always possible for a party to pay exact amounts and therefore, some additional services, like exchanges, are needed.

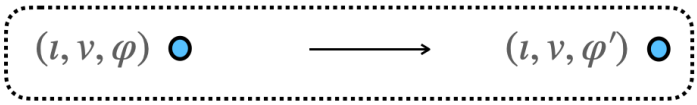

Figure 9. Bill transfer.

5.1.2 Extended Bill Money Scheme

Extended bill money scheme addresses the shortcomings of pure bill schemes by introducing split type payments (Fig. 10) that make exact payments always possible.

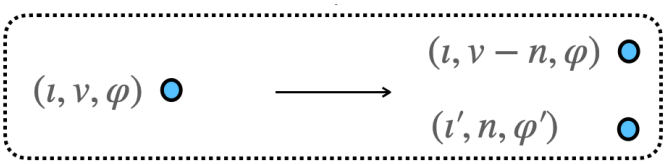

Figure 10. Bill split.

Split type transactions enable exact payments but introduce a new problem of having too many small-value bills (*dust bills*) in the end. Therefore, additional transactions and ledger mechanisms are needed to reduce the amount of dust bills by joining them to larger bills.

5.1.3 Dust Collection

Dust collection addresses the issue of dust bills by introducing new types of transactions as well as a new type of unit with value.

In the extended bill scheme, a special type of ownership – Dust Collector (DC) is used. Users can transfer their dust bills to DC via a special transfer type transaction transDC (Fig. 11) and get a proof of having done so.

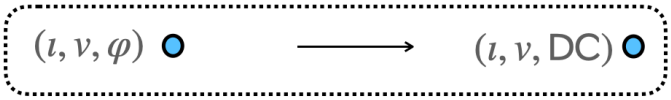

Figure 11. Transfer of dust bills to Dust Collector.

By presenting those proofs to the system, users can then obtain a new, larger-value bill via swapDC transactions (Fig. 12).

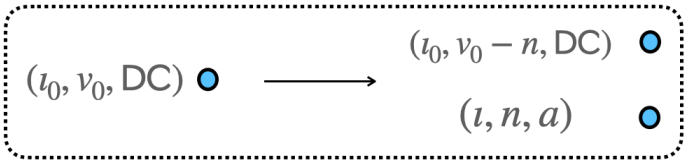

Figure 12. Swap with Dust Collector.

Formally, the Dust Collector (DC) controls a fraction of total money in the system. This money is called *dust collector money supply* and is represented as a special bill with identifier ι_{DC}. For issuing a new bill with value n to a user, the dust collector money supply is reduced by n.

If the system is sharded, then every shard must have its own DC money supply.

The transfers to DC and swaps with DC alone do not reduce the number of small-value bills in the system. There has to be a mechanism of joining the dust bills.

In the extended bill scheme, dust collection is introduced as a necessary automatic functionality related to block creation, i.e. every block creator has to regularly, as defined by the ledger rules, delete the dust bills and simultaneously rise the CB money supply by an equal amount. Such a method is depicted in Fig. 13, where dust bills $(\iota_1, v_1, DC), \ldots, (\iota_k, v_k, DC)$ are deleted and their value is added to the DC money supply by raising the value of the DC bill $(\iota_{\mathrm{DC}}, v_0, DC)$ by $d = v_1 + \ldots + v_k$.

All the activities related to dust collection preserve the total money of the system, including the DC money supply.

The DC money supply is just a technical system-related measure and not designed for actively supporting business transactions with the money.

5.1.4 Money Invariants

There are two types of money in Alphabill's ledger:

1. User money with total value v_{user} formed by the existing bills not owned by DC
2. Dust collector money with total value v_{DC} formed by the existing bills owned by DC and the DC money supply.

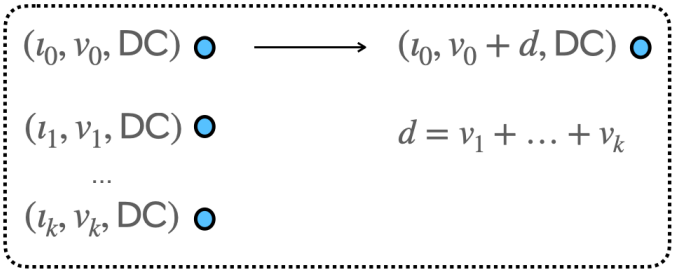

Figure 13. Dust collection.

Money invariant: The value $v_{\text{total}} = v_{\text{user}} + v_{\text{DC}}$ is constant in every shard

Swap money: The sum v_{swapDC} of the values of all dust bills paid to DC for which the swapDC has not yet been executed is not locally (shard-wise) verifiable, it is only verifiable from the global state (the combination of the states of all shards). We call such money *swap money*.

There are two more invariants that are not locally verifiable:

- $v^{\text{eff}}_{\text{user}} = v_{\text{user}} + v_{\text{swapDC}}$ – effective user money
- $v^{\text{eff}}_{\text{DC}} = v_{\text{DC}} - v_{\text{swapDC}}$ – effective dust collector money

These two invariants are locally verifiable only if $v_{\text{swapDC}} = 0$.

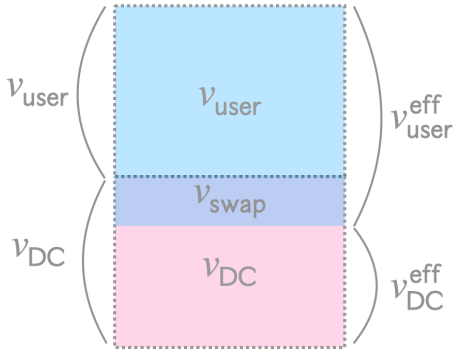

Figure 14. Types of money and money invariants.

5.2 Specification of the Money Partition

5.2.1 Parameters, Types, Constants, Functions

System type identifier: st $= 1$

Partition identifier for the primary instance managing the Alpha currency: $\beta_{\text{money}} = 1$

Type and unit identifier lengths for the primary instance: tidlen $= 1$, uidlen $= 32$

Summary value type $\mathbb{V}$: $\mathbb{N}_{64}$

Summary trust base: $\mathcal{V} = v_{\text{total}}$

Summary check: $\gamma(V, v_{\text{total}}) \equiv V = v_{\text{total}}$

Unit types: $\mathcal{U} = \{\text{bill} = 1, \text{fcr} = 16\}$ (bills, fee credit records)

Unit data $\mathbb{D}_u$ depends on the unit type u as follows:

- $\mathbb{D}_{\text{bill}}$: tuples (v, φ, ℓ, c) where:
  - v – value of type $\mathbb{N}_{64}$; the value is represented in fixpoint format with 8 fractional decimal digits; this means that a bill with value 1 ALPHA has $v = 100000000$ and $v = 123$ represents 0.00000123 ALPHA
  - φ – current owner predicate of type $\mathbb{L}$
  - ℓ – lock status of the bill, of type $\mathbb{N}_{64}$; allows locking of the bill at the beginning of a multi-step protocol that needs the bill to remain unmodified by other transactions during the protocol execution; $\ell = 0$ means the bill is not locked, any other value means it's locked
  - c – transaction counter of type $\mathbb{N}_{64}$
- $\mathbb{D}_{\text{fcr}} = (b, \varphi, \ell, c, t)$, where
  - $b \in \mathbb{N}_{64}$ is the current balance of this record, in fixpoint format with 8 fractional decimal digits
  - φ – owner predicate of type $\mathbb{L}$
  - $\ell \in \mathbb{N}_{64}$ is the lock status of the record; $\ell = 0$ means the record is not locked, any other value means it's locked
  - c – transaction counter of type $\mathbb{N}_{64}$
  - $t \in \mathbb{N}_{64}$ is the minimum lifetime of this record

Summary functions:

- $V_s(D) = D.v$ for $\mathbb{D}_{\text{bill}}$, or 0 otherwise
- $F_S(v, v_L, v_R) = v + v_L + v_R$
- $F_S(\bot, v_L, v_R) = v_L + v_R$

Summary value of zero-unit: $N[0_{\mathbb{I}}].V = 0$

Transaction types: $\mathbb{T} = \{\text{transB} = 1, \text{splitB} = 2, \text{transDC} = 3, \text{swapDC} = 4, \text{lockB} = 5, \text{unlockB} = 6, \text{transFC} = 14, \text{reclFC} = 15, \text{addFC} = 16, \text{closeFC} = 17, \text{lockFC} = 18, \text{unlockFC} = 19\}$ (transfer a bill, split a bill, transfer to dust collector, swap with dust collector, lock a bill, unlock a bill, transfer to fee credit, reclaim fee credit, add fee credit, close fee credit, lock a fee credit record, unlock a fee credit record)

5.2.2 Transfer a Bill

Transaction order $T = \langle \alpha, \beta, \iota, \text{transB}, A, M_C, P, s_f \rangle$ with $A = (v, \varphi, c)$, $P = (s)$ and $M_C = (T_0, f_m, \iota_f, \rho)$, where:

- $A.v \in \mathbb{N}_{64}$ is the amount to transfer;
- $A.\varphi \in \mathbb{L}$ is the new owner predicate;
- $A.c \in \mathbb{N}_{64}$ is the transaction counter value;

- $P.s \in \mathbb{OCT}^*$ is the owner proof.

Transaction-specific validity condition:

$$\begin{aligned}&\psi_{\mathsf{transB}}(T, S) \equiv \\ &\quad \mathsf{ExtrType}(T.\iota) = \mathsf{bill} \ \wedge \ S.N[T.\iota] \neq \bot \ \wedge \\ &\quad S.N[T.\iota].D.\ell = 0 \ \wedge \\ &\quad T.A.v = S.N[T.\iota].D.v \ \wedge \\ &\quad T.A.c = S.N[T.\iota].D.c \ \wedge \\ &\quad \mathsf{VerifyTxAuth}(N[T.\iota].D.\varphi, T, T.P.s) = 1\end{aligned}$$

That is,

- $T.\iota$ identifies an existing bill,
- the bill is not locked,
- the value to be transferred is the value of the bill,
- the order contains the correct transaction counter value, and
- the owner proof satisfies the bill's current owner predicate.

Actions $\mathsf{Action}_{\mathsf{transB}}$:

1. $N[T.\iota].D.\varphi \leftarrow T.A.\varphi$
2. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

5.2.3 Split a Bill

Transaction order $T = \langle \alpha, \beta, \iota, \mathsf{splitB}, A, M_C, P, s_f \rangle$ with $A = ((v_1, \varphi_1), \ldots, (v_m, \varphi_m); c)$, $P = (s)$ and $M_C = (T_0, f_m, \iota_f, \rho)$, where:

- $A.v_1, \ldots, A.v_m \in \mathbb{N}_{64}$ are the amounts to transfer;
- $A.\varphi_1, \ldots, A.\varphi_m \in \mathbb{L}$ are the new owner predicates;
- $A.c \in \mathbb{N}_{64}$ is the transaction counter value;
- $P.s \in \mathbb{OCT}^*$ is the owner proof.

Transaction-specific validity condition:

$$\begin{aligned}&\psi_{\mathsf{splitB}}(T, S) \equiv \\ &\quad \mathsf{ExtrType}(T.\iota) = \mathsf{bill} \ \wedge \ S.N[T.\iota] \neq \bot \ \wedge \\ &\quad S.N[T.\iota].D.\ell = 0 \ \wedge \\ &\quad T.A.v_1 > 0 \ \wedge \ \ldots \ \wedge \ T.A.v_m > 0 \ \wedge \\ &\quad T.A.v_1 + \ldots + T.A.v_m < S.N[T.\iota].D.v \ \wedge \\ &\quad T.A.c = S.N[T.\iota].D.c \ \wedge \\ &\quad \mathsf{VerifyTxAuth}(N[T.\iota].D.\varphi, T, T.P.s) = 1\end{aligned}$$

That is,

- $T.\iota$ identifies an existing bill,
- the bill is not locked,
- the values to be transferred are all non-zero,
- the sum of the values to be transferred is less than the value of the bill,
- the order contains the correct transaction counter value, and
- the owner proof satisfies the bill's owner predicate.

Actions $\text{Action}_{\text{splitB}}$:

1. for $i = 1, \ldots, m$:
   1.1 $\iota_i \leftarrow \text{NodeID}(\text{bill}, \text{PrndSh}(\text{ExtrUnit}(T.\iota), T.\iota || T.A || T.M_C || i))$, i.e. generate a new bill identifier in the same shard
   1.2 $\text{AddItem}(\iota_i, (T.A.v_i, T.A.\varphi_i, 0, 0))$ – create a new bill ι_i with value v_i and owner predicate φ_i
2. $N[T.\iota].D.v \leftarrow N[T.\iota].D.v - (T.A.v_1 + \ldots + T.A.v_m)$ – reduce the value of the source bill
3. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

Targets: For splitB transaction T, $\text{targets}(T) = \{T.\iota, \iota_1, \iota_2, \ldots, \iota_m\}$.

5.2.4 Lock a Bill

Transaction order $T = \langle \alpha, \beta, \iota, \text{lockB}, A, M_C, P, s_f \rangle$ with $A = (\ell, c)$, $P = (s)$ and $M_C = (T_0, f_m, \iota_f, \rho)$, where:

- $A.\ell \in \mathbb{N}_{64}$ is the new lock status;
- $A.c \in \mathbb{N}_{64}$ is the transaction counter value;
- $P.s \in \mathbb{OCT}^*$ is the owner proof.

Transaction-specific validity condition:

$$
\begin{aligned}
&\psi_{\text{lockB}}(T, S) \equiv \\
&\quad \text{ExtrType}(T.\iota) = \text{bill} \ \wedge \ S.N[T.\iota] \neq \bot \ \wedge \\
&\quad S.N[T.\iota].D.\ell = 0 \ \wedge \\
&\quad T.A.\ell > 0 \ \wedge \\
&\quad T.A.c = S.N[T.\iota].D.c \ \wedge \\
&\quad \text{VerifyTxAuth}(N[T.\iota].D.\varphi, T, T.P.s) = 1
\end{aligned}
$$

That is,

- $T.\iota$ identifies an existing bill,
- the bill is not locked,
- the new status is a "locked" one,
- the order contains the correct transaction counter value, and
- the owner proof satisfies the bill's owner predicate.

Actions $\text{Action}_{\text{lockB}}$:

1. $N[T.\iota].D.\ell \leftarrow T.A.\ell$
2. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

5.2.5 Unlock a Bill

Transaction order $T = \langle \alpha, \beta, \iota, \text{unlockB}, A, M_C, P, s_f \rangle$ with $A = (c)$, $P = (s)$ and $M_C = (T_0, f_m, \iota_f, \rho)$, where:

- $A.c \in \mathbb{N}_{64}$ is the transaction counter value;
- $P.s \in \mathbb{OCT}^*$ is the owner proof.

Transaction-specific validity condition:

$$\begin{aligned}&\psi_{\text{unlockB}}(T, S) \equiv \\ &\quad \text{ExtrType}(T.\iota) = \text{bill} \ \wedge \ S.N[T.\iota] \neq \perp \ \wedge \\ &\quad S.N[T.\iota].D.\ell > 0 \ \wedge \\ &\quad T.A.c = S.N[T.\iota].D.c \ \wedge \\ &\quad \text{VerifyTxAuth}(N[T.\iota].D.\varphi, T, T.P.s) = 1\end{aligned}$$

That is,

- ι identifies an existing bill,
- the bill is locked,
- the order contains the correct transaction counter value, and
- the owner proof satisfies the bill's owner predicate.

Actions $\text{Action}_{\text{unlockB}}$:

1. $N[T.\iota].D.\ell \leftarrow 0$
2. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

5.2.6 Dust Collection

As explained in Sec. 5.1.3, the purpose of the dust collection protocol is to join several smaller-value bills into a single larger-value bill. The process consists of several steps:

1. A target bill is selected to receive the value of the collected dust bills. The target bill may be any existing bill, but it must not be changed by other transactions during the execution of the dust collection protocol. To ensure that, the target bill should be locked using a lockB transaction.
2. The dust bills to be collected are "sent to dust collection" using transDC transactions. To prevent replay attacks, each of the transDC transactions must identify the selected target bill and its current state.
3. The total value of the dust bills is added to the target bill using a swapDC transaction. As this transaction completes the dust collection process, it automatically unlocks the target bill.

5.2.6.1 Transfer to Dust Collector

Transaction order $T = \langle \alpha, \beta, \iota, \text{transDC}, A, M_C, P, s_f \rangle$, with $A = (v, \iota', c', c)$, $P = (s)$ and $M_C = (T_0, f_m, \iota_f, \rho)$, where:

- $A.v \in \mathbb{N}_{64}$ is the target value;
- $A.\iota' \in \mathbb{I}$ identifies the target bill;
- $A.c' \in \mathbb{N}_{64}$ is the transaction counter value for the target bill;
- $A.c \in \mathbb{N}_{64}$ is the transaction counter value for the source bill;
- $P.s \in \mathbb{OCT}^*$ is the owner proof for the soure bill.

Transaction-specific validity condition:

$$\begin{aligned}
&\psi_{\text{transDC}}(T, S) \equiv \\
&\quad \text{ExtrType}(T.\iota) = \text{bill} \ \wedge \ S.N[T.\iota] \neq \bot \ \wedge \\
&\quad S.N[T.\iota].D.\ell = 0 \ \wedge \\
&\quad T.A.v = S.N[T.\iota].D.v \ \wedge \\
&\quad T.A.c = S.N[T.\iota].D.c \ \wedge \\
&\quad \text{VerifyTxAuth}(N[T.\iota].D.\varphi, T, T.P.s) = 1
\end{aligned}$$

That is,

- $T.\iota$ identifies an existing bill,
- the bill is not locked,
- the target value equals the value of the bill,
- the order contains the correct transaction counter value, and
- the owner proof satisfies the bill's owner predicate.

Actions $\text{Action}_{\text{transDC}}$:

1. $N[T.\iota].D.v \leftarrow 0$ – wipe out the bill value
2. $N[T.\iota].D.\varphi \leftarrow \text{DC}$ – mark the bill as collected
3. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$
4. $N[T.\iota_{\text{DC}}].D.v \leftarrow N[T.\iota_{\text{DC}}].D.v + T.A.v$ – increase the DC money supply by bill value

5.2.6.2 Swap with Dust Collector

Transaction order $T = \langle \alpha, \beta, \iota, \text{swapDC}, A, M_C, P, s_f \rangle$, with $A = ((T'_1, \Pi'_1), \ldots, (T'_m, \Pi'_m))$, $P = (s)$ and $M_C = (T_0, f_m, \iota_f, \rho)$, where:

- $A.T'_1, \ldots, A.T'_m \in \mathbb{TR}$ are bill transfer transaction records;
- $A.\Pi'_1, \ldots, A.\Pi'_m \in \mathbb{XP}$ are the transaction execution proofs of $A.T'_1, \ldots, A.T'_m$;
- $s \in \mathbb{OCT}^*$ is the owner proof for the target bill.

Transaction-specific validity condition:

$$
\begin{aligned}
&\psi_{\mathsf{swapDC}}(T, S) \equiv \\
&\quad \mathsf{ExtrType}(T.\iota) = \mathsf{bill} \ \wedge \ S.N[T.\iota] \neq \bot \ \wedge \\
&\quad T'_1.\alpha = \ldots = T'_m.\alpha = T.\alpha \ \wedge \\
&\quad T'_1.\beta = \ldots = T'_m.\beta = T.\beta \ \wedge \\
&\quad T'_1.\iota < \ldots < T'_m.\iota \ \wedge \\
&\quad T'_1.\tau = \ldots = T'_m.\tau = \mathsf{transDC} \ \wedge \\
&\quad T'_1.A.\iota' = \ldots = T'_m.A.\iota' = \iota \ \wedge \\
&\quad T'_1.A.c' = \ldots = T'_m.A.c' = S.N[T.\iota].D.c \ \wedge \\
&\quad \mathsf{VerifyTxProof}((T'_1, \Pi'_1), S.\mathcal{T}, S.\mathcal{PD}[T'_1.\beta]) = 1 \ \wedge \\
&\quad \ldots \ \wedge \\
&\quad \mathsf{VerifyTxProof}((T'_m, \Pi'_m), S.\mathcal{T}, S.\mathcal{PD}[T'_m.\beta]) = 1 \ \wedge \\
&\quad \mathsf{VerifyTxAuth}(N[T.\iota].D.\varphi, T, T.P.s) = 1 \ \wedge \\
&\quad T'_1.A.v + \ldots + T'_m.A.v \leq N[\iota_{\mathsf{DC}}].D.v
\end{aligned}
$$

That is,

- $T.\iota$ identifies an existing bill,
- transfers were in this network,
- transfers were in this partition,
- transfer orders are listed in strictly increasing order of bill identifiers (in particular, this ensures that no source bill can be included multiple times),
- bills were transferred to DC,
- bill transfer orders contain correct target identifiers,
- bill transfer orders contain correct target counter values,
- transaction proofs of the bill transfer orders verify,
- the owner proof satisfies the target bill's owner predicate, and
- there is sufficient DC-money supply.

Actions $\mathsf{Action}_{\mathsf{swapDC}}$:

1. $v \leftarrow T'_1.A.v + \ldots + T'_m.A.v$ – the value to join to target bill
2. $N[T.\iota_{\mathsf{DC}}].D.v \leftarrow N[T.\iota_{\mathsf{DC}}].D.v - v$ – decrease the DC money supply
3. $N[T.\iota].D.v \leftarrow N[T.\iota].D.v + v$ – increase the value of ι
4. $N[T.\iota].D.\ell \leftarrow 0$
5. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

5.2.7 Fee Credit Management

Adding and reclaiming fee credits are multi-step protocols and it's advisable to lock the target unit to prevent failures due to concurrent modifications by other transactions.

More specifically, for adding fee credits:

1. If the target fee credit record exists, it should be locked using a lockFC transaction in the target partition.
2. The amount to be added to fee credits should be paid using a transFC transaction in the money partition. To prevent replay attacks, the transFC transaction must identify the target record and its current state.
3. The transferred value is added to the target record using an addFC transaction in the target partition. As this transaction completes the fee transfer process, it automatically unlocks the target record.

And for reclaiming fee credits:

1. The target bill should be locked using a lockB transaction in the money partition.
2. The fee credit should be closed using a closeFC transaction in the target partition. To prevent replay attacks, the closeFC transaction must identify the target bill and its current state.
3. The reclaimed value is added to the target bill using a reclFC transaction in the money partition. As this transaction completes the fee transfer process, it automatically unlocks the target bill.

5.2.7.1 Transfer to Fee Credit

Transaction order $T = \langle \alpha, \beta, \iota, \text{transFC}, A, M_C, P, s_f \rangle$, with $A = (v, \beta', \iota', c', t', c)$, $P = (s)$ and $M_C = (T_0, f_m, \iota_f, \rho)$, where:

- $A.v \in \mathbb{N}_{64}$ is the amount to transfer;
- $A.\beta' \in \mathbb{P}$ is the target partition identifier;
- $A.\iota' \in \mathbb{I}_{A,\beta'}$ is the target fee credit record identifier;
- $A.c' \in \mathbb{N}_{64} \cup \{\bot\}$ is the the target transaction counter value; for the proof of transfer to be usable in a following addFC operation, this must be set to the current transaction counter value of the target credit record if the record exists, or to $\bot$ if the record does not exist yet;
- $A.t' \in \mathbb{N}_{64}$ is the latest round in which the corresponding "add fee credit" transaction can be executed in the target partition;
- $A.c \in \mathbb{N}_{64}$ is the transaction counter value for the source bill;
- $P.s \in \mathbb{OCT}^*$ is the owner proof for the source bill.

Transaction-specific validity condition:

$$
\begin{aligned}
&\psi_{\text{transFC}}(T, S) \equiv \\
&\quad \text{ExtrType}(T.\iota) = \text{bill} \ \wedge \ S.N[T.\iota] \neq \bot \ \wedge \\
&\quad S.N[T.\iota].D.\ell = 0 \ \wedge \\
&\quad T.A.v \leq S.N[T.\iota].D.v \ \wedge \\
&\quad T.A.c = S.N[T.\iota].D.c \ \wedge \\
&\quad T.M_C.f_m \leq T.A.v \ \wedge \\
&\quad T.M_C.\iota_f = \bot \ \wedge \ s_f = \bot \ \wedge \\
&\quad \text{VerifyTxAuth}(N[T.\iota].D.\varphi, T, T.P.s) = 1
\end{aligned}
$$

That is,

- ι identifies an existing bill,
- the bill is not locked,
- the amount to transfer does not exceed the value of the bill,
- the order contains the correct transaction counter value,
- the transaction fee can't exceed the transferred amount,
- there's no fee credit reference or separate fee authorization proof, and
- the owner proof satisfies the bill's owner predicate.

Actions $\text{Action}_{\text{transFC}}$:

1. $N[T.\iota].D.v \leftarrow N[T.\iota].D.v - T.A.v$
2. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

Note: The transferred credits will be aggregated and added to the target partition's fee bill at the end of the round. The processing fees will be aggregated and added to the money partition's fee bill at the end of the round. Reporting of earned fees and payouts from the partition's fee bill to the validators will be handled in the usual way.

5.2.7.2 Add Fee Credit

Transaction order $T = \langle \alpha, \beta, \iota, \text{addFC}, A, M_C, P, s_f \rangle$, with $A = (\varphi, (T', \Pi'))$, $P = (s)$ and $M_C = (T_0, f_m, \iota_f, \rho)$, where:

- $A.\varphi \in \mathbb{L}$ is the target fee credit record owner predicate;
- $A.T' \in \mathbb{TR}$ is a bill transfer transaction record;
- $A.\Pi' \in \mathbb{XP}$ is the transaction execution proof of $A.T'$;
- $P.s \in \mathbb{OCT}^*$ is the owner proof for the target record.

Transaction-specific validity condition:

$$
\begin{aligned}
&\psi_{\text{addFC}}(T, S) \equiv \\
&\quad \text{ExtrType}(T.\iota) = \text{fcr} \ \wedge \ (\\
&\qquad (S.N[T.\iota] = \bot \ \wedge \ \text{ExtrUnit}(T.\iota) = \text{PrndSh}(\text{ExtrUnit}(T.\iota), T.A.\varphi \| T'.A.t')) \vee \\
&\qquad (S.N[T.\iota] \neq \bot \ \wedge \ T.A.\varphi = S.N[T.\iota].\varphi) \\
&\quad) \wedge \\
&\quad T'.\alpha = T.\alpha \ \wedge \ T'.\beta = \beta_{\text{money}} \ \wedge \ T'.\tau = \text{transFC} \wedge \\
&\quad T'.A.\beta' = T.\beta \ \wedge \ T'.A.\iota' = T.\iota \ \wedge \ (\\
&\qquad (S.N[T.\iota] = \bot \ \wedge \ T'.A.c' = \bot) \vee \\
&\qquad (S.N[T.\iota] \neq \bot \ \wedge \ T'.A.c' = S.N[T.\iota].c) \\
&\quad) \wedge \\
&\quad T'.A.t' \geq S.n \wedge \\
&\quad \text{VerifyTxProof}((T', \Pi'), S.\mathcal{T}, S.\mathcal{PD}[T'.\beta]) = 1 \wedge \\
&\quad T'.M_S.f_a + T.M_C.f_m \leq T'.A.v \wedge \\
&\quad T.M_C.\iota_f = \bot \ \wedge \ s_f = \bot \wedge \\
&\quad \text{VerifyTxAuth}(T.A.\varphi, T, T.P.s) = 1
\end{aligned}
$$

That is,

- $T.\iota$ identifies a fee credit record (either new or pre-existing),
- if the target does not exist, the identifier must agree with the owner predicate,
- if the target exists, the owner predicate must match,
- the bill was transferred to fee credits in this network,
- the bill was transferred to credits for this partition and the target record,
- the bill transfer order contains the correct target counter value;
- the bill transfer is valid to be used in this block,
- proof of the bill transfer order verifies;
- the transaction fee can't exceed the transferred amount,
- there's no fee credit reference or separate fee authorization proof, and
- the owner proof satisfies the bill's owner predicate.

Actions $\text{Action}_{\text{addFC}}$:

1. $v' \leftarrow T'.A.v - T'.M_S.f_a - M_S.f_a$ – the net value of credit
2. if $S.N[T.\iota] = \bot$:
   2.1 AddCredit$(T.\iota, v', T.A.\varphi, T'.A.t')$
3. else:
   3.1 IncrCredit$(T.\iota, v', T'.A.t')$

Note: Reporting of earned fees and payouts from the partition's fee bill to the validators will be handled in the usual way.

5.2.7.3 Close Fee Credit

Transaction order $T = \langle \alpha, \beta, \iota, \text{closeFC}, A, M_C, P, s_f \rangle$ with $A = (v, \iota', c', c)$, $P = (s)$ and $M_C = (T_0, f_m, \iota_f, \rho)$, where:

- $A.v \in \mathbb{N}_{64}$ is the amount to transfer;
- $A.\iota' \in \mathbb{I}_{\text{money}}$ identifies the target bill;
- $A.c' \in \mathbb{N}_{64}$ is the target transaction counter value; for the proof of closure to be usable in a following reclFC operation, this must be set to the current counter value of the target bill;
- $A.c \in \mathbb{N}_{64}$ is the transaction counter value for the source record;
- $P.s \in \mathbb{OCT}^*$ is owner proof for the source record.

Validity Condition

$$
\begin{aligned}
&\psi_{\mathsf{closeFC}}(T, S) \equiv \\
&\quad \mathsf{ExtrType}(T.\iota) = \mathsf{fcr} \ \wedge \ S.N[T.\iota] \neq \bot \ \wedge \\
&\quad S.N[T.\iota].D.\ell = 0 \ \wedge \\
&\quad T.A.v = S.N[T.\iota].b \ \wedge \\
&\quad T.A.c = S.N[T.\iota].c \ \wedge \\
&\quad T.M_C.f_m \leq S.N[T.\iota].b \ \wedge \\
&\quad T.M_C.\iota_f = \bot \ \wedge \ s_f = \bot \ \wedge \\
&\quad \mathsf{VerifyTxAuth}(N[\iota].D.\varphi, T, T.P.s) = 1
\end{aligned}
$$

That is,

- $T.\iota$ identifies an existing fee credit record,
- the record is not locked,
- the amount is the current balance of the record,
- the order contains the correct transaction counter value,
- the transaction fee can't exceed the current balance of the record,
- there's no fee credit reference or separate fee authorization proof, and
- the owner proof satisfies the record's owner predicate.

Actions $\mathsf{Action}_{\mathsf{closeFC}}$:

1. $N[T.\iota].D.v \leftarrow 0$
2. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

Note: Reporting of earned fees and payouts from the partition's fee bill to the validators will be handled in the usual way.

5.2.7.4 Reclaim Fee Credit

Transaction order $T = \langle \alpha, \beta, \iota, \mathsf{reclFC}, A, M_C, P, s_f \rangle$, with $A = ((T', \Pi'))$, $P = (s)$ and $M_C = (T_0, f_m, \iota_f, \rho)$, where:

- $A.T' \in \mathbb{TR}$ is a fee credit closure transaction record;
- $A.\Pi' \in \mathbb{XP}$ is the transaction proof of $A.T'$;
- $P.s \in \mathbb{OCT}^*$ is the owner proof.

Transaction-specific validity condition:

$$
\begin{aligned}
&\psi_{\mathsf{reclFC}}(T, S) \equiv \\
&\quad \mathsf{ExtrType}(T.\iota) = \mathsf{bill} \ \wedge \ S.N[T.\iota] \neq \bot \ \wedge \\
&\quad T'.\alpha = T.\alpha \wedge T'.\tau = \mathsf{closeFC} \ \wedge \\
&\quad T'.A.\iota' = T.\iota \ \wedge \\
&\quad T'.A.c' = S.N[T.\iota].D.c \ \wedge \\
&\quad \mathsf{VerifyTxProof}((T', \Pi'), S.\mathcal{T}, S.\mathcal{PD}[T'.\beta]) = 1 \ \wedge \\
&\quad T'.M_S.f_a + T.M_C.f_m \leq T'.A.v \ \wedge \\
&\quad T.M_C.\iota_f = \bot \ \wedge \ s_f = \bot \ \wedge \\
&\quad \mathsf{VerifyTxAuth}(N[T.\iota].D.\varphi, T, T.P.s) = 1
\end{aligned}
$$

That is,

- $T.\iota$ identifies an existing bill,
- the order is a credit closure in this network,
- the order targets the current bill,
- the order contains the correct target counter value,
- the proof of the credit closure order verifies,
- the transaction fees can't exceed the transferred value,
- there's no fee credit reference or separate fee authorization proof, and
- the owner proof satisfies the bill's owner predicate.

Actions $\mathsf{Action}_{\mathsf{reclFC}}$:

1. $v' \leftarrow T'.A.v - T'.M_S.f_a - M_S.f_a$ – net value reclaimed
2. $N[T.\iota].D.v \leftarrow N[T.\iota].D.v + v'$
3. $N[T.\iota].D.\ell \leftarrow 0$
4. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

Note: The reclaimed credits will be aggregated and removed from the target partition's fee bill at the end of the round. The processing fees will be aggregated and added to the money partition's fee bill at the end of the round. Reporting of earned fees and payouts from the partition's fee bill to the validators will be handled in the usual way.

5.2.7.5 Lock a Fee Credit Record

Transaction order $T = \langle \alpha, \beta, \iota, \mathsf{lockFC}, A, M_C, P, s_f \rangle$ with $A = (\ell, c)$, $P = (s)$ and $M_C = (T_0, f_m, \iota_f, \rho)$, where:

- $A.\ell \in \mathbb{N}_{64}$ is the new lock status;
- $A.c \in \mathbb{N}_{64}$ is transaction counter value;
- $P.s \in \mathbb{OCT}^*$ is the owner proof.

Transaction-specific validity condition:

$$
\begin{aligned}
&\psi_{\mathsf{lockFC}}(T, S) \equiv \\
&\quad \mathsf{ExtrType}(T.\iota) = \mathsf{fcr} \ \wedge \ S.N[T.\iota] \neq \bot \ \wedge \\
&\quad S.N[T.\iota].D.\ell = 0 \ \wedge \\
&\quad T.A.\ell > 0 \ \wedge \\
&\quad T.A.c = S.N[T.\iota].D.c \ \wedge \\
&\quad T.M_C.f_m \leq S.N[T.\iota].D.b \ \wedge \\
&\quad T.M_C.\iota_f = \bot \ \wedge \ s_f = \bot \ \wedge \\
&\quad \mathsf{VerifyTxAuth}(N[T.\iota].D.\varphi, T, T.P.s) = 1
\end{aligned}
$$

That is,

- $T.\iota$ identifies an existing fee credit record,
- the record is not locked,
- the new status is a "locked" one,
- the order contains the correct transaction counter value,
- the transaction fee can't exceed the record balance,
- there's no fee credit reference or separate fee authorization proof, and
- the owner proof satisfies the record's owner predicate.

Actions $\mathsf{Action}_{\mathsf{lockFC}}$:

1. $N[T.\iota].D.b \leftarrow N[T.\iota].D.b - M_S.f_a$
2. $N[T.\iota].D.\ell \leftarrow T.A.\ell$
3. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

Note: Reporting of earned fees and payouts from the partition's fee bill to the validators will be handled in the usual way.

5.2.7.6 Unlock a Fee Credit Record

Transaction order $T = \langle \alpha, \beta, \iota, \mathsf{unlockFC}, A, M_C, P, s_f \rangle$ with $A = (c)$, $P = (s)$ and $M_C = (T_0, f_m, \iota_f, \rho)$, where:

- $A.c \in \mathbb{N}_{64}$ is the transaction counter value;
- $P.s \in \mathbb{OCT}^*$ is the owner proof.

Transaction-specific validity condition:

$$
\begin{aligned}
&\psi_{\mathsf{unlockFC}}(T, S) \equiv \\
&\quad \mathsf{ExtrType}(T.\iota) = \mathsf{fcr} \ \wedge \ S.N[T.\iota] \neq \bot \ \wedge \\
&\quad S.N[T.\iota].D.\ell > 0 \ \wedge \\
&\quad T.A.c = S.N[T.\iota].D.c \ \wedge \\
&\quad T.M_C.f_m \leq S.N[T.\iota].D.b \ \wedge \\
&\quad T.M_C.\iota_f = \bot \ \wedge \ s_f = \bot \ \wedge \\
&\quad \mathsf{VerifyTxAuth}(N[T.\iota].D.\varphi, T, T.P.s) = 1
\end{aligned}
$$

That is,

- $T.\iota$ identifies an existing fee credit record,
- the record is locked,
- the order contains the correct transaction counter value,
- the transaction fee can't exceed the record balance,
- there's no fee credit reference or separate fee authorization proof, and
- the owner proof satisfies the record's owner predicate.

Actions $\text{Action}_{\text{unlockFC}}$:

1. $N[T.\iota].D.b \leftarrow N[T.\iota].D.b - M_S.f_a$
2. $N[T.\iota].D.\ell \leftarrow 0$
3. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

Note: Reporting of earned fees and payouts from the partition's fee bill to the validators will be handled in the usual way.

5.2.8 Round initialization and completion

5.2.8.1 Round Initialization: $\text{RInit}_{\text{money}}$

1. Delete all bills with zero value and expired lifetime:
   1.1 Find all such bills:
   $\mathcal{I} \leftarrow \{\iota : \text{ExtrType}(\iota) = \text{bill} \ \wedge \ N[\iota] \neq \bot \ \wedge \ N[\iota].D.v = 0 \ \wedge \ N[\iota].D.T_{\text{dust}} < S.n\}$
   1.2 For each known partition identifier β:
   $\mathcal{I} \leftarrow \mathcal{I} \backslash \{S.\mathcal{PD}[\beta].\iota_{\text{FC}}\}$
   1.3 For each $\iota \in \mathcal{I}$: DelItem(ι)

5.2.8.2 Round Completion: $\text{RCompl}_{\text{money}}$

1. Perform fee accounting
   1.1 For each known partition identifier β:
      1.1.1 Compute v_+ as the sum of $T.A.v - T.M_S.f_a$ over all transFC records T with $T.A.\beta = \beta$ in the current block
      1.1.2 Compute v_- as the sum of $T.A.T'.A.v - T.A.T'.M_S.f_a$ over all reclFC records T with $T.A.T'.\beta = \beta$ in the current block
      1.1.3 Set $N[S.\mathcal{PD}[\beta].\iota_{\text{FC}}].D.v \leftarrow N[S.\mathcal{PD}[\beta].\iota_{\text{FC}}].D.v + v_+ - v_-$
   1.2 Compute v as the sum of the $M_S.f_a$ fields over all transFC and reclFC records in the current block
   1.3 Set $N[S.\mathcal{PD}[\beta_{\text{money}}].\iota_{\text{FC}}].D.v \leftarrow N[S.\mathcal{PD}[\beta_{\text{money}}].\iota_{\text{FC}}].D.v + v$

6 User-Defined Token Partition Type

6.1 Motivation and General Description

[**TODO**: hierarchical type system]

6.2 Specification

6.2.1 General Parameters

System type identifier: st $= 2$

Partition identifier for the primary public instance: $\beta_{\text{token}} = 2$

Type and unit identifier lengths for the primary public instance: tidlen $= 1$, uidlen $= 32$

Summary value type $\mathbb{V}$: $\mathbb{N}_{64}$

Summary trust base: $\mathcal{V} = v_{\text{total}}$

Summary check: $\gamma(V, v_{\text{total}}) \equiv V = v_{\text{total}}$

Unit types: $\mathcal{U} = \{\text{ftype} = 1, \text{ntype} = 2, \text{ftoken} = 3, \text{ntoken} = 4, \text{fcr} = 16\}$ (fungible and non-fungible token types, fungible and non-fungible tokens, fee credit records).

Unit data $\mathbb{D}_u$ depends on the unit type u as follows:

- $\mathbb{D}_{\text{ntype}} = (sym, nam, ico, \iota_p, \varphi_{\text{s}}, \varphi_{\text{t}}, \varphi_{\text{i}}, \varphi_{\text{d}})$, where
  - sym is the symbol (short name) of this token type, up to 16 B in the UTF-8 encoding; note that the symbols are not guaranteed to be unique;
  - nam is the optional name of this token type, up to 256 B in the UTF-8 encoding; the names are not guaranteed to be unique either (only the type identifiers are);
  - ico is the optional icon for this token type; if given, the icon definition consists of a content type and up to 64 KiB of image data; for compatibility across clients, PNG and SVG are the preferred image formats; for PNG, the content type should be '`image/png`'; for SVG, the UTF-8 text encoding should be used and the content type should be '`image/svg+xml`' for plain SVG and '`image/svg+xml; encoding=gzip`' for compressed SVG;
  - $\iota_p \in \mathbb{I} \cup \{\perp\}$ identifies the parent type that this type derives from; $\iota_p = \perp$ indicates there is no parent type;
  - $\varphi_{\text{s}} \in \mathbb{L}$ is the predicate that controls defining new subtypes of this type;
  - $\varphi_{\text{t}} \in \mathbb{L}$ is the predicate that controls minting new tokens of this type;

  - $\varphi_{\mathsf{i}} \in \mathbb{L}$ is the clause that all tokens of this type (and of its subtypes) inherit into their owner predicates;
  - $\varphi_{\mathsf{d}} \in \mathbb{L}$ is the clause that all tokens of this type (and of its subtypes) inherit into their data update predicates;
- $\mathbb{D}_{\mathsf{ntoken}} = (\iota_t, nam, uri, dat, \varphi, \varphi_{\mathsf{d}}, \ell, c)$, where
  - $\iota_t \in \mathbb{I}$ identifies the type of this token;
  - nam is the optional name of this token, up to 256 B in the UTF-8 encoding; the purpose of token names is to identify individual tokens within a collection;
  - uri is the optional URI of an external resource associated with this token; if given, this must comply with RFC 3986, assuming the UTF-8 text encoding; the size of this field is not allowed to exceed 4 KiB;
  - dat is the optional data associated with this token; this can be any data type supported by the Alphabill platform, including a structure whose fields can in turn be of any supported data type; the only restriction enforced by the platform is that the size of this field is not allowed to exceed 64 KiB;
  - $\varphi \in \mathbb{L}$ is the current owner predicate of this token;
  - $\varphi_{\mathsf{d}} \in \mathbb{L}$ is the predicate that controls the updates to the data field;
  - $\ell \in \mathbb{N}_{64}$ is the lock status of the token; allows locking of the token at the beginning of a multi-step protocol that needs the token to remain unmodified by other transactions during the protocol execution; $\ell = 0$ means the token is not locked, any other value means it's locked;
  - $c \in \mathbb{N}_{64}$ is the transaction counter for this token;
- $\mathbb{D}_{\mathsf{ftype}} = (sym, nam, ico, \iota_p, dec, \varphi_{\mathsf{s}}, \varphi_{\mathsf{t}}, \varphi_{\mathsf{i}})$, where
  - sym is the symbol (short name) of this token type, up to 16 B in the UTF-8 encoding; note that the symbols are not guaranteed to be unique;
  - nam is the optional name of this token type, up to 256 B in the UTF-8 encoding; the names are not guaranteed to be unique either (only the type identifiers are);
  - ico is the optional icon for this token type; if given, the icon definition consists of a content type and up to 64 KiB of image data; for compatibility across clients, PNG and SVG are the preferred image formats; for PNG, the content type should be '`image/png`'; for SVG, the UTF-8 text encoding should be used and the content type should be '`image/svg+xml`' for plain SVG and '`image/svg+xml; encoding=gzip`' for compressed SVG;
  - $\iota_p \in \mathbb{I} \cup \{\bot\}$ identifies the parent type that this type derives from; $\iota_p = \bot$ indicates there is no parent type;
  - $dec \in \mathbb{N}_8$, $0 \leq dec \leq 8$ is the number of decimal places to display for values of tokens of this type;
  - $\varphi_{\mathsf{s}} \in \mathbb{L}$ is the predicate that controls defining new subtypes of this type;
  - $\varphi_{\mathsf{t}} \in \mathbb{L}$ is the predicate that controls minting new tokens of this type;
  - $\varphi_{\mathsf{i}} \in \mathbb{L}$ is the clause that all tokens of this type (and of its subtypes) inherit into their owner predicates;
- $\mathbb{D}_{\mathsf{ftoken}} = (\iota_t, v, \varphi, \ell, c, t)$, where
  - $\iota_t \in \mathbb{I}$ is the type of this token;

  - $v \in \mathbb{N}_{64}$ is the value of this token;
  - $\varphi \in \mathbb{L}$ is the current owner predicate of this token;
  - $\ell \in \mathbb{N}_{64}$ is the lock status of the token; allows locking of the token at the beginning of a multi-step protocol that needs the token to remain unmodified by other transactions during the protocol execution; $\ell = 0$ means the token is not locked, any other value means it's locked;
  - $c \in \mathbb{N}_{64}$ is the transaction counter for this token;
  - $t \in \mathbb{N}_{64}$ is the minimum lifetime of this token;
- $\mathbb{D}_{\text{fcr}} = (b, \varphi, \ell, c, t)$, where
  - $b \in \mathbb{N}_{64}$ is the current balance of this record, in fixed point format with 8 fractional decimal digits;
  - $\varphi \in \mathbb{L}$ is the owner predicate of this record;
  - $\ell \in \mathbb{N}_{64}$ is the lock status of the record; $\ell = 0$ means the record is not locked, any other value means it's locked;
  - $c \in \mathbb{N}_{64}$ is transaction counter;
  - $t \in \mathbb{N}_{64}$ is the minimum lifetime of this record.

Summary functions:

- $V_s(D) = D.b$ for $\mathbb{D}_{\text{fcr}}$, or 0 otherwise
- $F_S(v, v_L, v_R) = v + v_L + v_R$
- $F_S(\perp, v_L, v_R) = v_L + v_R$

Transaction types: $\mathbb{T}$ = {defFT = 1, defNT = 2, mintFT = 3, mintNT = 4, transFT = 5, transNT = 6, lockT = 7, unlockT = 8, splitFT = 9, burnFT = 10, joinFT = 11, updateNT = 12, addFC = 16, closeFC = 17, lockFC = 18, unlockFC = 19, setFC = 20, delFC = 21} (define a fungible/non-fungible token type, mint a fungible/non-fungible token, transfer a fungible/non-fungible token, lock/unlock a token, split a fungible token, burn a fungible token, join fungible tokens, update a non-fungible token, add fee credit, close fee credit, lock/unlock a fee credit record, set fee credit, delete fee credit)

6.2.1.1 Notation

The transaction validity conditions in the following sections include evaluating multipart predicates on multipart inputs. We define the result of evaluating the multipart predicate $\pi = (\pi_1, \ldots, \pi_n)$ on the transaction order T and the multipart input $s = (s_1, \ldots, s_m)$ as follows:

$$\pi(T, s) = (n = m) \wedge \text{VerifyTxAuth}(\pi_1, T, s_1) \wedge \ldots \wedge \text{VerifyTxAuth}(\pi_n, T, s_n).$$

6.2.2 Define a Fungible Token Type

Transaction order $T = \langle \alpha, \beta, \iota, \text{defFT}, A, M_C, P, s_f \rangle$ with $A = (sym, nam, ico, \iota_p, dec, \varphi_{\text{s}}, \varphi_{\text{t}}, \varphi_{\text{i}})$, $P = (s_s)$, where

- $A.sym$ is the short name of the new token type;
- $A.nam$ is the optional full name of the new token type;
- $A.ico$ is the optional icon of the new token type, given as a pair (typ, dat), where

  - typ is the MIME content type identifying an image format, given as a string of up to 64 B in the UTF-8 encoding;
  - dat is a byte string up to 64 KiB in size, representing an image in the format specified by typ;
- $A.\iota_p \in \mathbb{I} \cup \{\bot\}$ identifies the parent type that the new type derives from;
- $A.dec \in \mathbb{N}_8$, $0 \leq A.dec \leq 8$ is the number of decimal places to display for values of tokens of the new type;
- $A.\varphi_\mathsf{s} \in \mathbb{L}$ is the predicate that controls defining subtypes of the new type;
- $A.\varphi_\mathsf{t} \in \mathbb{L}$ is the predicate that controls minting tokens of the new type;
- $A.\varphi_\mathsf{i} \in \mathbb{L}$ is the clause that all tokens of the new type (and of its subtypes) inherit into their owner predicates;
- $P.s_s \in (\mathbb{OCT}^*)^*$ is the input to satisfy the subtype predicates of the parent types.

Validity Condition

$$\begin{aligned}
&\Psi_{\mathsf{defFT}}(T, S) \equiv \\
&\quad \mathsf{ExtrType}(T.\iota) = \mathsf{ftype} \ \wedge \ S.N[T.\iota] = \bot \ \wedge \\
&\quad (T.A.\iota_p = \bot \ \vee \ \mathsf{ExtrType}(T.A.\iota_p) = \mathsf{ftype} \ \wedge \ S.N[T.A.\iota_p] \neq \bot) \ \wedge \\
&\quad (T.A.\iota_p = \bot \ \vee \ S.N[T.A.\iota_p].D.dec = T.A.dec) \ \wedge \\
&\quad \Pi(T.A.\iota_p)(T, T.P.s_s) = 1,
\end{aligned}$$

where

$$\Pi(\iota) = \begin{cases} () & \text{if } \iota = \bot \\ \Pi(S.N[\iota].D.\iota_p) \| S.N[\iota].D.\varphi_\mathsf{s} & \text{otherwise.} \end{cases}$$

That is,

- $T.\iota$ identifies a fungible token type that does not yet exist,
- the new type either has no parent or the parent is an existing fungible token type,
- the new type either has no parent or displays the token values with the same number of decimal places as the parent, and
- the input s_s satisfies the multipart predicate obtained by joining all the subtype predicates along the type inheritance chain.

Actions

1. AddItem$(T.\iota, 1, (T.A.sym, T.A.nam, T.A.ico, T.A.\iota_p, T.A.dec, T.A.\varphi_\mathsf{s}, T.A.\varphi_\mathsf{t}, T.A.\varphi_\mathsf{i}))$

6.2.3 Define a Non-Fungible Token Type

Transaction order $T = \langle \alpha, \beta, \iota, \mathsf{defNT}, A, M_C, P, s_f \rangle$ with $A = (sym, nam, ico, \iota_p, \varphi_\mathsf{s}, \varphi_\mathsf{t}, \varphi_\mathsf{i}, \varphi_\mathsf{d})$, $P = (s_s)$, where

- $A.sym$ is the short name of the new token type;
- $A.nam$ is the optional full name of the new token type;

- $A.ico$ is the optional icon of the new token type, given as a pair (typ, dat), where
  - typ is the MIME content type identifying an image format, given as a string of up to 64 B in the UTF-8 encoding;
  - dat is a byte string up to 64 KiB in size, representing an image in the format specified by typ;
- $A.\iota_p \in \mathbb{I}$ identifies the parent type that the new type derives from;
- $A.\varphi_\mathsf{s} \in \mathbb{L}$ is the predicate that controls defining subtypes of the new type;
- $A.\varphi_\mathsf{t} \in \mathbb{L}$ is the predicate that controls minting tokens of the new type;
- $A.\varphi_\mathsf{i} \in \mathbb{L}$ is the clause that all tokens of the new type (and of its subtypes) inherit into their owner predicates;
- $A.\varphi_\mathsf{d} \in \mathbb{L}$ is the clause that all tokens of the new type (and of its subtypes) inherit into their data update predicates;
- $P.s_s \in (\mathbb{OCT}^*)^*$ is the input to satisfy the subtype predicates of the parent types.

Validity Condition

$$\begin{aligned}&\Psi_{\mathsf{defNT}}(T, S) \equiv \\ &\quad \mathsf{ExtrType}(T.\iota) = \mathsf{ntype} \ \wedge \ S.N[T.\iota] = \bot \wedge \\ &\quad (T.A.\iota_p = \bot \ \vee \ \mathsf{ExtrType}(T.A.\iota_p) = \mathsf{ntype} \ \wedge \ S.N[T.A.\iota_p] \neq \bot) \wedge \\ &\quad \Pi(T.A.\iota_p)(T, T.P.s_s) = 1,\end{aligned}$$

where

$$\Pi(\iota) = \begin{cases} () & \text{if } \iota = \bot \\ \Pi(S.N[\iota].D.\iota_p) \| S.N[\iota].D.\varphi_\mathsf{s} & \text{otherwise.} \end{cases}$$

That is,

- $T.\iota$ identifies a non-fungible token type that does not yet exist,
- the new type either has no parent or the parent is an existing non-fungible token type, and
- the input s_s satisfies the multipart predicate obtained by joining all the subtype predicates along the type inheritance chain.

Actions

1. AddItem$(T.\iota, 1, (T.A.sym, T.A.nam, T.A.ico, T.A.\iota_p, T.A.\varphi_\mathsf{s}, T.A.\varphi_\mathsf{t}, T.A.\varphi_\mathsf{i}, T.A.\varphi_\mathsf{d}))$

6.2.4 Mint a Fungible Token

Transaction order $T = \langle \alpha, \beta, \iota, \mathsf{mintFT}, A, M_C, P, s_f \rangle$ with $A = (\iota_t, v, \varphi, \eta)$, $P = (s_t)$, where

- $A.\iota_t \in \mathbb{I}$ identifies the type of the new token;
- $A.v \in \mathbb{N}_{64}$ is the value of the new token;
- $A.\varphi \in \mathbb{L}$ is the initial owner predicate of the new token;
- $A.\eta \in \mathbb{N}_{64} \cup \{\bot\}$ is an optional nonce (could be used to ensure multiple tokens with otherwise identical attibutes get unique identifiers);
- $P.s_t \in \mathbb{OCT}^*$ is the input to satisfy the token minting predicate of the type.

Validity Condition

$$\begin{aligned}&\Psi_{\mathsf{mintFT}}(T, S) \equiv \\ &\quad \mathsf{ExtrType}(T.\iota) = \mathsf{ftoken} \ \wedge \ S.N[T.\iota] = \bot \ \wedge \\ &\quad \mathsf{ExtrUnit}(T.\iota) = \mathsf{PrndSh}(\mathsf{ExtrUnit}(T.A.\iota_t), T.A \| T.M_C)) \ \wedge \\ &\quad \mathsf{ExtrType}(T.A.\iota_t) = \mathsf{ftype} \ \wedge \ S.N[T.A.\iota_t] \neq \bot \ \wedge \\ &\quad T.A.v > 0 \ \wedge \\ &\quad \mathsf{VerifyTxAuth}(S.N[T.\iota].D.\varphi_{\mathsf{t}}, T, T.P.s_t) = 1.\end{aligned}$$

That is,

- $T.\iota$ identifies a fungible token that does not yet exist,
- the identifier of the new token has been correctly generated,
- the type of the new token is an existing fungible token type,
- the new token has non-zero value, and
- the input s_t satisfies the token minting predicate of the type.

Actions

1. $\mathsf{AddItem}(T.\iota, (T.A.\iota_t, T.A.v, T.A.\varphi, 0, 0, T.M_C.T_0))$

6.2.5 Mint a Non-Fungible Token

Transaction order $T = \langle \alpha, \beta, \iota, \mathsf{mintNT}, A, M_C), P, s_f \rangle$ with $A = (\iota_t, nam, uri, dat, \varphi, \varphi_{\mathsf{d}}, \eta)$, $P = (s_t)$, where

- $A.\iota_t \in \mathbb{I}$ identifies the type of the new token;
- $A.nam$ is the optional name of the new token;
- $A.uri$ is the optional URI of an external resource associated with the new token;
- $A.dat$ is the optional data associated with the new token;
- $A.\varphi \in \mathbb{L}$ is the initial owner predicate of the new token;
- $A.\varphi_{\mathsf{d}} \in \mathbb{L}$ is the data update predicate of the new token;
- $A.\eta \in \mathbb{N}_{64} \cup \{\bot\}$ is an optional nonce (could be used to ensure multiple tokens with otherwise identical attibutes get unique identifiers);
- $P.s_t \in \mathbb{OCT}^*$ is the input to satisfy the token minting predicate of the type.

Validity Condition

$$\begin{aligned}&\Psi_{\mathsf{mintNT}}(P, S) \equiv \\ &\quad \mathsf{ExtrType}(T.\iota) = \mathsf{ntoken} \ \wedge \ S.N[T.\iota] = \bot \ \wedge \\ &\quad \mathsf{ExtrUnit}(T.\iota) = \mathsf{PrndSh}(\mathsf{ExtrUnit}(T.A.\iota_t), T.A \| T.M_C)) \ \wedge \\ &\quad \mathsf{ExtrType}(T.A.\iota_t) = \mathsf{ntype} \ \wedge \ S.N[T.A.\iota_t] \neq \bot \ \wedge \\ &\quad \mathsf{VerifyTxAuth}(S.N[T.\iota].D.\varphi_{\mathsf{t}}, T, T.P.s_t) = 1.\end{aligned}$$

That is,

- $T.\iota$ identifies a non-fungible token that does not yet exist,
- the identifier of the new token has been correctly generated,
- the type of the new token is an existing non-fungible token type, and
- the input s_t satisfies the token minting predicate of the type.

Actions

1. AddItem$(T.\iota, (T.A.\iota_t, T.A.nam, T.A.uri, T.A.dat, T.A.\varphi, T.A.\varphi_{\mathsf{d}}, 0, 0))$

6.2.6 Transfer a Fungible Token

Transaction order $T = \langle \alpha, \beta, \iota, \mathsf{transFT}, A, M_C, P, s_f \rangle$ with $A = (\iota_t, v, \varphi, c)$, $P = (s, s_i)$, where

- $A.\iota_t \in \mathbb{I}$ identifies the type of the token;
- $A.v \in \mathbb{N}_{64}$ is the value to transfer;
- $A.\varphi \in \mathbb{L}$ is the new owner predicate of the token;
- $A.c \in \mathbb{N}_{64}$ is the transaction counter;
- $P.s \in \mathbb{OCT}^*$ is the input to satisfy the current owner predicate of the token;
- $P.s_i \in (\mathbb{OCT}^*)^*$ is the input to satisfy the owner predicates inherited from the types.

Validity Condition

$$
\begin{aligned}
&\Psi_{\mathsf{transFT}}(T, S) \equiv \\
&\quad \mathsf{ExtrType}(T.\iota) = \mathsf{ftoken} \ \wedge S.N[T.\iota] \neq \bot \wedge \\
&\quad S.N[T.\iota].D.\ell = 0 \wedge \\
&\quad T.A.\iota_t = S.N[T.\iota].D.\iota_t \wedge \\
&\quad T.A.v = S.N[T.\iota].D.v \wedge \\
&\quad T.A.c = S.N[T.\iota].D.c \wedge \\
&\quad \mathsf{VerifyTxAuth}(S.N[T.\iota].D.\varphi, T, T.P.s) = 1 \wedge \\
&\quad \Pi(S.N[T.\iota].D.\iota_t)(T, T.P.s_i) = 1,
\end{aligned}
$$

where

$$
\Pi(\iota) = \begin{cases} () & \text{if } \iota = \bot \\ \Pi(S.N[\iota].D.\iota_p) || S.N[\iota].D.\varphi_{\mathsf{i}} & \text{otherwise.} \end{cases}
$$

That is,

- $T.\iota$ identifies an existing fungible token,
- the token is not locked,
- the token type in the transaction order matches the actual token type,
- the value transferred is the value of the token,
- the order contains the correct transaction counter value,
- the input s satisfies the token's current owner predicate, and
- the input s_i satisfies the multipart predicate obtained by joining all the inherited owner predicate clauses along the type inheritance chain.

Actions

1. $N[T.\iota].D.\varphi \leftarrow T.A.\varphi$
2. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

6.2.7 Transfer a Non-Fungible Token

Transaction order $T = \langle \alpha, \beta, \iota, \mathsf{transNT}, A, M_C, P, s_f \langle$ with $A = (\iota_t, \varphi, c)$, $P = (s, s_i)$, where

- $A.\iota_t \in \mathbb{I}$ identifies the type of the token;
- $A.\varphi \in \mathbb{L}$ is the new owner predicate of the token;
- $A.c \in \mathbb{N}_{64}$ is the transaction counter;
- $P.s \in \mathbb{OCT}^*$ is the input to satisfy the current owner predicate of the token;
- $P.s_i \in (\mathbb{OCT}^*)^*$ is the input to satisfy the owner predicates inherited from the types.

Validity Condition

$$
\begin{aligned}
&\Psi_{\mathsf{transNT}}(T, S) \equiv \\
&\quad \mathsf{ExtrType}(T.\iota) = \mathsf{ntoken} \ \wedge S.N[T.\iota] \neq \bot \ \wedge \\
&\quad S.N[T.\iota].D.\ell = 0 \ \wedge \\
&\quad T.A.\iota_t = S.N[T.\iota].D.\iota_t \ \wedge \\
&\quad T.A.c = S.N[T.\iota].D.c \ \wedge \\
&\quad \mathsf{VerifyTxAuth}(S.N[T.\iota].D.\varphi, T, T.P.s) = 1 \ \wedge \\
&\quad \Pi(S.N[T.\iota].D.\iota_t)(T, T.P.s_i) = 1,
\end{aligned}
$$

where

$$
\Pi(\iota) = \begin{cases} () & \text{if } \iota = \bot \\ \Pi(S.N[\iota].D.\iota_p) \| S.N[\iota].D.\varphi_{\mathsf{i}} & \text{otherwise.} \end{cases}
$$

That is,

- $T.\iota$ identifies an existing non-fungible token,
- the token is not locked,
- the token type in the transaction order matches the actual token type,
- the order contains the correct transaction counter value,
- the input s satisfies the token's current owner predicate, and
- the input s_i satisfies the multipart predicate obtained by joining all the inherited owner predicate clauses along the type inheritance chain.

Actions

1. $N[T.\iota].D.\varphi \leftarrow T.A.\varphi$
2. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

6.2.8 Lock a Token

Transaction order $T = \langle \alpha, \beta, \iota, \mathsf{lockT}, A, M_C, P, s_f \rangle$ with $A = (\ell, c)$, $P = (s)$, where

- $A.\ell \in \mathbb{N}_{64}$ is the new lock status of the token;
- $A.c \in \mathbb{N}_{64}$ is the transaction counter;
- $P.s \in \mathbb{OCT}^*$ is the input to satisfy the owner predicate of the token.

Validity Condition

$$
\begin{aligned}
&\Psi_{\mathsf{lockT}}(T, S) \equiv \\
&\quad (\mathsf{ExtrType}(T.\iota) = \mathsf{ftoken} \ \vee \ \mathsf{ExtrType}(T.\iota) = \mathsf{ntoken}) \ \wedge \ S.N[T.\iota] \neq \bot \ \wedge \\
&\quad S.N[T.\iota].D.\ell = 0 \ \wedge \\
&\quad T.A.\ell > 0 \ \wedge \\
&\quad T.A.c = S.N[T.\iota].D.c \ \wedge \\
&\quad \mathsf{VerifyTxAuth}(S.N[T.\iota].D.\varphi, T, T.P.s) = 1.
\end{aligned}
$$

That is,

- $T.\iota$ identifies an existing fungible or non-fungible token,
- the token is not locked,
- the new status is a “locked” one,
- the order contains the correct transaction counter value, and
- the input s satisfies the token’s owner predicate.

Actions

1. $N[T.\iota].D.\ell \leftarrow T.A.\ell$
2. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

6.2.9 Unlock a Token

Transaction order $T = \langle \alpha, \beta, \iota, \mathsf{unlockT}, A, M_C, P, s_f \rangle$ with $A = (c)$, $P = (s)$, where

- $A.c \in \mathbb{N}_{64}$ is the transaction counter;
- $P.s \in \mathbb{OCT}^*$ is the input to satisfy the owner predicate of the token.

Validity Condition

$$
\begin{aligned}
&\Psi_{\mathsf{unlockT}}(T, S) \equiv \\
&\quad (\mathsf{ExtrType}(T.\iota) = \mathsf{ftoken} \ \vee \ \mathsf{ExtrType}(T.\iota) = \mathsf{ntoken}) \ \wedge \ S.N[T.\iota] \neq \bot \ \wedge \\
&\quad S.N[T.\iota].D.\ell > 0 \ \wedge \\
&\quad T.A.c = S.N[T.\iota].D.c \ \wedge \\
&\quad \mathsf{VerifyTxAuth}(S.N[T.\iota].D.\varphi, T, T.P.s) = 1.
\end{aligned}
$$

That is,

- $T.\iota$ identifies an existing fungible or non-fungible token,
- the token is locked,
- the order contains the correct transaction counter value, and
- the input s satisfies the token's owner predicate.

Actions

1. $N[T.\iota].D.\ell \leftarrow 0$
2. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

6.2.10 Split a Fungible Token

Transaction order $T = \langle \alpha, \beta, \iota, \text{splitFT}, A, M_C, P, s_f \rangle$ with $A = (\iota_t, v, \varphi, c)$, $P = (s, s_i)$, where

- $A.\iota_t \in \mathbb{I}$ identifies the type of the token;
- $A.v \in \mathbb{N}_{64}$ is the amount to transfer;
- $A.\varphi \in \mathbb{L}$ is the initial owner predicate of the new token;
- $A.c \in \mathbb{N}_{64}$ is the transaction counter;
- $P.s \in \mathbb{OCT}^*$ is the input to satisfy the owner predicate of the source token;
- $P.s_i \in (\mathbb{OCT}^*)^*$ is the input to satisfy the owner predicates inherited from the types.

Validity Condition

$$
\begin{aligned}
&\Psi_{\text{splitFT}}(T, S) \equiv \\
&\quad \text{ExtrType}(T.\iota) = \text{ftoken} \ \wedge S.N[T.\iota] \neq \bot \wedge \\
&\quad S.N[T.\iota].D.\ell = 0 \wedge \\
&\quad T.A.\iota_t = S.N[T.\iota].D.\iota_t \wedge \\
&\quad T.A.v > 0 \wedge \\
&\quad T.A.v < S.N[T.\iota].D.v \wedge \\
&\quad T.A.c = S.N[T.\iota].D.c \wedge \\
&\quad \text{VerifyTxAuth}(S.N[T.\iota].D.\varphi, T, T.P.s) = 1 \wedge \\
&\quad \Pi(S.N[T.\iota].D.\iota_t)(T, T.P.s_i) = 1,
\end{aligned}
$$

where

$$
\Pi(\iota) = \begin{cases} () & \text{if } \iota = \bot \\ \Pi(S.N[\iota].D.\iota_p) \| S.N[\iota].D.\varphi_{\text{i}} & \text{otherwise.} \end{cases}
$$

That is,

- $T.\iota$ identifies an existing fungible token,
- the token is not locked,
- the token type in the transaction order matches the actual token type,
- the value to be transferred is non-zero,
- the value to be transferred is less than the value of the source token,

- the order contains the correct transaction counter value,
- the input s satisfies the token's current owner predicate, and
- the input s_i satisfies the multipart predicate obtained by joining all the inherited owner predicate clauses along the type inheritance chain.

Actions

1. $\iota' \leftarrow \text{NodeID}(\text{ftoken}, \text{PrndSh}(\text{ExtrUnit}(T.\iota), T.\iota \| T.A \| T.M_C))$
2. $\text{AddItem}(\iota', (T.A.D.\iota_t, T.A.v, T.A.\varphi, 0, 0, 0))$
3. $N[T.\iota].D.v \leftarrow N[T.\iota].D.v - T.A.v$
4. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

Targets

For splitFT transaction T, $\text{targets}(T) = \{T.\iota, \iota'\}$.

6.2.11 Join Fungible Tokens

Joining of fungible tokens collects the value represented by several tokens of the same type into one token. The process consists of several steps:

1. A target token is selected to receive the value of the joined tokens. The target token may be any existing token, but it must not be changed by other transactions during the execution of the joining protocol. To ensure that, the target token should be locked using a lockT transaction.
2. The source tokens are "burned" (deleted) using burnFT transactions. To prevent replay attacks, each of the burnFT transactions must identify the selected target token and its current state.
3. The value of the source tokens is added to the target token using a joinFT transaction. As this transaction completes the joining process, it automatically unlocks the target token.

6.2.11.1 Burning Step

Transaction order $T = \langle \alpha, \beta, \iota, \text{burnFT}, A, M_C, P, s_f \rangle$ with $A = (\iota_t, v, \iota', c', c)$, $P = (s, s_i)$, where

- $A.\iota_t \in \mathbb{I}$ identifies the type of the token to burn;
- $A.v \in \mathbb{N}_{64}$ is the value to burn; note that for the proof of burn to be usable in a following joinFT operation, the resulting value of the target token must not overflow $\mathbb{N}_{64}$;
- $A.\iota' \in \mathbb{I}$ is the identifier of the target token;
- $A.c' \in \mathbb{N}_{64}$ is the transaction counter of the target token;
- $A.c \in \mathbb{N}_{64}$ is the transaction counter of the source token;
- $P.s \in \mathbb{OCT}^*$ is the input to satisfy the owner predicate of the source token;
- $P.s_i \in (\mathbb{OCT}^*)^*$ is the input to satisfy the owner predicates inherited from the types.

Validity Condition

$$\begin{aligned}
&\Psi_{\mathsf{burnFT}}(T, S) \equiv \\
&\quad \mathsf{ExtrType}(T.\iota) = \mathsf{ftoken} \ \wedge S.N[T.\iota] \neq \bot \ \wedge \\
&\quad S.N[T.\iota].D.\ell = 0 \ \wedge \\
&\quad T.A.\iota_t = S.N[T.\iota].D.\iota_t \ \wedge \\
&\quad T.A.v = S.N[T.\iota].D.v \ \wedge \\
&\quad T.A.c = S.N[T.\iota].D.c \ \wedge \\
&\quad \mathsf{VerifyTxAuth}(S.N[T.\iota].D.\varphi, T, T.P.s) = 1 \ \wedge \\
&\quad \Pi(S.N[T.\iota].D.\iota_t)(T, T.P.s_i) = 1,
\end{aligned}$$

where

$$\Pi(\iota) = \begin{cases} () & \text{if } \iota = \bot \\ \Pi(S.N[\iota].D.\iota_p) \| S.N[\iota].D.\varphi_{\mathsf{i}} & \text{otherwise.} \end{cases}$$

That is,

- $T.\iota$ identifies an existing fungible token,
- the token is not locked,
- the type of token to burn matches the actual type of the token,
- the value to be burned is the value of the token,
- the order contains the correct transaction counter value,
- the input s satisfies the token's current owner predicate, and
- the input s_i satisfies the multipart predicate obtained by joining all the inherited owner predicate clauses along the type inheritance chain.

Actions

1. $N[T.\iota].D.v \leftarrow 0$
2. $N[T.\iota].D.\varphi \leftarrow 0$
3. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

6.2.11.2 Joining Step

Transaction order $T = \langle \alpha, \beta, \iota, \mathsf{joinFT}, A, M_C, P, s_f \rangle$ with $A = ((T'_1, \Pi'_1), \ldots, (T'_m, \Pi'_m))$, $P = (s, s_i)$, where

- $A.T'_1, \ldots, A.T'_m \in \mathbb{TR}$ are records of the transactions that burned the source tokens;
- $A.\Pi'_1, \ldots, A.\Pi'_m \in \mathbb{XP}$ are the transaction execution proofs of $A.T'_1, \ldots, A.T'_m$;
- $P.s \in \mathbb{OCT}^*$ is the input to satisfy the owner predicate of the target token;
- $P.s_i \in (\mathbb{OCT}^*)^*$ is the input to satisfy the owner predicates inherited from the types.

Validity Condition

$$
\begin{aligned}
&\Psi_{\mathsf{joinFT}}(T, S) \equiv \\
&\quad \mathsf{ExtrType}(T.\iota) = \mathsf{ftoken} \ \wedge S.N[T.\iota] \neq \bot \ \wedge \\
&\quad T.A.T'_1.\alpha = \ldots = T.A.T'_m.\alpha = T.\alpha \ \wedge \\
&\quad T.A.T'_1.\beta = \ldots = T.A.T'_m.\beta = T.\beta \ \wedge \\
&\quad T.A.T'_1.\iota < \ldots < T.A.T'_m.\iota \ \wedge \\
&\quad T.A.T'_1.\tau = \ldots = T.A.T'_m.\tau = \mathsf{burnFT} \ \wedge \\
&\quad T.A.T'_1.A.\iota_t = \ldots = T.A.T'_m.A.\iota_t = S.N[T.\iota].D.\iota_t \ \wedge \\
&\quad T.A.T'_1.A.\iota' = \ldots = T.A.T'_m.A.\iota' = T.\iota \ \wedge \\
&\quad T.A.T'_1.A.c' = \ldots = T.A.T'_m.A.c' = S.N[T.\iota].D.c \ \wedge \\
&\quad \mathsf{VerifyTxProof}(T.A.\Pi'_1, T.A.T'_1, S.\mathcal{T}, S.\mathcal{PD}[T.A.T'_1.\alpha]) \ \wedge \\
&\quad \ldots \ \wedge \\
&\quad \mathsf{VerifyTxProof}(T.A.\Pi'_m, T.A.T'_m, S.\mathcal{T}, S.\mathcal{PD}[T.A.T'_m.\alpha]) \ \wedge \\
&\quad S.N[T.\iota].D.v + T.A.T'_1.A.v + \ldots + T.A.T'_m.A.v < 2^{64} \ \wedge \\
&\quad \mathsf{VerifyTxAuth}(S.N[T.\iota].D.\varphi, T, T.P.s) = 1 \ \wedge \\
&\quad \Pi(S.N[T.\iota].D.\iota_t)(T, T.P.s_i) = 1,
\end{aligned}
$$

where

$$
\Pi(\iota) = \begin{cases} () & \text{if } \bot \\ \Pi(S.N[\iota].D.\iota_p) || S.N[\iota].D.\varphi_{\mathsf{i}} & \text{otherwise.} \end{cases}
$$

That is,

- $T.\iota$ identifies an existing fungible token,
- the transactions $T'_1, \ldots, T'_m$ were in this network,
- the transactions $T'_1, \ldots, T'_m$ were in this partition,
- burning transaction orders are listed in strictly increasing order of token identifiers (in particular, this ensures that no source token can be included multiple times),
- the transactions $T'_1, \ldots, T'_m$ were burning transactions,
- the types of the burned source tokens match the type of target token,
- the source tokens were burned to join them to the target token,
- the burning transactions contain correct target transaction counter values,
- the burning transactions were valid transactions,
- the value of the joined token would not overflow $\mathbb{N}_{64}$,
- the input s satisfies the token's current owner predicate, and
- the input s_i satisfies the multipart predicate obtained by joining all the inherited owner predicate clauses along the type inheritance chain.

Actions

1. $N[T.\iota].D.v \leftarrow N[T.\iota].D.v + T.A.T'_1.A.v + \ldots + T.A.T'_m.A.v$
2. $N[T.\iota].D.\ell \leftarrow 0$
3. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

6.2.12 Update a Non-Fungible Token

Transaction order $T = \langle \alpha, \beta, \iota, \text{updateNT}, A, M_C, P, s_f \rangle$ with $A = (dat, c)$, $P = (s, s_i)$, where

- $A.dat$ is the new data to replace the data currently associated with the token;
- $A.c \in \mathbb{N}_{64}$ is the transaction counter;
- $P.s \in \mathbb{OCT}^*$ is the input to satisfy token's data update predicate;
- $P.s_i \in (\mathbb{OCT}^*)^*$ is the input to satisfy the data update predicates inherited from the types.

Validity Condition

$$\begin{aligned}
&\Psi_{\text{updateNT}}(T, S) \equiv \\
&\quad \text{ExtrType}(T.\iota) = \text{ntoken} \ \wedge S.N[T.\iota] \neq \bot \wedge \\
&\quad S.N[T.\iota].D.\ell = 0 \wedge \\
&\quad T.A.c = S.N[T.\iota].D.c \wedge \\
&\quad \text{VerifyTxAuth}(S.N[T.\iota].D.\varphi_d, T, T.P.s) = 1 \wedge \\
&\quad \Pi(S.N[T.\iota].D.\iota_t)(T, T.P.s_i) = 1,
\end{aligned}$$

where

$$\Pi(\iota) = \begin{cases} () & \text{if } \iota = \bot \\ \Pi(S.N[\iota].D.\iota_p) \| S.N[\iota].D.\varphi_{\text{d}} & \text{otherwise.} \end{cases}$$

That is,

- ι identifies an existing non-fungible token,
- the token is not locked,
- the order contains the correct transaction counter value,
- the input s satisfies the token's data update predicate, and
- the input s_i satisfies the multipart predicate obtained by joining all the inherited data update predicate clauses along the type inheritance chain.

Actions

1. $N[T.\iota].D.dat \leftarrow T.A.dat$
2. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

6.2.13 Fee Credit Handling

The addFC (add fee credit), closeFC (close fee credit), lockFC (lock fee credit record), and unlockFC (unlock fee credit record) transactions are handled the same way as in the money partition.

6.2.14 Round initialization and completion

6.2.14.1 Round Initialization: $\text{RInit}_{\text{token}}$

1. Delete all fungible tokens with zero value and expired lifetime:
   1.1 Find all such tokens:
   $\mathcal{I} \leftarrow \{\iota : \text{ExtrType}(\iota) = \text{ftoken} \ \wedge \ N[\iota] \neq \perp \ \wedge \ N[\iota].D.v = 0 \ \wedge \ N[\iota].D.t_\ell < S.n\}$
   1.2 For each $\iota \in \mathcal{I}$: DelItem(ι)

6.2.14.2 Round Completion: $\text{RCompl}_{\text{token}}$

No transaction system specific completion steps.

6.3 Permissioned Mode

For some use cases, it may be desirable to support running a dedicated instance of the user-defined token partition in a permissioned mode where only pre-authorized parties can send transaction orders. In validator implementations that support such a permissioned mode, it is activated by defining an administrative authorization predicate φ_{adm} in the configuration of the partition. With the predicate defined ($\varphi_{\text{adm}} \neq \perp$), the partition will not accept the usual addFC, closeFC, lockFC, unlockFC transactions and will instead accept the setFC and delFC transactions defined below. The purpose of the setFC transaction is to permit a client to send transaction orders to the partition and the purpose of the delFC transaction is to revoke the permission.

In such a permissioned partition, it may additionally be desirable to turn off the usual fee accounting mechanism and handle validator remuneration in some other way. Even in the fee-less mode, regular transactions will still need to reference a valid fee credit record and contain a "fee authorization proof" for the purpose of authenticating the transactions as coming from authorized clients; also, the cost of executing a transaction will still be tracked, to prevent the evaluation of a malformed predicate from locking up validators; however, the transaction execution cost will not be subtracted from the balance of the fee credit record; this allows the presence of a fee credit record with a nominal balance to serve as an authorization for unlimited number of transactions, until the record is explicitly deleted to revoke the access.

Note that the fee-less mode is only an option for permissioned partition instances; public permission-less partitions are not intended to be operated in fee-less mode.

6.3.1 Set Fee Credit

Transaction order $T = \langle \alpha, \beta, \iota, \text{setFC}, A, M_C, P, s_f \rangle$, with $A = (v, \varphi, c)$, $P = (s)$, where:

1. $A.v \in \mathbb{N}_{64}$ is the credit amount;
2. $A.\varphi \in \mathbb{L}$ is the target fee credit record owner predicate;
3. $A.c \in \mathbb{N}_{64}$ is the transaction counter;
4. $P.s \in \mathbb{OCT}^*$ is the authorization proof.

Validity Condition

$$
\begin{aligned}
&\Psi_{\mathsf{setFC}}(T, S) \equiv \\
&\quad \mathsf{ExtrType}(T.\iota) = \mathsf{fcr} \ \wedge \ (\\
&\qquad (S.N[T.\iota] = \bot \ \wedge \ \mathsf{ExtrUnit}(T.\iota) = \mathsf{PrndSh}(\mathsf{ExtrUnit}(T.\iota), T.A.\varphi \| P.M_C.T_0)) \vee \\
&\qquad (S.N[T.\iota] \neq \bot \ \wedge \ T.A.\varphi = S.N[T.\iota].\varphi) \\
&\quad) \ \wedge \ (\\
&\qquad (S.N[T.\iota] = \bot \ \wedge \ T.A.c = \bot) \vee \\
&\qquad (S.N[T.\iota] \neq \bot \ \wedge \ T.A.c = S.N[T.\iota].D.c) \\
&\quad) \wedge \\
&\quad T.M_C.\iota_f = \bot \wedge s_f = \bot \wedge \\
&\quad \mathsf{VerifyTxAuth}(\varphi_{\mathsf{adm}}, T, T.P.s)
\end{aligned}
$$

That is,

- $T.\iota$ identifies a fee credit record (either new or pre-existing),
- if the target does not exist, the identifier must agree with the owner predicate,
- if the target exists, the owner predicate must match,
- the transaction order contains correct target counter value,
- there's no fee credit reference or separate fee authorization proof, and
- the transaction has been correctly authorized.

Actions

1. if $S.N[T.\iota] = \bot$:
   1.1 AddCredit$(T.\iota, T.A.v, T.A.\varphi, P.M_C.T_0)$
2. else:
   2.1 IncrCredit$(T.\iota, T.A.v, P.M_C.T_0)$

Note: Since the setFC transactions are issued by the owner of the partition, the fees are not subtracted from the credit amount (in contrast with the addFC transactions). If fees have not been disabled, then it is expected that reporting of fees earned for processing setFC transactions is handled in the usual way, but settlement is out of band.

6.3.2 Delete Fee Credit

Transaction order $T = \langle \alpha, \beta, \iota, \mathsf{delFC}, A, M_C, P, s_f \rangle$, with $A = (c)$, $P = (s)$, where:

1. $A.c \in \mathbb{N}_{64}$ is the transaction counter;
2. $P.s \in \mathbb{OCT}^*$ is the authorization proof.

Validity Condition

$$\begin{aligned}&\Psi_{\mathsf{delFC}}(T, S) \equiv \\ &\quad \mathsf{ExtrType}(T.\iota) = \mathsf{fcr} \ \wedge \ S.N[T.\iota] \neq \bot \ \wedge \\ &\quad T.A.c = S.N[T.\iota].c \ \wedge \\ &\quad T.M_C.\iota_f = \bot \ \wedge \ s_f = \bot \ \wedge \\ &\quad \mathsf{VerifyTxAuth}(\varphi_{\mathsf{adm}}, T, T.P.s) = 1\end{aligned}$$

That is,

- $T.\iota$ identifies an existing fee credit record,
- the transaction order contains correct target counter value,
- there's no fee credit reference or separate fee authorization proof, and
- the transaction has been correctly authorized.

Actions

1. $N[T.\iota].D.b \leftarrow 0$
2. $N[T.\iota].D.\varphi \leftarrow 0$
3. $N[T.\iota].D.c \leftarrow N[T.\iota].D.c + 1$

Note: Since the delFC transactions are issued by the owner of the partition, the fees are not subtracted from the credit amount (in contrast with the closeFC transactions). If fees have not been disabled, then it is expected that reporting of fees earned for processing delFC transactions is handled in the usual way, but settlement is out of band.

7 Alphabill Distributed Machine

7.1 Background

7.1.1 Definitions

Block is a set of transactions, grouped together for mostly efficiency reasons. At the shard level, a block is an ordered set of transactions + proofs: UC and shard certificate. Root Partition does not produce an explicit blockchain – its certificates are persisted as proofs within the shard ledgers.

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 shard validators and Root Partition validators operate in **rounds**. Roughly, a round is an attempt to produce a block.

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

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

System has one Root Partition 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 Root Partition $k > 3f$. 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 **transaction** (tx) orders (txo).

7.1.2 Scope

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

rules are provided.

7.1.3 Repeating Notation

n_r – round number of the Root Partition
$n_{\beta,\sigma}$ – round number of shard σ of partition β
$k_{\beta,\sigma}$ – number of validators in shard σ of partition β, $k_{\beta,\sigma} = |\mathcal{V}_{\beta,\sigma}|$
v – validator identifier, unique within the Alphabill System instance; set of a shard's validators is $\mathcal{V} = \{v_i\}_{i\in\{1,\ldots,k_\sigma\}}$
$v_l \leftarrow$ LEADERFUNC$(\cdot)$ – leader identifier for this block production attempt
h – state tree root hash
h' – previous state tree 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.

7.2 Partitions and Shards

7.2.1 Timing

A 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 block 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 transactions, the leader starts assembling his next block proposal, extending the latest block with a valid UC.

When timer `t1` runs out the leader stops accepting new transactions, finishes state updates and broadcasts a block proposal to followers and then sends Certification Request to the Root Partition. See Figure 15.

Root Partition 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. Block is finalized when its UC is embedded into the block. The retry mechanism is illustrated by Fig. 16.

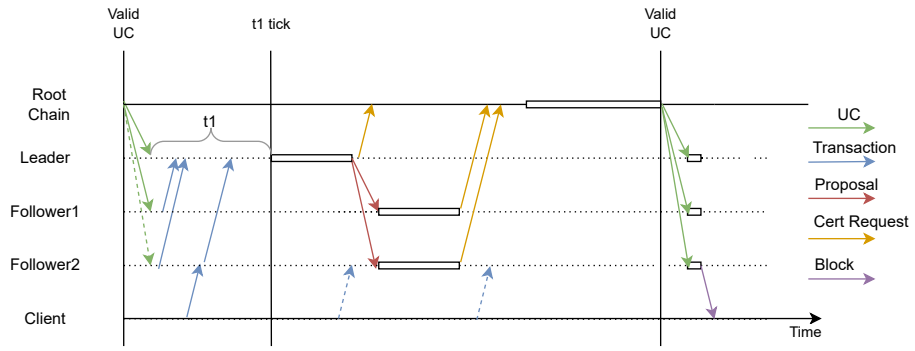

Figure 15. Successful Shard Round

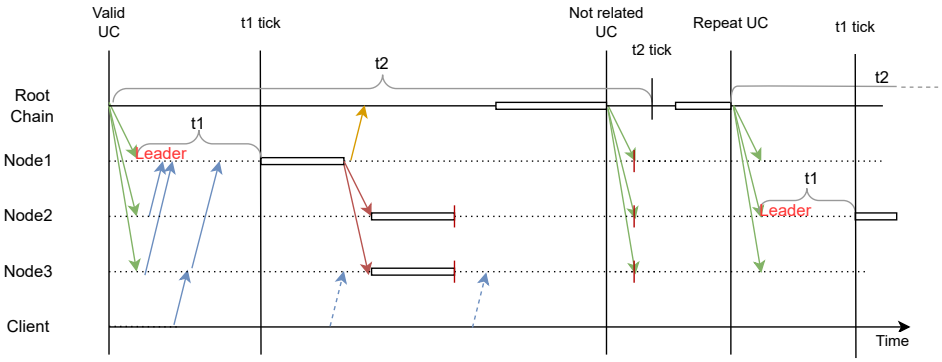

Figure 16. Shard Round attempt which did not produce a valid UC for this shard

7.2.2 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.
- Validator Identifier (ν). There are $k_{\beta,\sigma}$ validators in shard σ of partition β.
- Timeout value `t1`: after a validator sees a UC which appoints a leader, the leader waits for `t1` time units before stopping accepting new transaction orders and creating a block proposal.

Communication layer:

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

- communication addresses of the Root Partition validators

Data layer:

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

Variables:

- v_l: Current round leader's identifier; NULL if not known
- buf: buffer with pending transaction orders
- 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 block gets finalized.
- luc: latest UC. Importantly, this structure encapsulates the *state hash* of the last certified state, to be extended by the block production attempt.
- lte: latest Technical Record, certified by luc. Carries the required *round number*, leader identifier, epoch of the block production attempt.
- log: log of verified and executed (but not final) transactions; 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 transactions and round number as the time reference used for transaction 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 blockchain; append-only persistent shared ledger.

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 transaction(s)*: There are changes in the state tree; executed transactions are recorded in $proposal$ (that is, cp is the starting point and N and log are updated in sync). Processing of transactions 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 transactions 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 transactions 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
v: Node Identifier
$k_{\beta,\sigma}$: number of validators in the shard σ of partition β
$\mathcal{T}$: Unicity Trust Base, may *evolve*
Variables:
$v_l \leftarrow$ NULL: leader validator identifier of the current round; NULL if not known
$N \leftarrow \{\}$: State Tree
cp $\leftarrow \perp$: State Tree 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 transaction orders buffer
$log \leftarrow \{\}$: executed transaction log for proposal creation
$pr \leftarrow$ NULL: pending proposal corresponding to a CR request waiting for UC
sr: technical data payload, sent with Certification Request. See Statistical Record
$\mathcal{B}$: blockchain

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()
  $log \leftarrow \{\}$
  $pr \leftarrow \{\}$
  $v_l \leftarrow te.v_l$
  $luc \leftarrow uc, lte \leftarrow te$
  RESET_TIMER(t1)
  if $v_l = v$ **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 $\{v_i \mid i \leftarrow \text{integer}(H(uc)) \bmod k_{\beta,\sigma} + 1\}$ ▷ 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.

7.2.3 Subcomponents

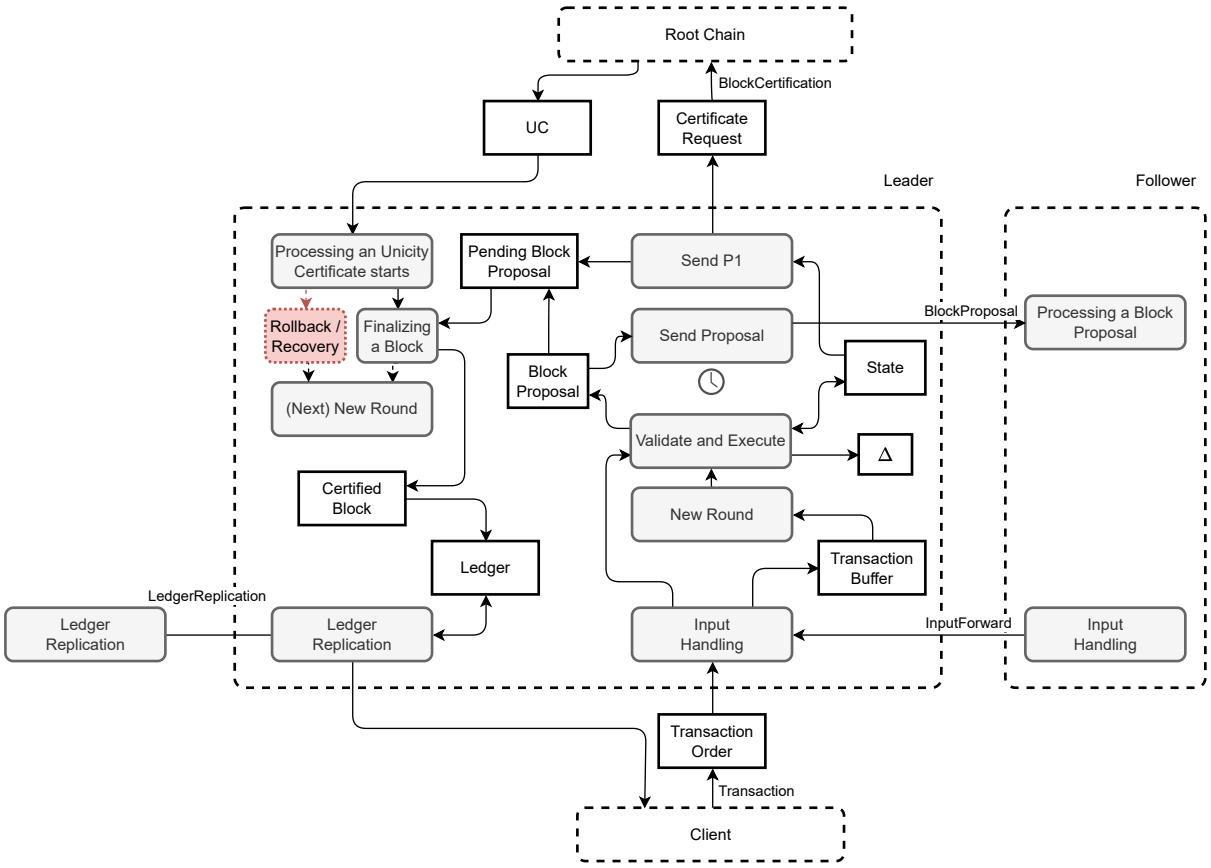

Figure 17. Data Flow of the Shard Leader Node

7.2.3.1 Input Handling

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

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

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

transaction orders. 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 transaction orders in a buffer and try again when the next leader is known and accepts transaction orders. If the validator is the current leader, then he processes available transaction orders 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 transaction orders from other validators.

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

Validators may limit the number of times a transaction order is forwarded.

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

7.2.3.2 Block Proposal

Summary: On clock tick, stop immediate validation and execution of incoming Transaction Orders. Validate and execute transaction orders from the Transaction Buffer, updating the State Tree (N) and log for proposal creation. Executed transactions 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 block must extend a previously certified block. 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 block, would be an unrecoverable error.

Please refer to Algorithm 3 for details.

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

7.2.3.3 Validation and Execution

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

Validating transaction orders is performing their full verification, according to Alphabill Platform Specification, section Valid Transaction Orders, and performing transaction system specific additional checks. The transactions must appear in the *proposal* in the same order. Transactions without interdependencies (i.e., affecting distinct units) can be executed in parallel. Invalid transactions are not executed and not included into produced proposal. There may be a receipt instead to justify the deduction of fees.

Algorithm 2 Input Handling

```
upon message <TransactionMsg | T> do
    if SANITY_CHECK(T) then
        if ν_l = ν then                                   ▷ this process is the leader
            PROCESS({T})                                  ▷ Beware of parallel execution
        else if ν_l ≠ NULL then                           ▷ someone else is the leader
            if ¬SEND_INPUTFORWARDMSG(ν_l, T) then         ▷ ν_l is the recipient
                buf ← buf ∪ T                             ▷ Store on failure
            end if
        else                                              ▷ Buffer transactions until leader is known
            buf ← buf ∪ T
        end if
    end if
end upon

upon message <InputForwardMsg | txs> do
    if ν_l = ν then                                       ▷ this process is the leader
        defer PROCESS(txs)
        return "accepted"
    else
        return "reject"
    end if
end upon

on event next_leader_elected do
    PRUNE_EXPIRED(buf)
    if ν_l = ν then                                       ▷ this process is the leader
        PROCESS(buf)                                      ▷ Beware of parallel execution
    else
        if SEND_INPUTFORWARDMSG(ν_l, buf) then            ▷ ν_l is the recipient
            buf ← {}                                      ▷ 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 ν_l = ν then                                       ▷ this validator is the leader
        ν_l ← NULL
        RCOMPL()                                          ▷ Transaction processing must have stopped by now
        sr ← PRODUCESTATISTICS()
        SEND_BLOCKPROPOSALMSG(α, β, σ, ν, luc, lte, log, sr)   ▷ Sign and Broadcast
        DO_CERT_REQ(log, ν, sr)
    end if
    ν_l ← NULL                                            ▷ leader stops accepting new txs
end on
```

Algorithm 4 Producing a Certification Request

function DO_CERT_REQ(txs, ν_l, sr)
  $h' \leftarrow luc.IR.h$, $n \leftarrow lte.n$, $e \leftarrow lte.e$, $t \leftarrow luc.C^{\mathsf{r}}.t$
  $h \leftarrow \textsc{StateRoot}(N)$, $V \leftarrow \textsc{DataSummary}(n, t, N, log)$, $f_B \leftarrow \textsc{TotalFees}(n, t, N, log)$
  $pr \leftarrow (n, e, h', h, t, \nu_l, log, sr)$ ▷ Pending Block Proposal
  $b \leftarrow ((\beta, \sigma, luc.IR.h_B, \nu_l); txs; \textsc{Null})$ ▷ temporary block
  $h_B \leftarrow$ BLOCK_HASH(b)
  if STORE_IN_DURABLE_WAY(pr) **then**
    $IR \leftarrow (n, e, h', h, V, t, h_B, f_B)$
    SEND_CR($\alpha, \beta, \sigma, \nu, IR, sr$) ▷ Sign and send
  else
    ▷ Do nothing as the current node can not guarantee data availability
    ▷ The round may get finalized though thanks to other shard validators
  end if
end function
function DO_CERT_REQ_AGAINST(sr) ▷ Voting against the proposal
  $h' \leftarrow luc.IR.h$, $n \leftarrow lte.n$, $e \leftarrow lte.e$, $t \leftarrow luc.IR.C^{\mathsf{r}}.t$
  $h \leftarrow 0_{\mathbb{H}}$, $V \leftarrow \perp$, $f_B \leftarrow 0$
  $h_B \leftarrow 0_{\mathbb{H}}$
  $IR \leftarrow (n, e, h', h, V, t, h_B, f_B)$
  SEND_CR($\alpha, \beta, \sigma, \nu, IR, sr$) ▷ Sign and send
end function

Algorithm 5 Validate and Execute Transaction Orders

function SANITY_CHECK($T; t \leftarrow lte.n$) ▷ Transactions will be fully validated later
  return $T.\alpha = \alpha \wedge T.\beta = \beta \wedge T.\sigma = \sigma \wedge T.M_C.T_0 > t$
end function

function VALIDATE($T; n \leftarrow lte.n, t \leftarrow luc.C^{\mathsf{r}}.t$)
  ▷ Omitted, see Platform Specification, Valid Transaction Orders
end function

function PROCESS(txs) ▷ Should be implemented as processing queue
  for all $T \in txs$ **do**
    if VALIDATE(T) **then**
      EXECUTE(N, T) ▷ Can be executed in parallel if non-related units
      $log \leftarrow log \cup T$ ▷ Retaining the ordering of input
    end if
  end for
end function

The expiration of Transaction Orders 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 block's $UC.IR.t$.

Refer to Algorithm 5 for details. Note that fee processing is omitted for brevity.

7.2.3.4 Processing an Unicity Certificate and Finalizing a Block

Summary: On receiving a UC, block 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:

$$uc.IR.h = uc.IR.h' \Rightarrow uc.IR.h_B = 0_{\mathbb{H}}$$

4. UC is checked for equivocation, that is, for arbitrary uc and uc', the following must hold:

$$\begin{aligned}
uc.IR.n = uc'.IR.n &\Rightarrow uc.IR = uc'.IR \\
uc.IR.h' = uc'.IR.h' &\Rightarrow uc.IR.h = uc'.IR.h \\
&\quad \vee uc.IR.h' = uc.IR.h \vee uc'.IR.h' = uc'.IR.h \\
uc.IR.h = uc'.IR.h &\Rightarrow uc.IR.h' = uc'.IR.h' \\
&\quad \vee uc.IR.h' = uc.IR.h \vee uc'.IR.h' = uc'.IR.h \\
uc.IR.h = uc'.IR.h &\Rightarrow uc.IR.h_B = uc'.IR.h_B \\
&\quad \vee uc.IR.h_B = 0_{\mathbb{H}} \vee uc'.IR.h_B = 0_{\mathbb{H}} \\
uc.IR.h_B = uc'.IR.h_B \neq 0_{\mathbb{H}} &\Rightarrow uc.IR = uc'.IR \\
uc.IR.n = uc'.IR.n + 1 &\Rightarrow uc.IR.h' = uc'.IR.h \\
uc.IR.n < uc'.IR.n &\Rightarrow uc.C^{\mathrm{r}}.n < uc'.C^{\mathrm{r}}.n
\end{aligned}$$

   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 block proposal, recovery is initiated, unless the state is already up-to-date with the UC.
7. Alternatively, if UC certifies the pending block proposal then block is finalized.
8. Alternatively, if UC certifies the block which 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. Block Finalization is presented in Algorithm 8.

On arbitrary timeout / *lost connection*: re-establish connection to the Root Partition.

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 unsaved block.

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 $uc.IR.h = pr.h \wedge uc.IR.h' = pr.h' \wedge uc.IR.h_B = pr.h_B$ **then**
      FINALIZE_BLOCK(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

7.2.3.5 Processing a Block Proposal

Summary: Upon receiving a BlockProposalMsg message, create a rollback checkpoint and then validate the signature and header fields, execute transaction orders from the proposal and updating the State Tree (N). Executed transactions 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. 18):

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 $uc.IR.h = uc.IR.h'$ **then** ▷ state does not change
    ensure($uc.IR.h_B = 0_{\mathbb{H}}$) ▷ ...then block is empty
  else ▷ state changes
    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 $uc.IR.h_B \neq 0_{\mathbb{H}} \wedge uc.IR.h_B = uc'.IR.h_B$ **then**
      ensure($uc.IR = uc'.IR$) ▷ non-empty block hash can only repeat in repeat UC
    end if
  end if
  if $uc.IR.n = uc'.IR.n + 1$ **then**
    ensure($uc.IR.h' = uc'.IR.h$)
  end if
  return 1
end function

Algorithm 8 Finalizing a Block

function FINALIZE_BLOCK(pr, uc)
  $B \leftarrow (\beta, \sigma, luc.IR.h_B, pr.l, pr.txs, uc)$
  ensure(BLOCK_HASH(B) $= uc.IR.h_B$)
  $\mathcal{B} \leftarrow \mathcal{B} \cup B$ ▷ Adding a new block to shard's blockchain
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 transaction orders in proposal are validated; on encountering an invalid transaction order 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 Root Partition—that is, positive vote for the proposal.

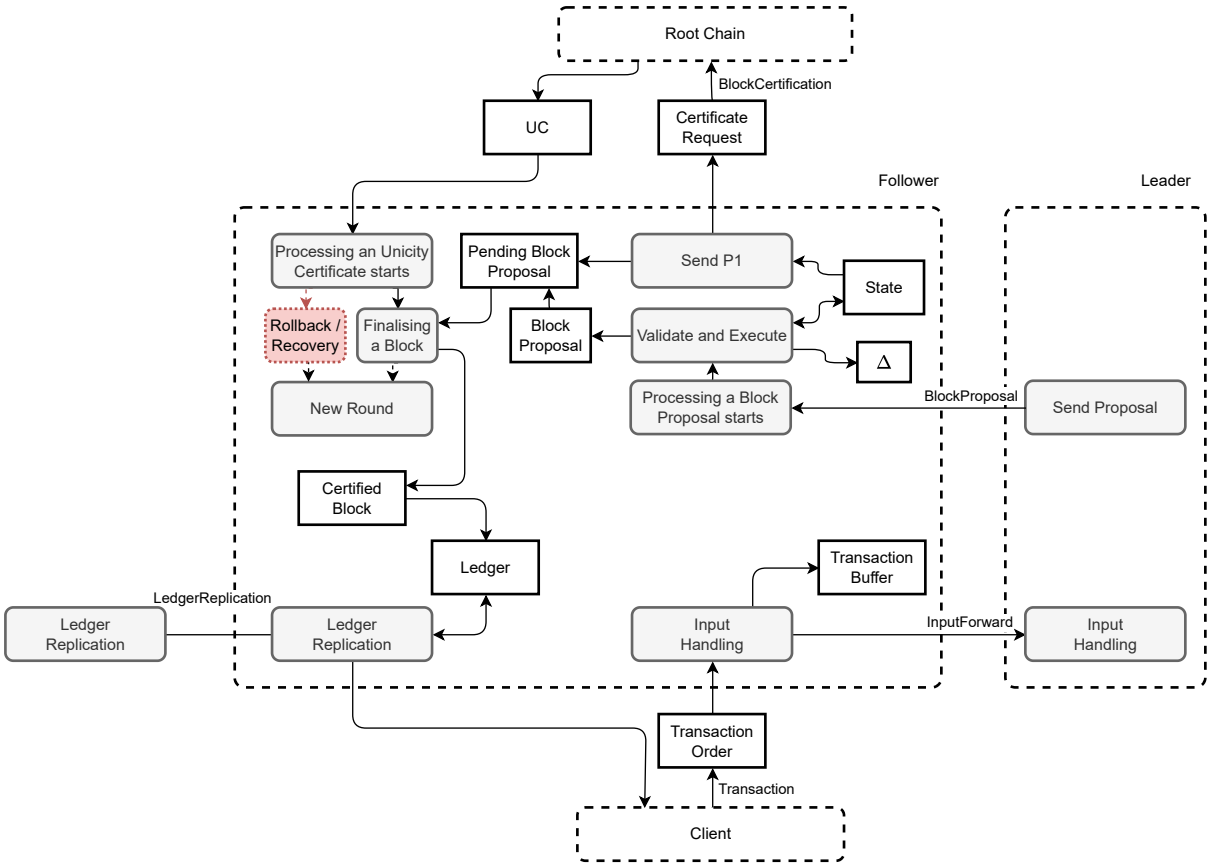

Figure 18. Data Flow of a non-leader shard validator

As an optimization, it is possible to vote against a proposal by sending a negative vote, where proposed state's hash is $0_{\mathbb{H}}$. This helps the Root Partition to determine that convergence to consensus is not possible before waiting until time-out.

Please refer to Algorithm 9 for details.

7.2.3.6 Ledger Replication

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

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

7.2.4 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 Root Partition.

Summary: Missing blocks 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 transactions of a not finalized block applied.

Algorithm 9 Processing a received Block Proposal

upon message <BlockProposalMsg | $m = (\alpha, \beta, \sigma, \nu, uc, te, txs, sr; s)$> **do**
  ensure(VALID(m)) ▷ Consistent and authorized message
  ensure(VERIFYUNICITYCERT(uc, T))
  ensure($uc.h_t = h(te)$)
  ensure($m.\nu = te.\nu_l$) ▷ 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) $= h' \wedge \{ \forall\ T \in m.txs \mid$ VALIDATE(T)$\}$ **then**
    cp $\leftarrow$ CHECKPOINT(N)
    RINIT()
    for all $T \in m.txs$ **do**
      EXECUTE(N, T)
    end for
    RCOMPL()
    DO_CERT_REQ($m.txs, m.\nu, sr$) ▷ 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

In more details:

1. Input UC is validated,
2. Missing blocks are fetched from other (random) validator(s),
3. Each block is verified: cryptographically using embedded UC, and for correct partition and shard ID;
4. Each transaction within the block is validated,

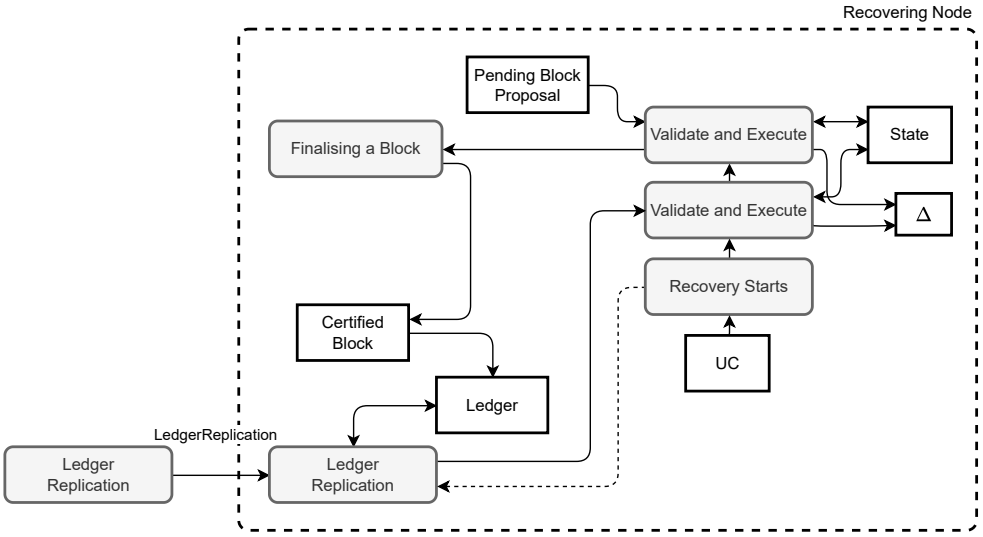

Figure 19. Data Flow of an out-of-sync (recovering) Shard Node

5. Transactions are applied to the state tree,
6. Last known UC is updated if a block has newer one.
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 block 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.

7.2.5 Protocols – Shard Validators

7.2.5.1 Protocol TransactionMsg – Transaction Order Delivery

Users deliver their transaction orders to one or more validators (to account for byzantine validators censoring or re-ordering transactions).

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

7.2.5.2 Protocol CR – Block Certification Request

This section extends Sec. 3.4.1, 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 Root Partition 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. A shard validator can have only one pending CR per round number; subsequent messages are ignored. A message with higher round number is always preferred.

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 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
```

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 block, that is, there is another set of Root Partition 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 a shard validator has reasons to suspect that Root Partition 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 shard validators to the Root Partition.

7.2.5.3 Protocol BlockCertificationResponse (CReS)

This section extends Sec. 3.4.2, 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 Root Partition to shard validators. 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: $\langle \texttt{CReS} \mid \alpha, \beta, \sigma, UC, \text{TE}, [\text{RootTrustBaseEntry}] \rangle$

7.2.5.4 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 block, that is, there is another set of Root Partition 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: $\langle \texttt{CReS} \rangle$

7.2.5.5 Protocol InputForwardMsg – Input Forwarding

Forward a set of transaction orders.

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

7.2.5.6 Protocol BlockProposalMsg – Block Proposal

Leader broadcasts its block proposal to other shard validators.

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

7.2.5.7 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 available block and every block can be

verified based on the Unicity Trust Base.

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

(Full) Clients, possibly in the role of helper service for light clients, use the same protocol to obtain blocks in order to provide their services.

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 blocks, either because the queried node is behind or it prefers to return blocks by smaller chunks.

7.3 Root Partition

7.3.1 Summary

Leader-based BFT consensus based SMR. Roughly, Root Partition validators:

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

7.3.2 Timing

Root Partition serves many partitions and shards, each implemented as a cluster of parallel validator machines. Operation cycles work like this:

1. Shards operate in parallel:
   - Shard validators send BlockCertification requests.
   - Once there is a quorum of requests for a shard, the Root Partition updates an entry in the array of input records.
2. *Eventually*, Root Partition 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 transaction systems. 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 block 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 Root Partition rounds fit into one shard round; $m \approx (t_r + t_p)/t_b$.

Let's denote cumulative (normal) distribution of BlockCertification 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. 20):
Start Root Partition 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.

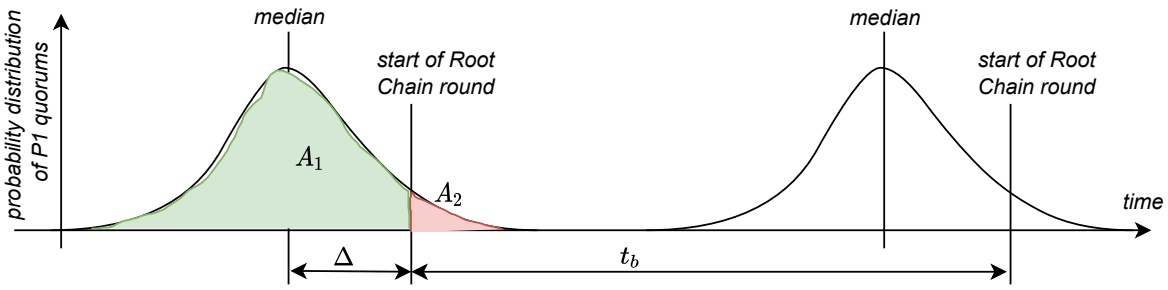

Figure 20. Optimal average latency is achieved when $\frac{A_1}{A_2} \approx \frac{t_b}{\Delta}$, where A represents the respective area (number of messages). $t_b - \Delta \approx t_r + t_p$.

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.

7.3.3 State

Please refer to Platform Configuration, State of the Root Partition, and Algorithm 12.

Configuration is provided by Orchestration and other management processes. State must be persisted and synchronized (recoverable from other Root Partition validators).

Time-outs and timers:

- `t2` – Root Partition waits for `t2` time units before re-issuing a UC for a shard. This triggers a retry of shard block creation, 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 Root Partition only); Target Block Rate (t_b) – Distributed Root Partition specific parameter

Communication layer:

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

7.3.4 Analysis

7.3.4.1 Safety

Root Partition enforces that each block is 'extended' by one block only. This excludes conflicting blocks (forks). Root Partition must not generate "equivocating" UCs (see Section 7.2.3.4 and Algorithm 7).

7.3.4.2 Liveness

Usual properties of a partially synchronous communication model apply.

7.3.4.3 Data Availability

If a validator issues a BlockCertification call then it must not lose the block proposal (transaction 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 Root Partition have issued a repeat UC, then all BlockCertification requests must be based on round number suggested by this response. Arriving and pending BlockCertification requests, referring to different round numbers, are dropped.

2. All BlockCertification 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 BlockCertification 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 a shard validator have issued multiple BlockCertification 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 shard validators must retain their produced blocks 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.

7.3.5 Monolithic Implementation

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

For a deployable implementation, please refer to Distributed Root Partition (Section 7.3.6); it provides the same functionality to shards without being a single point of failure, and can provide necessary level of decentralization.

7.3.5.1 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.

Algorithm 12 Global Parameters and Variables

Configuration:
α: Network Instance identifier
$\mathcal{T}$: Unicity Trust Base
$\mathcal{P}[]$: Partition Identifiers
$\mathcal{PD}[]$: Partition Description Records
State:
$n \leftarrow 0$: Root Partition's Round number
$e \leftarrow 0$: Root Partition'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 $\beta \in \mathcal{P}$
Variables:
$\forall\, \beta \in \mathcal{P}, \sigma \in \mathcal{SH}_\beta$: $\mathcal{IRT}_\beta[\sigma] \leftarrow (0_{\mathbb{H}})$
χ_β: the shard tree for every $\beta \in \mathcal{P}$ (Sec. 3.1.6)
υ: unicity tree (Sec. 2.9.4)
$changes[][] \leftarrow \perp$ Changes to IR, applied at the end of round, indexed by partition ID and shard number
$req[][][] \leftarrow \perp$: BlockCertification requests, indexed by partition ID, shard number and validator ID
$conn[][][] \leftarrow \perp$: Shard validator connections
Timers:
RESET_TIMER(t3)
$\forall\, \beta \in \mathcal{P},\ \forall\, \sigma \in \mathcal{SH}_\beta$: RESET_TIMER($t2_{\beta,\sigma}$)

7.3.5.2 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[2], increment Root Partition round number and reset the timer t3 which triggers the Unicity Certificate Generation (Algorithm 16).

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

Algorithm 13 CR Message Handling.

upon message <CR | $m = (\alpha, \beta, \sigma, \nu, IR, sr; s); context$> **do**
  ensure(VALID(m))
  ensure($m.\alpha = \alpha$) ▷ Right network id
  ensure($\beta \in \mathcal{P}$) ▷ Valid part. id
  ensure($\sigma \in \mathcal{PD}[\beta].\mathcal{SH}$) ▷ 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
    $\vee$ $(IR.h' = IR.h) \neq (IR.h_B = 0_{\mathcal{H}})$ ▷ No state change <=> blank
    $\vee$ $\neg \mathcal{PD}[\beta].\gamma(IR.v, \mathcal{PD}[\beta].\mathcal{V}[\sigma])$ ▷ Failed summary value check
    $\vee$ $IR.t \neq \mathcal{SI}[\beta, \sigma].UC_{-}.C^{\mathrm{r}}.t$ **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($\mathcal{PD}[\beta], IR$)))
  $req[\beta, \sigma][\nu] \leftarrow IR$ ▷ Add the new message
  $c \leftarrow \max_{r \in req[\beta][\sigma]} \sum_{p \in req[\beta][\sigma]} [r = p \wedge r.h \neq 0_{\mathbb{H}}]$ ▷ Number of the max. matching votes,
    ▷ ignoring $0_{\mathbb{H}}$, 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

7.3.6 Distributed Implementation

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

The messaging between a shard and the Root Partition is illustrated by Figure 21, where a shard with two validators ν_1 and ν_2 requests a UC. Root Partition is also depicted with two validators, the next leader after reaching a quorum of shard requests is ν_l^r.

7.3.6.1 Summary of Execution

The summary follows the processing flow of a Certification Request by a Root Partition validator. The flow is illustrated in Fig. 22; loosely moving counter-clockwise.

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 Root Partition
    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
```

Peer Node Selection In order to use the distributed root partition, shard validators must choose an appropriate subset of Root Partition validators to communicate with. The set must be shared by all shard validators 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 shard validator signatures which can be verified independently. UC responses must be checked for equivocation.

If a shard validator haven't received a UC-s from chosen Root Partition validators within $2 \times$ t2 then it sends SubscriptionMsg requests to random other Root Partition 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 Root Partition 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 Root Partition leader, using the Atomic Broadcast submodule.

Algorithm 16 Unicity Certificate Generation

on event `t3` **do** ▷ Simplified, see section “Timing”
  for all $\beta \in \mathcal{P}$ **do** ▷ Process partition timeouts
    for all $\sigma \in \mathcal{SH}_\beta$ **do**
      if $\neg changes[\beta, \sigma] \wedge$ EXPIRED_TIMER($\mathtt{t2}_{\beta,\sigma}$) **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 $\beta \in \mathcal{P}$ **do**
    $\chi_\beta \leftarrow$ CreateShardTree($\mathcal{SH}_\beta, \mathcal{IRT}_\beta$) ▷ Sec. 3.1.6.1
    $\mathcal{IH}[\beta] \leftarrow \chi_\beta(\sqcup)$
  end for
  $r \leftarrow$ CreateUnicityTree($\mathcal{P}, \mathcal{PD}, \mathcal{IH}$) ▷ Sec. 3.2.1
  $t \leftarrow$ GETUTCDATETIME()
  $C^{\mathsf{r}} \leftarrow$ CREATEUNICITYSEAL($\alpha, n, e, t, r_-, r; \mathsf{sk}_r$)
  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($\beta, \mathcal{P}, \mathcal{PD}, \mathcal{IH}$)
    $uc \leftarrow (\mathcal{IRT}_\beta[\sigma].IR, \mathcal{IRT}_\beta[\sigma].h_t, C^{\mathsf{shard}}, C^{\mathsf{uni}}, C^{\mathsf{r}})$
    $\mathcal{SI}[\beta, \sigma].UC_- \leftarrow uc$
    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($\mathtt{t2}_{\beta,\sigma}$)
  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

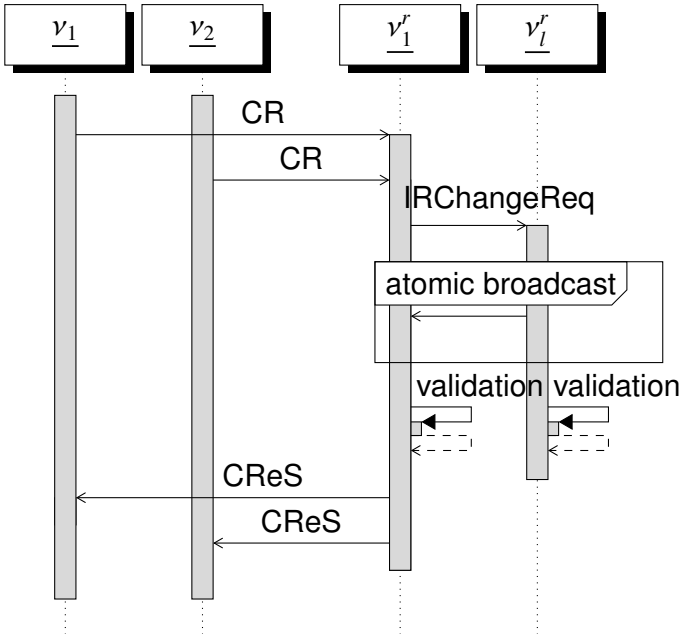

Figure 21. Message flow (simplified)

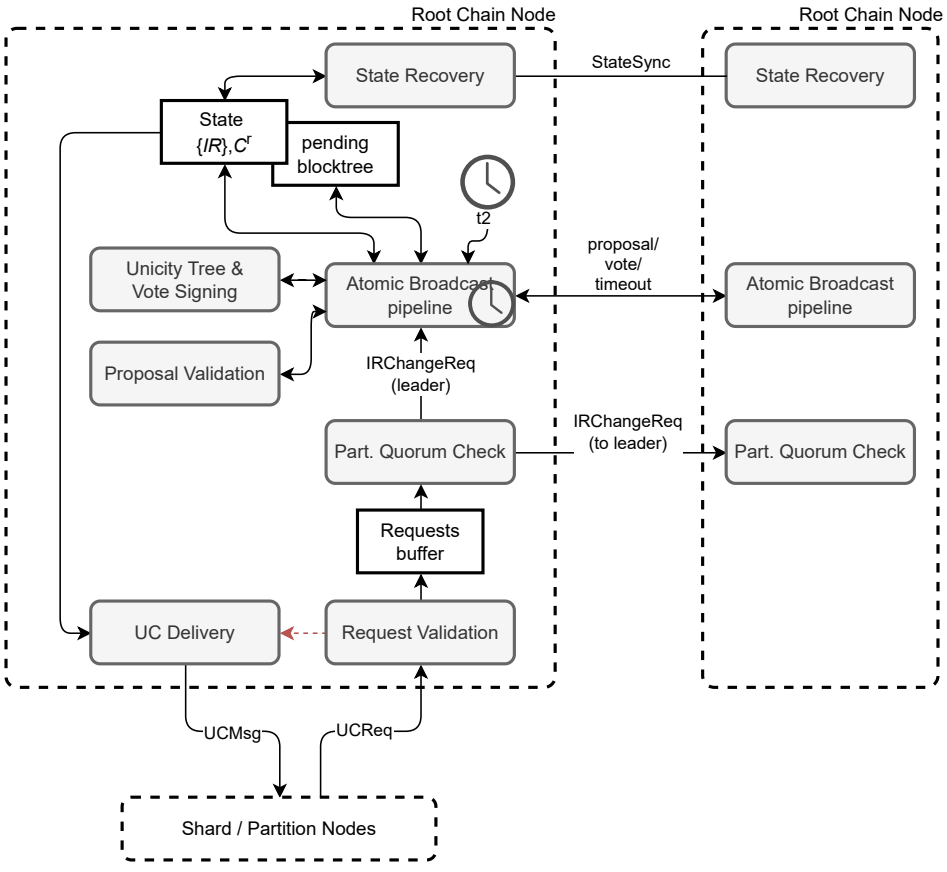

Figure 22. Data-flow of a Distributed Root Partition Validator

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.

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 Root Partition 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 a shard. More than half of the shard validators 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 7.3.6.2).

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 shard validators.

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).

7.3.6.2 Proposal

Proposal is a signed set of IR change requests, supplemented with proofs–the necessary number of signed shard validator 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 Root Partition validator must have up-to-date vector of Input Records of all shards. There is no persistent block storage. Returned Unicity Certificates, certifying IRs, are possibly persisted within shard blocks.

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.

7.3.6.3 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, QC – Quorum Certificate, TC – Timeout Certificate.

Rule 1. Voting Rule
$B.n >$ last vote round
$B.n = B.QC.n + 1 \vee (B.n = TC.n + 1 \wedge B.QC.n \geq max(\text{TC.tmo\_high\_qc\_round}))$

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 Root Partition Blockchain. However, Unicity Trust Base is somewhat blockchain-like: it gets a new entry added once per Root Partition 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 Root Partition should not tick faster than configured Target Block 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.

7.4 Dynamic System

This section describes mechanisms making Alphabill a dynamic system, by re-configuring partitions, shards and the Root Partition 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 Root Partition are not synchronized with each other, they have independent rounds and epochs.

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

7.4.1 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 shard validators		✓
2.	Adding, removing Root Partition 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 7.4.6.1)	✓	
+ 7.	RP's operation rules	✓	
8.	Shard's data structure version changes resulting in changes in block 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.

7.4.2 Root Partition Epoch Change

In order to facilitate a dynamic, responsive Root Partition, 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 Orchestration Process whose output is change records called Validator Assignment Records (Table 4).

Any Root Partition 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.

Table 2. Root Partition 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)

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.

Root Partition'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 7.4.6.6); and the new record gets propagated to Shard validators together with the next Unicity Certificate, as an extra field of the UCResp message.

Rule. Root Partition 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.

7.4.3 Shard Epoch Change

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

The next shard round after finalizing a block 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.

7.4.4 Controlling Shard Epochs

Root Partition triggers shard epoch changes. This happens in the following steps:

1. Root Partition validators obtain the Change Record for the next epoch of a shard.
2. If a Root Partition 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 Root Partition's validation rules: the next Certification Request must be presented by a valid quorum of next epoch's shard validators.

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.

7.4.5 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.

7.4.5.1 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 transactions
   On incoming client transaction:
   - If it is possible to determine the leader then forward client transactions to the leader.
   - Reject otherwise.
4. Obtain the latest UC for the shard (β, σ) from the Root Partition.
5. Launch recovery if behind
6. Subscribe to block feed of the shard (β, σ).
   On arrival of a new block:
   - Verify the block, apply the block 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 block 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 transactions
   On incoming client transaction:
   - If it is possible to determine the leader then forward client transactions to the leader.
   - Reject otherwise.
4. **loop:**
   - Obtain the latest UC from the Root Partition
   - 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 23.

7.4.5.2 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.

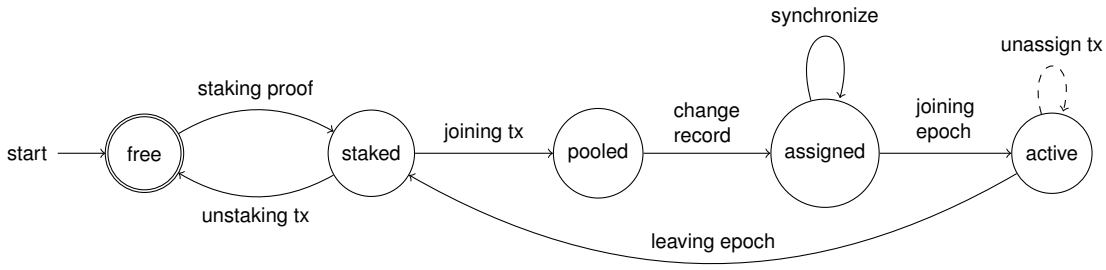

Figure 23. Node state graph (node's view)

2. Stop accepting proposals.
3. Stop accepting client transaction messages.
4. OK to reset the state.
5. Continue serving the block replication protocol; it is mandatory to retain the storage until the next quorum have committed at least one block.
6. OK to reset the block storage.

The node (more specifically the node operator) can initiate the leaving process by sending an "unassign transaction" to the Orchestration Partition. 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 Orchestration Partition to be assigned again as a validator, 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.

7.4.5.3 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.

7.4.5.4 Root Partition Node, joining

On receiving a change record:

1. Obtain the current Root Partition 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 Root Partition proposal messages.
   On receiving a proposal where ProposalMsg.BlockData.Epoch is incremented and the epoch includes the node:

- Use the Root Partition recovery protocol (GetStateMsg / StateMsg) to synchronize the state,
- Process the proposal,
- Start processing shard Certification Requests,
- Continue as full Root Partition validator.

7.4.5.5 Root Partition Node, leaving

A Root Partition 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 Root Partition node (more specifically, the operator of the node) can initiate the leaving process by sending a specific "unassign transaction" to the Orchestration Partition. Processing may take arbitrary time, and the unassignment may happen for other reasons as well. Root Partition validators are strongly encouraged to not drop off without executing the unassignment process.

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

7.4.6 Dynamic Data Structures

7.4.6.1 Versioning

The content of Unicity Seal and Unicity Trust Base, globally used data structures across the Alphabill Platform, depend on Distributed Root Partition implementation details. These data structures must be versioned to accommodate necessary iterative changes while the Root Partition implementation roadmap is being executed. That is, $\mathcal{T} = (v, \cdot)$ and $C^r = (v, \cdot)$, 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.

7.4.6.2 Evolving

Unicity Trust Base itself evolves (e.g., new records are added, while format/version stays the same) when the Root Partition 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[3] Unicity Trust Base as $\mathcal{T}_{\text{base}}$ and updated Unicity Trust Base as $\mathcal{T}$, and for each version of data structures define the function

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

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.

7.4.6.3 Monolithic, Static Root Partition

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

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

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

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

7.4.6.4 Monolithic, Dynamic Root Partition

In the case of dynamic Root Partition 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.

- $T_e = (\alpha, e, \text{pk}_e; s)$
  is Unicity Trust Base Record, where $s = \text{Sig}_{\text{sk}_{e-1}}(T_e)$ 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 Root Partition.
- $\mathcal{T} = (T_j, T_{j+1}, \ldots, T_k)$,
  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: $\forall\ i, j \colon T_i.\alpha = T_j.\alpha$.

[3] 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*.

- $C^{\mathsf{r}} = (\alpha, n_r, t_r, r_-, r; (s, e))$
  as defined in the Platform Specification; s is a cryptographic signature created using Root Partition'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 Alphabill 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 $T \in \mathcal{T}$ **do**
    if $\mathcal{T}_{\mathsf{base}}[bmax].\alpha \neq T.\alpha$ **then**
      return error ▷ non-invariant system id
    end if
  end for
  for $j \in \{tmin \,..\, bmax + 1\}$ **do** ▷ verify based on available trust base
    if $\mathsf{Ver}_{\mathcal{T}_{\mathsf{base}}[j-1].\mathsf{pk}}(\mathcal{T}[j], \mathcal{T}[j].s) = 0$ **then**
      return 0
    end if
  end for
  for $j \in \{tmin + 1 \,..\, \max_{T \in \mathcal{T}} T.e\}$ **do** ▷ verify consistency of new
    if $\mathsf{Ver}_{\mathcal{T}[j-1].\mathsf{pk}}(\mathcal{T}[j], \mathcal{T}[j].s) = 0$ **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.

7.4.6.5 Distributed, Static Root Partition

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

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

Unicity Trust Base does not change as the Root Partition configuration is static.

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 $\neg(\forall\, m, n \in 1 \,..\, |C^{\mathsf{r}}.s| \colon C^{\mathsf{r}}.s[m].i \neq C^{\mathsf{r}}.s[n].i)$ **then**
    return 0 ▷ Duplicate signers
  end if
  if $|C^{\mathsf{r}}.s| \leq \mathcal{T}.q$ **then**
    return 0 ▷ No quorum
  end if
  for $(i, s) \in C^{\mathsf{r}}.s$ **do**
    if $(\mathrm{Ver}_{\mathcal{T}.\mathsf{pk}_i}(C^{\mathsf{r}}, s) = 0)$ **then** ▷ calculated over all fields except signature
      return 0 ▷ Invalid signature
    end if
  end for
  return 1 ▷ Success
end function

7.4.6.6 Distributed, Dynamic Root Partition

The Unicity Trust Base Record is defined by Table 3.

Table 3. Unicity Trust Base Record of Dynamic Distributed Root Partition.

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	$\{\nu, b_\nu\}_e$	$\{(\mathbb{OCT}^*, \mathbb{N}_{64})\}$
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 Orchestration Change 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 7.4.6.4, with an additional requirement of using the required multi-party signature scheme.

The verification functions are as defined in 7.4.6.4, where the signatures are interpreted in broader sense as multi-party signatures, created by respective quorums of validators.

7.4.6.7 Signature Aggregation

This is an optimization of Distributed Root Partition 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[4], 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 by combining at least q partial signatures, and a bit-field identifying the signers. Partial signatures are created by individual Root Partition 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.

7.5 Root Partition Data Structures (illustrative)

Below is an informal illustration on how Alphabill data structures integrate to HotStuff consensus primitive; in ABNF format[5]. The structures document the distributed, dynamic Root Partition.

```
; defined above
  UC = IR TechRecordHash ShardTreeCertificate UnicityTreeCertificate UnicitySeal ; UC = (IR, h_t, C^shard, C^uni, C^r)
  IR = RoundNumber Epoch PreviousHash Hash SummaryValue Time BlockHash SumOfEarnedFees
       ; IR = (n, e, h', h, v, t, h_B, f_B)
  ShardTreeCertificate = ShardIdentifier *SiblingHash ; C^shard = (σ; h_1^s, ..., h_|σ|^s)
  UnicityTreeCertificate = PartitionIdentifier PartitionDescriptionHash *HashStep
  HashStep = PartitionIdentifier SiblingHash ; C^uni = (β, dhash; (β_2, h_2), ..., (β_ℓ, h_ℓ)))

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

; internals:
  LedgerCommitInfo = UnicitySeal
  QC = VoteInfo LedgerCommitInfo *Signatures

  VoteInfo = RoundInfo
  RoundInfo = RoundNumber Epoch Timestamp ParentRoundNumber CurrentRootHash

; rc messages:
```

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

[5] `https://tools.ietf.org/html/rfc5234`

```
  VoteMsg = VoteInfo LedgerCommitInfo HighQC Author Signature

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

  ; evolving trust base:
  RootTrustBase = Version *RootTrustBaseEntry
  RootTrustBaseEntry = NetworkIdentifier Epoch *(NodeID Pubkey Stake) QuorumThreshold StateHash
       ChangeRecordHash PreviousEntryHash *Signatures

  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

; rc helpers:
  GetStateMsg = NodeId ; id of the validator requesting the state
  StateMsg = *UC CommittedHead BlockNode RootTrustBaseEntry
  CommittedHead = BlockNode = RecoveryBlock
  RecoveryBlock = BlockData *InputData QC CommitQC
  InputData = PartitionIdentifier Shard IR Sdrh

; shard-rc messages
  CR = NetworkIdentifier PartitionIdentifier Shard NodeIdentifier IR BlockSize StateSize Signature
  ; Certification Request CR = ⟨α, β, σ, ν; ℐℛ, ℓ_B, ℓ_S; s⟩
  ; if prevStateTreeHash is already 'extended' with UC then return latest UC immediately.
  ; otherwise validation and cert.  generation continues, cert is returned once available.
  ; 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 [RootTrustBaseEntry]
       ; Certification Response

  StatisticalRecord = NumBlocks SumFees SumBlockSize SumStateSize MaxFee MaxBlockSize MaxStateSize
       ; SR = (n_e, f̄_B, ℓ̄_B, ℓ̄_S, f̂_B, ℓ̂_B, ℓ̂_S)
  FeeRecord = *(ValidatorIdentifier Fees) ; VF
  TechRecord = Round Epoch LeaderID SRHash VFHash ; TE = (n_r, e_r, ν_ℓ, h_sr, h_vf)

  ShardInfo = Round PreviousHash PrevStatisticalRecord StatisticalRecord *ValidatorIdentifier PrevFeeRecord
       FeeRecord LeaderIdentifier PrevUC ; (n, h_−, SR_−, SR, 𝒱, VF_−, VF, ν_ℓ, UC_−)

  ; 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
```

8 Orchestration

8.1 Introduction

By *orchestration*, we mean the operational management and re-configuration of a dynamic Alphabill Platform instance. Orchestration processes manage the validators, manage the lifecycle of partitions and govern sharding, manage incentives to the participants, and manage updates to the system. Orchestration executes the on-chain mechanisms like Proof of Stake and Tokenomics.

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

8.1.1 Orchestration 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 Alphabill 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 Alphabill as a continuously running, decentralized and permission-less system. In the final form, it relies on algorithmic, on-chain orchestration 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 Alphabill Foundation, and a gradual roadmap to full decentralization follows. While executing the roadmap, automated, on-chain orchestration processes, briefly described in the following sections, are taking over.

The orchestration is architected in a modular way. The Orchestration Partition is a plug-in replaceable component, possibly implementing different mechanisms, like Proof of Authority, Proof of Stake, Delegated Proof of Stake.

The Orchestration Partition implements a number of modular Orchestration Processes.

The partitions of Alphabill Platform are either fully managed by the Orchestration, or are

implemented as “permissioned partitions”: managed by a permissioned entity with administrative rights given by the partition lifecycle orchestration process.

8.2 Data Flow

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

8.2.1 Orchestration Partition

The Orchestration Partition is a usual partition within an Alphabill instance. It produces a ledger with blocks, recording transactions what we call *change records*. Change record formats of different orchestration processes are documented in the following subsections.

8.2.2 Configuration Agent

Orchestration Partition just saves the change records in its ledger. The partition nodes are also isolated processes, not connecting directly to other partitions and shards, except the Root Partition. In order to transport the relevant change records to concerned nodes, an independent process called “Configuration Agent” is used. An agent can be considered as a part of a logical validator instance.

Agent replicates Orchestration Partition blocks using usual block replication protocols, either directly or from a public or private RPC Node. Agent searches for relevant change requests. On encountering a relevant change request, it is pushed to validator’s partition node process over the Config API, see Fig. 24.

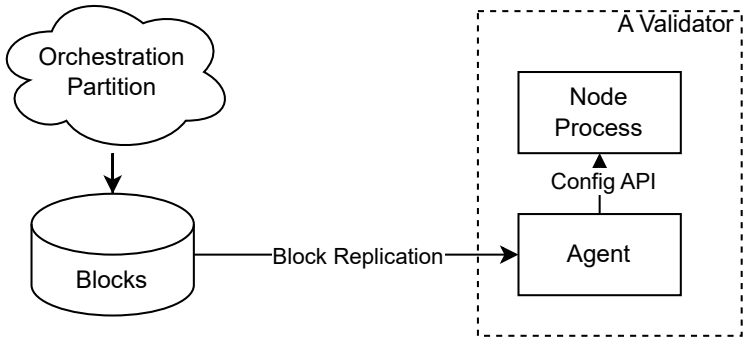

Figure 24. Orchestration Information Flow for Partition Nodes

Relevant change requests: change the set of validators in particular shard, change the Root Partition validators, change the parameters of particular partition and shard. If a validator hasn’t joined a shard, then a relevant record provides a future configuration of a partition and shard which the validator will be a member of, signaling that the validator must sync with the shard and join the validator set at a future epoch change. Conversely, when a validator identity does not appear as a member of a future validator set in the shard, then it must leave the active validator set at the required epoch change.

Node processes authenticate (at the transport level, e.g. localhost access only) and trust their configuration agents; nevertheless, change records must be validated using trans-

action execution proofs, based on pre-agreed identity of the Orchestration Partition and Unicity Trust Base. Agents can be shared within trusting entities.

8.2.2.1 How a Validator joins a Partition

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

- Node Agent detects a new Change Record it is involved with, learns the partition identifier and shard number
- Node Agent obtains the current epoch number from the shard
- Node Agent obtains the Change Records from current to the joining one (if not known)
- Node Agent activates the node process by making config API call providing necessary information
- Node continues as described in Section 7.4.5.1.

8.2.3 Permissioned Partitions

A Permissioned Partition is fully owned and controlled by a centralized entity. The entity identifier is part of the partition configuration. The entity can freely choose partition parameters, including the validator identities, implementing the “bring your own validator” scheme.

Permissioned Partitions are still integral parts of the Alphabill Framework and request Unicity Certificates from the Root Partition. In order to do so, a permissioned partition must:

- provide the identities of validators to the Orchestration Partition, so that Root Partition can fetch this information and authenticate UC Requests, in the form of transactions with change requests,
- fetch the Root Partition configuration and keep the node processes updated,
- present proofs of Root Partition incentive contributions (fees) to the respective Orchestration Process.

Permissioned Partition management can be implemented by a centralized configuration management system (automation platform) which connects to node Config API directly, see Fig. 25.

A Permissioned Partition may keep its ledger private.

It is up to the permissioned partition instance to decide if, how and how much to charge from its users. The maintainer of a permissioned partition must have a fee deposit to pay for Alphabill platform services.

8.2.4 Root Partition

The Root Partition plays active role in enforcing and synchronizing the Orchestration. 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 change requests.

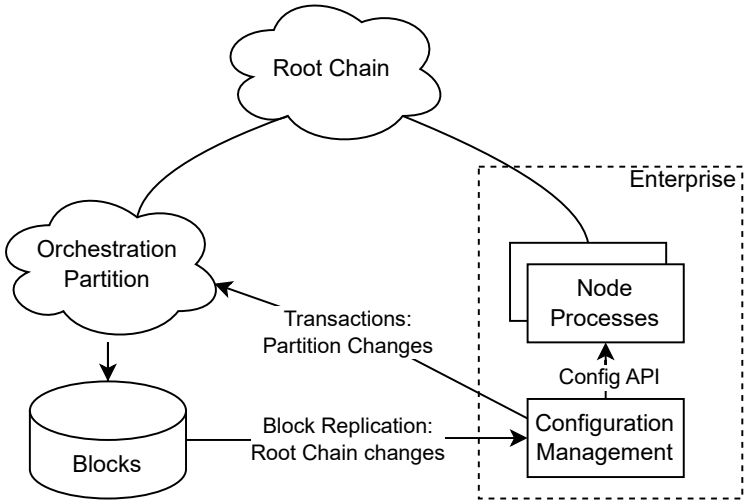

Figure 25. Orchestration Information Flow for Permissioned Partition

The recommended way is to allow every active Root Partition Validator (as an entity) to be a validator in the Orchestration Partition, by running two node processes and facilitating direct information flow.

8.3 Orchestration Mechanisms

8.3.1 Proof of Authority

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

8.3.2 Proof of Stake

Proof of Stake mechanisms provide decentralization, permissionlessness and *Sybil attack*[6] protection.

Proof of Stake mechanism is based on the fact that there is a finite amount of certain backing asset, in Alphabill's case it is the native currency, ALPHA. When each validator 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 Alphabill 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 (Section 8.4.4).

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

8.3.3 Tokenomics Toolbox

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

Orchestration 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 Alphabill 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.

8.4 Orchestration Processes

On-chain Orchestration is executed by the Orchestration Partition, and it is divided logically to a set of independent processes. Following subsections detail the interface between Alphabill Platform, the Orchestration and Governance: expectations on Orchestration Processes and their output data structures. This fixed interface allows Orchestration Partition to be a plug-in replace component, for example to use either Proof of Authority or Delegated Proof of Stake implementations; or to change tokenomics mechanisms.

8.4.1 Validator Assignment

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

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

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}	$\mathbb{H}$

Switching Condition may be a Root Partition 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 Root Partition, $k > 2/3 \sum b_v$. 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.

8.4.2 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.

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.	Cluster Size	k	$\mathbb{N}_{32}$
8.	Target Block Rate	$\texttt{t}$	$\mathbb{N}_{32}$(ms)
9.	Time-outs	$\texttt{t2}, \texttt{t3}$	$\mathbb{N}_{32}$(ms)
10.	Hash of previous record	h_{e-1}	$\mathbb{H}$

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

8.4.3 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 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 Root Partition. 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 shard validators.

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

8.4.4 Incentive Payouts

This process makes payments to Alphabill Validators. The payouts are correlated with the quality and quantity of provided services, collected fees of the particular partition, the platform-wide impact ("common good" factor), and the amount of locked stake. Payouts correlate negatively with unwanted behavior: instability, not following the ledger rules, not following the orchestration and governance decisions, and more severely in the case of equivocation or other acts with malicious intent.

The process should encourage stability and prefer on-chain data (data with cryptographic proofs) for payout calculation. For example: only operational validators can become block proposers (executed by the leader election algorithm; probability depends on stake amounts), and every successful block proposal (as seen in block headers) earns a credit unit for the proposer. A valid "fraud proof" (e.g., two conflicting signed messages from a validator) resets the credit and may expel the validator at the next epoch (executed by Validator Assignment). Node reputation and "tokenomics" are discussed elsewhere.

Payouts are not immediate—they are delayed by few epochs.

The Input data is collected by the Root Partition. It maintains a rolling "earned fee" amount for every validator of every shard for the ongoing shard epoch; and keeps the cumulative amount of the previous epoch. The rolling amount is incremented for the partition's leader by the amount of collected fees when an UC of non-empty block gets issued.

Each shard validator (or the "representative", i.e., validator's agent process) is responsible for fetching the record certifying the previous epoch's total fees collected by blocks proposed by the validator, and submitting it as a transaction to the Orchestration Partition. The cumulative fee record is certified by any UC of the following epoch.

Orchestration Partition then runs the Incentive Payout process, generating transaction orders unlocking reward payouts. Shard validators are responsible for fetching the transaction orders and submitting them to the Money Partition.

Incentive Payout process takes into consideration possible fraud-proofs, submitted by anyone before a deadline a few epochs later when payout transaction orders are generated.

8.4.5 Gas Rate Multiplier

The process updates periodically the Gas Rate Multiplier value, which provides relatively constant fees (as measured in external reference currencies). This absorbs possible fluctuations of the ALPHA exchange rate; without the overhead of maintaining a "stablecoin" for the fee payments.

8.4.6 Software and Version management

This process manages software updates, and thereby possible changes in the ledger rules. The output helps to coordinate possibly compatibility-breaking changes, up to the precision of a Root Partition round number when the switch happens.

8.4.7 On-chain Governance

In order to approve arbitrary changes expressible in human language, a stake-weighted, in-person voting mechanism is implemented (elsewhere called "coinvote"). The decisions become binding to the Foundation and/or members of the Alphabill community.

8.5 Proof of Authority Orchestration Partition Type

8.5.1 Summary

This is the definition of Proof of Authority (PoA) Orchestration Partition. The configuration of the system is defined and managed by an authorized party, recorded by the Orchestration Partition, and executed by the dynamic Alphabill Platform.

The partition is an append-only database of Validator Assignment Records (Table 4).

8.5.2 Motivation and General Description

Each unit in the Orchestration Partition represents a partition or a shard. Each such unit of type var specifies the configuration for its corresponding shard or its corresponding partition without shards. The units get automatically created by the first transaction that updates their value.

The units have one transaction type (addVar) which appends a configuration for one epoch. The epoch numbers start at 0 and increase strictly by 1 for each unit in each successive invocation of the addVar transaction.

All units have the same fixed owner predicate φ_{orc}. For example, the predicate could assign the ownership to a permissioned entity. No fees are applied to the transactions and there are no fee authorization proofs.

The units of type var store little specific data. Instead, parties interested in validator assignment records should read the transaction processing results of transactions of type addVar which work on the units. There is no transaction content validation.

Every such transaction invocation specifies a new epoch configuration for a single shard or a partition with no shards; the configuration includes a new epoch start condition and a new validator list for the epoch. The assignment of a particular validator is revoked by future transaction which specifies a new decision that does not include the validator into the same partition/shard any more. If a shard or partition stops working (for example, when a shard gets split into new shards), the stopped entity will have a validator assignment record g with $g.v_e = \bot$.

As validators need to configure themselves according to this information, it is expected that the unit owner submits the information to the blockchain early enough to give validators ample time to reconfigure and synchronize their state before the epoch switching condition is reached.

The genesis block B_0 of the orchestration partition contains the first of such transactions of type addVar for all active partitions and shards, listing the initial state as a validator assignment record of type $\mathbb{VAR}$.

8.5.3 Specification of the Orchestration Partition

8.5.3.1 Parameters, Types, Constants, Functions

System type identifier: st $= 4$

Partition identifier: $\beta_{\text{orc}} = 4$

Type and unit identifier lengths: tidlen $= 1$, uidlen $= 32$

Summary value type $\mathbb{V}$: $\bot$

Summary trust base: $\mathcal{V} = \bot$

Summary check: none

Constants: φ_{orc} – fixed owner predicate of type $\mathbb{L}$

Unit types: $\mathcal{U} = \{\text{var} = 1\}$ (validator assignment records)

Unit data:

- $\mathbb{D}_{\text{var}}$: tuples (e) where e is the epoch number from the validator assignment record of the previous transaction order on the same unit.

For every active shard and every active partition without shards there is always exactly one unit of type var with a constant (fixed) owner predicate. Unit identifier ι is calculated as a hash of the partition identifier and shard identifier: $\iota = \text{NodeID}(\text{var}, H(\beta, \sigma))$; if the partition has no shards, the shard identifier is assumed to be an empty string for this calculation ($\sigma = \lfloor\rfloor$).

A validator assignment record VAR describes an epoch configuration for a shard or a partition without shards. The record is of type $\mathbb{VAR}$ which is a tuple (e, n_r, v_e), where:

- e – epoch number of type $\mathbb{N}_{64}$

- n_r – epoch switching condition: root partition round number of type $\mathbb{N}_{64}$. The epoch starts when the unicity seal on the given block on the given shard contains a root partition round number that is equal to or bigger than the specified number: $C^r.n_r \geq n_r$
- v_e – validator assignment of type $\mathbb{VA} \cup \{\bot\}$. If and only if the shard or partition without shards does not work in this epoch, the following condition must hold: $v_e = \bot$.

In the future, the type of the epoch switching condition n_r can be made more generic. It could come to represent a suggested or enforced time or round number, or an arbitrary predicate; the implementation can work as a black box. For example, a switching condition could be a Root Partition round number range with soft enforcement – the responsible validators will not receive any fees for their work outside the expected epoch.

The validator assignment type $\mathbb{VA}$ defines a tuple $(\{v, b_v\}_e, k_e)$, where:

- $\{v, b_v\}_e^{\geq 1}$ – validator identifiers and stakes of type $\{(\mathbb{OCT}^*, \mathbb{N}_{64})\}$. Validator identifiers v refer to their public keys they sign validation messages with and stakes b_v refer to the amount of Alpha tokens that are staked for them.
- k_e – quorum size, measured in total amount of stake behind validator votes to reach consensus. For the Root Partition, $k > 2/3 \sum b_v$. For partitions / shards, $k > 1/2 \sum b_v$. The value is represented in Alpha tokens, typed as $\mathbb{N}_{64}$.

Transaction types: $\mathbb{T} = \{\text{addVar} = 1\}$ (add a validator assignment record)

8.5.3.2 Transactions

Add a Validator Assignment Record Records a new Validator Assignment Record for a shard or a partition without shards. The input parameter $A.g$ of type $\mathbb{VAR}$ is simply recorded as a transaction processing result.

Transaction order $T = \langle \alpha, \beta, \iota, \text{addVar}, A, M_C, P, s_f \rangle$ with $A = (g)$, $P = (s)$, where:

- $A.g \in \mathbb{VAR}$ is the new validator assignment record;
- $P.s \in \mathbb{OCT}^*$ is the owner proof.

The transaction record of type $\mathbb{TR}$ contains the same data as the transaction order T and additionally includes the server metadata M_S with the following constraints:

- $M_S.f_a = 0$ – actual fee charged is always 0.
- $M_S.R = A.g$ – the argument $A.g$ is recorded as the transaction processing result.

Transaction-specific validity condition:

$$
\begin{aligned}
&\psi_{\text{transB}}(T, S) \equiv \\
&\quad T.\iota = \text{NodeID}(\text{var}, H(T.A.g.\beta, T.A.g.\sigma)) \ \wedge \ (\\
&\qquad (S.N[T.\iota] = \bot \ \wedge \ T.A.g.e = 0) \vee \\
&\qquad (S.N[T.\iota] \neq \bot \ \wedge \ T.A.g.e = S.N[T.\iota].D.e + 1) \\
&\quad) \wedge \\
&\quad \text{VerifyTxAuth}(\varphi_{\text{orc}}, T, T.P.s) = 1
\end{aligned}
$$

That is,

- $T.\iota$ identifies a unit of type var, either new or pre-existing, and matches the partition and shard,
- if it is a freshly created unit, the epoch number will be 0,
- if the unit exists, the new epoch number is incremented by 1 from the previous valid transaction with the unit, and
- the input s satisfies the predefined authority predicate.

Actions $\text{Action}_{\text{addVar}}$:

1. **if** $N[\iota] = \perp$ **then**: AddItem$(\iota, (0))$
2. $N[T.\iota].D.e \leftarrow T.A.g.e$

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: $\{00 \prec 0 \prec 01 \prec \sqcup \prec 10 \prec 1 \prec 11\}$.

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 $\sqcup$) 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 $c||0, c||1 \in \overline{\mathcal{C}}$.

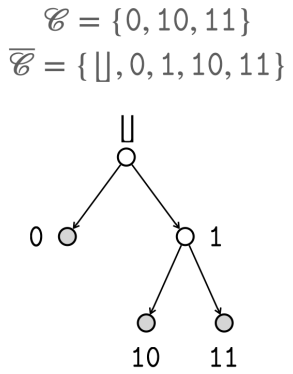

Figure 26. Irreducible prefix free code C and its closure $\overline{C}$.

B Encodings

B.1 CBOR

Most data structures on the Alphabill platform are serialized in the Concise Binary Object Representation (CBOR), as defined in the IETF RFC 8949[7].

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 Bit-strings

Bit-strings are not directly supported in CBOR. On the Alphabill 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.3 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.

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

B.4 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.5 Cryptographic Algorithms

“ECDSA” denotes the Elliptic Curve Digital Signature Algorithm using P-256 (secp256k1) curve, specified by NIST FIPS 186-4.

“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_i are the sibling hash values on the path from the i-th leaf to the root and b_i indicate whether the corresponding y_i 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_i are the sibling hash values on the path from the i-th leaf to the root and b_i indicate whether the corresponding y_i 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.

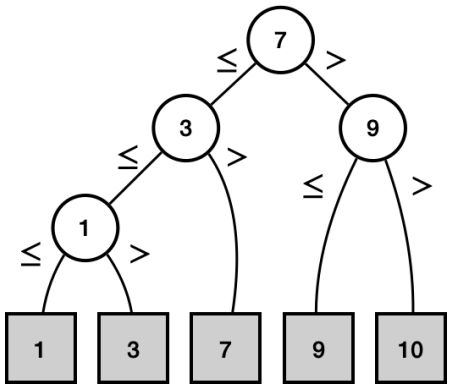

Figure 27. Keys of the nodes of an indexed hash tree.

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_i are the keys in the nodes on the path from the leaf to the root and y_i 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_i are the keys in the nodes on the path from the leaf to the root and y_i 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.

D State File

State file consists of the following components:

- Header
- List of Node Records
- Checksum

D.1 Header

State file header consists of the following components:

- α – network identifier of type $\mathbb{A}$
- β – partition identifier of type $\mathbb{P}$
- σ – shard identifier of type $\{0,1\}^{\mathcal{SH}.k}$
- $\mathcal{T}$ – unicity trust base of type $\mathbb{UB}$
- $\mathcal{PD}$ – partition description records of type $\mathbb{PD}[\mathbb{P}]$ for all registered partitions (including $\mathcal{PD}[\beta]$)
- UC – unicity certificate for the round from which the state tree was exported
- m – the number of Node Records of type $\mathbb{N}_{64}$

D.2 Node Record

Node Record consists of the following components:

- ι – unit identifier, of type $\mathbb{I}$
- D – unit data, of type $\mathcal{PD}[\beta].\mathbb{D}$
- x – state hash, of type $\mathbb{H}$
- $\langle(b_1, y_1), \ldots, (b_m, y_m)\rangle$ – the hash chain linking D and x to the root of the unit tree, where b_i are of type $\mathbb{B}$ and y_i of type $\mathbb{H}$
- `hasLeft` – existence of left child, of type $\mathbb{B}$ (1-exists, 0-does not exist)
- `hasRight` – existence of right child, of type $\mathbb{B}$ (1-exists, 0-does not exist)

Note that D and x are the final values at the end of the round and $\langle(b_1, y_1), \ldots, (b_m, y_m)\rangle$ links them to the root hash of the unit tree as of at the end of the round. The earlier states of the unit would be pruned as the first step of the next round, so these are omitted from the state file. The hash chain is extracted as specified for CreateUnitTreeCert$(\iota, |N[\iota].S|, N)$ in Sec. 2.9.1.1.

D.3 Checksum

Checksum of type $\mathbb{N}_{32}$ is the CRC32 of all contents (except the Checksum itself)

D.4 Writing (Serialization) Algorithm

For the serialization, given as input the state S, the following calls are made:

1. `writeheader` – writes out the file header based on the state S
2. `traverse`$(S.\iota_r)$ – traverses the state tree, starting from the root, and writes out the node records
3. `addchecksum` – computes and writes out the checksum

The function `traverse`(ι) is defined as follows:

if $\iota \neq 0_{\mathbb{I}}$ **then**
  `traverse`$(N[\iota].\iota_L)$
  `traverse`$(N[\iota].\iota_R)$
  `writenode`$(N[\iota])$
end if

where `writenode`$(N[\iota])$ writes down a Node Record R with R.`hasLeft` $= (N[\iota].\iota_L \neq 0_{\mathbb{I}})$, R.`hasRight` $= (N[\iota].\iota_R \neq 0_{\mathbb{I}})$, $R.x = N[\iota].S_{|N[\iota].S|}.x$, and $\langle (b_1, y_1), \ldots, (b_m, y_m) \rangle$ as specified in Sec. 2.9.1.

D.5 Reading (Deserialization) Algorithm

The function $S \leftarrow$ `readstate`(File) is defined as follows (using N instead of $S.N$):

$H \leftarrow$ readHeader(File)
$S \leftarrow$ NewState(H)
while $R \leftarrow$ readItem(File) **do**
  $\iota \leftarrow R.\iota$
  $N[\iota] \leftarrow$ NewNode(R)
  if R.hasRight **then**
    $(N[\iota].\iota_R, h_R) \leftarrow$ pop()
  else
    $(N[\iota].\iota_R, h_R) \leftarrow (0_{\mathbb{I}}, 0_{\mathbb{H}})$
  end if
  if R.hasLeft **then**
    $(N[\iota].\iota_L, h_L) \leftarrow$ pop()
  else
    $(N[\iota].\iota_L, h_L) \leftarrow (0_{\mathbb{I}}, 0_{\mathbb{H}})$
  end if
  $N[\iota].V \leftarrow (S.\mathcal{PD}[\beta].F_S)((S.\mathcal{PD}[\beta].V_S)(N[\iota].D), N[N[\iota].\iota_L].V, N[N[\iota].\iota_R].V)$
    ▷ Compute pre-pruning value of the sub-tree summary hash
  $h_s \leftarrow$ PLAIN_TREE_OUTPUT$(\langle (b_1, y_1), \ldots, (b_m, y_m) \rangle, H(R.x, H(D)))$
  $h \leftarrow H(\iota, h_s, N[\iota].V; h_L, N[N[\iota].\iota_L].V; h_R, N[N[\iota].\iota_R].V)$
  push$((\iota, h))$
    ▷ Compute post-pruning value of the sub-tree summary hash

$N[\iota].S \leftarrow \langle(\bot, R.x, N[\iota].D)\rangle$
$N[\iota].h_s \leftarrow$ PLAIN_TREE_ROOT$(\langle H(R.x, H(N[\iota].D))\rangle)$
$N[\iota].h \leftarrow H(\iota, N[\iota].h_s, N[\iota].V; N[N[\iota].\iota_L].h, N[N[\iota].\iota_L].V; N[N[\iota].\iota_R].h, N[N[\iota].\iota_R].V)$
end while
$(S.\iota_r, h_r) \leftarrow$ pop()
assert VerifyUnicityCert$(H.UC) \ \wedge \ h_r = H.UC.IR.h \ \wedge \ N[S.\iota_r].V = H.UC.IR.v$
return S

Here, the functions used by readstate are as follows:

- $H \leftarrow$ readHeader(File) – reads the header from the state file.
- $S \leftarrow$ NewState(H) – stores the corresponding values from H as components of the state S.
- $R \leftarrow$ readItem(File) – reads a node record R from the state file. It also indicates whether there are any more node records in the file. The while-loop can be replaced with a for-loop based on the m parameter of the header.
- $N[\iota] \leftarrow$ newNode(R) – creates a new node $N[\iota]$ and sets its data fields according to the existing fields of the node record R.
- push, pop – standard stack operations, assuming that the stack is empty in the beginning. In this specification, stack elements are of type $\mathbb{I}$. In program code, stack elements can be pointers to nodes.

Index